JAVA FOR THE STUDY OF EVOLUTION

VOLUME II v2

EVOLUTION AS SOFTWARE DEVELOPER

José del Carmen Rodríguez Santamaría
The EvolJava Community

Contents

Preface v

Introduction vii

1 Starting up 1
- 1.1 Installing Java . . . 1
- 1.2 Purposes and projects . . . 2
- 1.3 Our first program . . . 4
- 1.4 Review . . . 5

2 The core of evolution 7
- 2.1 Describing evolution . . . 7
- 2.2 Explaining complexity . . . 12
- 2.3 Better than evolution . . . 48
- 2.4 A new weasel . . . 51
- 2.5 Conclusion . . . 60

3 The genome is software 63
- 3.1 General terminology . . . 63
- 3.2 The ribosome is a programmable computer . . . 64
- 3.3 Software and dynamical systems . . . 66
- 3.4 Conclusion . . . 67

4 Evolution as software developer 69
- 4.1 The worm . . . 69
- 4.2 Evolution in perspective . . . 75
- 4.3 Getting free . . . 105
- 4.4 Enabling evolution . . . 117
- 4.5 Conclusion . . . 147

5 Evolution with crossing over **149**
5.1 Recombinant operators 149
5.2 Conclusion 176

6 Accelerating evolution **177**
6.1 Local maxims 177
6.2 Implementation 178
6.3 Hurrying worms 197
6.4 Conclusion 224

7 Where is arithmetic? **225**
7.1 Where is addition? 225
7.2 Beyond copyPaste 242
7.3 Binary adder by human design 243
7.4 A nice hint for evolution 261
7.5 Binary adder with a weaker hint 287
7.6 Conclusion 312

8 Evolution of genetic codes **315**
8.1 A hierarchy with two levels 315
8.2 Three level hierarchy 339
8.3 In hindsight 381
8.4 Conclusion 389

9 Conclusions **391**

10 Answers **397**

11 Glossary **471**

Preface

The series *Java for the Study of Evolution* is directed to scientists that want to manage a serious but not excessively expensive tool to study evolution by direct experimentation under perfectly controlled conditions. These requirements cannot be met in nature but only in simulations and mathematical models. In consequence, the series has three main purposes:

1. To endow the community of **researchers in biology and evolution** with *high level programming*, enabling an accurate study of models and simulations of the most diverse nature.

2. To clearly show how this tool is used to study the fundamental questions of evolution.

3. To suggest that the study of Java could be *very fruitful* for **undergraduates** in biological sciences even more than calculus alone.

We studied the fundamentals of Java, Science and Evolution in the first volume. Its content is important for the reader in the measure in which he or she desires to pursue the study of own ideas and to develop own personality.

This is the second volume, whose purpose is to present and develop a view of evolution that has been maturing since the 1930s: the genome is software and evolution is a software developer. Those that already worked out the first volume version 2 will find here exercises and new Java material to continue growing in the mastering of Java, science and evolution. Version 2: migration to NetBeans, less bugs, more readability + all programs are accepted by Java 8.

Bogotá, Colombia, *José Rodríguez*
Oct 2016

Introduction

Java is important for us because it allows to devise powerful simulations at a reasonable cost. A **simulation** consists in the building in the memory of a computer of a world with the specific properties we want, with specific instruments of measurement and with appropriate procedures of reporting what is happening there.

We learn in chapter I how to acquire and install Java and NetBeans in order to run an already extant program written in Java. We find in chapter II, the core of evolution, a classic simulation of evolution, Weasel, which helps one to understand why evolution is so cherished among the academic community.

We explain why the genome is software in chapter III and conclude that evolution is a software synthesizer. A first example of a synthesis of software is presented in chapters IV through VI: evolution is given the task of developing the software to drive a worm in the upwards direction.

On the other hand, life awakes admiration because of the excellence of its design a fact that is scientifically explained by evolution. But we are unable to accept that. Thus, our challenge is to invent clear-cut tests. In regard with the aforementioned claim, science predicts that evolution is capable of carrying complex mathematical operations of optimization. So, in chapter VII we impose over evolution the burden of designing the code for a binary adder. If we pose the problem at one step of being solved, evolution succeeds. If we harden the problem, it is very nice to see how evolution achieves some humble results. Ensuing considerations lead us to relate our simulations with the evolution of the genetic code, which is predicted to show a lot of variability. This prediction fails to be observed in nature because the variability in the genetic code seems to be incipient and explained by modification by descent from a single ancestor. Such a discordance is best interpreted as a direct falsification of the Evolutionary Theory.

Chapter 1

Starting up

`Running a first program.`

1 ***Purpose.*** *To enable journalistic visits to this volume, we see here how to acquire and install **Java** and **NetBeans** and to run an already existing Java program. If you want to solve the exercises and to enjoy the material to the full, you must go through the first volume of this series, where you can learn the fundamentals of Java, science and evolution.*

1.1 Installing Java

Java is a language to program computers. It is very popular and very powerful. Tools have been created to facilitate its use. These are the IDEs (Integrated Development Environments) that can be understood as Java editors. The following are very popular: Eclipse Classic, NetBeans, IntelliJ IDEA. We will use NetBeans because it comes with all we need without further work and it is moreover very friendly. It is distributed freely by Oracle.

Java needs a folder for work. Create and name one, for instance *workspace3*. You can do this before installing your interface or during installation.

To download *NetBeans + Java* prompt Google with `netbeans jdk bundle` and look for a link to Oracle. Follow the link and arrive to the last distribution, which by September/2016 is *JDK 8u101 with NetBeans 8.1*. Accept license agreement, choose your system (Windows, Linux or Mac), load and install the bundle on your desktop. To launch NetBeans one click or maybe two over an icon with a link to the execution file will suffice.

Play with this new tool: in the *start page* select *Try a sample project*, choose *Samples*, next *JavaFX*. Select project *Fireworks*. Expand the directory and find

the source folder. Expand it and find the source of the Java program, which ends with *.java*. Select it with return or with the secondary button of the mouse to open it. To run it, click over the editor window and with the secondary button of the mouse open a pop up menu and select the command to run the file. You must see a wonderful animation.

A very clear tutorial to create and run the first application or program in NetBeans is (NetBeans, [36] 2015). The complete guide to NetBeans would be very useful (NetBeans, [35] 2014).

2 ***Guarantee.*** *To your disposal, we present many programs that are guaranteed to work perfectly on the NetBeans Java bundle 8. Express your discomfort to*

jose@evoljava.com

Related material can be found at

http://www.evoljava.com

the site of the evolJava community.

Let us learn how to copy a program into NetBeans that it could be run. The first step is opening a project.

1.2 Purposes and projects

In Java, before copying or designing a program, one must make clear what the purpose one has in mind is. Our present purpose could be just to study this material. This intention defines a project: MyVol2. Then, we must communicate to Java our intention.

3 ***Opening a project***

To open a project follow the sequence of options beginning from the main menu: $File \rightarrow newproject \rightarrow Java \rightarrow Next$ A window will pop up for you with an invitation to fill the relevant data about your project. Give it a name: *MyVol2*. To end, click *Finish*.

By default, a source package is automatically generated together with a seed program. These have the same name as the project but all chars of the name of the source package come in lower case.

Now that we have a purpose, a project, and a seed program, let us copy and run a previously written program.

4 ***Copying a program***

The simplest programs in Java have the next basic structure, where comments begin with the marker // and instructions have no initial marker. Officially, a program in Java is called **class**.

```
// This is the structure of the simplest program.
//The name in Java of this program is structure
public class structure
{
 //The next is the head of the program
public static void main(String[] args)
{
   //Here are the instructions
}
}
```

Programs are not typed except when they are created. Instead, they are transcribed by a copy-paste procedure, for which you have four possibilities:

1. To copy each program from the present pdf document. In this case, you must resolve a little problem: the copy operation also copies navigating signals of the document, headers and page numbers, so they must be identified and deleted. The identification is done automatically after the paste operation is complete: extraneous lines of text appear with a red mark at the right margin. They must be deleted.

2. To get the Latex source, which can be opened with a Latex editor. You

   can copy any program from it without suffer anyhow. Additionally, you are invited to use our Latex source to devise your own material for yourself or for your pupils. In general, feel free to do anything good for forming a healthy and strong **EvolJava** community.

3. To get a file, in plain text, with all programs, from which you can copy the desired program immediately. To get the file, go to http://www.evoljava.com

4. If you are a teacher, you might prefer to import the accompanying zipped package with all programs in this volume. Our programs come in the file

eJVol2P.zip. Download it and memorize its location. Open NetBeans, go to menu *File* → *Import Project* → *From zip*. Search the zipped file and click OK. As *Folder*, agree on the already chosen workspace. This must expand the file and generate a project.

You can run anyone program in this project. But we need a special procedure because they are not projects on their own. To **run a program**, say *MyProgram*, go to *projects* tab, choose project *eJVol2P*, expand the tree: *source packages* → *eJVol2P* → *MyProgram.java*. In the window of *projects tab* (not in the editor window) right click over the name of the program and choose in the emergent menu *Run file*. Or, use *Shift* + *F6*. To close a running program use the menu *Run* → *Stop Build/Run* or look for a red square under the *Output tab* and activate it.

1.3 Our first program

The following is a program whose purpose is to write a message to the console. The **console** is a special window to report results and errors. If whatever cause hides the console, one can restore its visibility as follows:

For NetBeans: In the main menu follow the path: *Windows* → *Output*. Else use Ctrl + 4.

5 ***Printing to the console.*** *The next code contains a program, whose purpose is to print a message to the console.*

```
package ejvol2p;

//  Program  B5 Printing
//  This is my first program or class.
//  It prints a message to the console.
//  The name of this class is Printing.
//  On NetBeans run this program with Shift + F6.

public class Printing
{
    //This is the main method
    public static void main(String[] args)
    {
        // This is a comment.
```

```
        // Comments help to understand programs.

        // The next line contains an instruction
        // that prints a message on the console:
        System.out.println("I am starting up! ");
    }
}//End of main class B5 Printing
```

6 ***Running our program***

To copy and run this class, go to the Run menu. We must before everything else save and build the project and then run the file. To save: click over the Save All icon. To build the project: run + Build Project. To run the file: Run File.

We shall see the output of our first program that writes in the console the phrase: *I am starting up!*

7 ***License.*** *Dear Reader: as part of the exercises, you must modify the programs. So, you have the* ***license*** *to use all programs written in this book in whatever form you want even to get rich and very specially to modify them as you are asked in the exercises. This freedom might help the formation of a strong EvolJava Community.*

1.4 Review

We have learned how to acquire and install Java and its companion NetBeans. We have created a project to which we appended our first program and we have run it. With minor variations, any other program can be run just like this.

Chapter 2

The core of evolution

How we see at evolution

8 ***Purpose: We present a vision over evolution as it is seen today and that characterizes it as a world view. A classic simulation helps us to understand how it has been proposed to use evolution to explain the origin of complexity.***

2.1 Describing evolution

We offer here a vision over evolution as it is seen today from the stand point of a social scientist. The vision of evolution as reported for the lay public such as it is seen by official Authorities can be found at (Ayala et al, [3] 2008). Strikingly enough, those Authorities present as a virtue of evolution its witnessed possibility of strengthening the faith in God. In our evolJava community, we welcome the joy to be a human with the possibility of living feelings, intuitions and irrational convictions. Express your convictions succinctly, if that pleases you. But be serious: let the force of your convictions be translated into wisdom and endurance: wisdom to formulate new insights leading to fruitful research programs and endurance to accomplish begun work.

9 ***Evolution today***

It is enough to look at the entrails of a hen, a fish and a sheep to get convinced that all these species are variations on the same theme. Can we provide a natural mechanism to explain those variations beginning from just some few origins or maybe from one or maybe from no origin at all? The election of modern science is to provide a full fledged mechanism to explain variations among living beings

beginning from inanimate matter which in its turn is explained by the Big Bang, i.e. by a fluctuation of the quantum vacuum.

Let us call the proposed natural mechanism **evolution** and let us work it out as a null hypotheses of science.

As a first constitutive ingredient of our project, we have mutation or change in the genome from parents to offspring. In particular, we enjoy a direct estimation of the per nucleotide mutation rate in the fly, *Drosophila melanogaster*. The corresponding estimated mean value is 8.4×10^{-9} per generation (Haag-Liautard et al., [26] 2007).

Let us highlight the fact that actual life is a sophisticated compromise between evolution and anti-evolution, say, we cannot stop mutation (it hangs ultimately from quantum mechanics) and there are genes that warm/cold the mutation rate (Radman, [41] 2000) but nevertheless the genetic code of alive beings appears to be finely tunned to minimize the effects of mistranslation and point mutation (Freeland et al, [21] 1998). And all this does not happen by randomness alone, instead evolution is promoted by a special purpose machinery that is so amazing that it can execute even topological operations and in consequence it is very complex; actually, it is composed of various tens of enzymes that work in ensemble (Darnell et al, [8] 1986). In general, one cannot avoid to get invaded by perplexity because of the very high level of the complexity of life.

Once mutation is recognized, the next natural proposal is that mutations along lineages add together to give rise to the origin of races, new species and phyla. We cannot deny that this could happen spontaneously, but random changes probably will tend to annihilate one another. So, we need a mechanism to generate a directional tendency. The glory for imaging and documenting from nature such a mechanism belongs to Darwin ([12], 1859), even though he knew nothing about genes: if a change in the genetic information produces a variation that fits better a challenge of the environment, that genetic change will propagate even more than the other variants. A trend toward complexity is automatically produced because any environment is a very hard challenge for any species. By the same reason, diverse environments would produce over time genetic diversity. The connection of the changes of the environment with those in the genome or in the frequencies of their representatives is called **selection**.

Spontaneous changes plus selected traits would eventually cause a reproductive separation and/or morphological and ethological divergence. But from the fact that the domestic fly is the same all over the planet, one deduces that our proposals are not to be believed at once and that they need to be experimentally documented. Reality has shown that this problem is not that easy: after 160 years of hard work we must get content with incipient proofs of the belief that selection correlates with speciation (Dettman et al, [15], 2007). The problem is that an incipient proof is no

proof at all but due to the difficulty of the task, incipient proofs are considered as hits of science. Nevertheless, once and for ever, let us make it clear that science does not rely on proofs. Science is a project that relies on conviction + hard and wise work.

So, what does fuel the conviction of a modern scientist in that the microscopic mutation + selection + time are enough to explain the origin of species, phyla and life in general?

The first and fundamental reason is that a human heart is a fabric of convictions. The second is that there are various kinds of incipient proofs of speciation and Darwin himself provided some of them. Just think of the finches of the Galapagos Islands, whose variability of beaks reveal an abundant adaptive radiation, now studied at the gene level (Patel, [40], 2006).

A most loved evidence in favor of evolution relies on the **genome project**, which is the systematic study of all forms of genomes. According to results that enjoy a certain tradition and are so well established, the evolutionary trees of **systematics** correlates with evolutionary trees of molecular biology and many genes, such as cytochrome, can be traced very clearly from mammals down to the bacterium. Additionally, related genomes differ more by the form of genetic regulation than by genes themselves, say, the 94% of the human genes already appear in the genome of the chimpanzee (The Chimpanzee Sequencing and Analysis Consortium, [48], 2005). Furthermore we know for certain that microbes absorb and discard genes according to their needs in response to a change in the environment (Goldenfeld and Woeso, [25] 2007). This implies that mutation does not only means changes from parents to offspring but additionally includes the live changes in the genome due to insertions of new genes taking from other organisms.

We have now a combinatorial theory of evolution: the genes are permanently flushed as cards and a handful corresponds to the genome of a species.

Evolution also has the support of computer simulations that show that speciation might be an unavoidable event along the normal reproductive dynamics of a population that live in a competitive complex environment. A celebrated example is given by *terra* (Ray, [43] 1990).

In general, science has been continually offering advances in regard with the justification of evolution. Thus, science is dominated by **evolutionism**, which is the philosophical certainty in the power of evolution (plus its natural extensions that cover all materialism) to explain everything in spite of the fact that at present there are vacuums and paradoxes. In evolutionism, evolution does not belong in the realm of science because it cannot be the object of doubt or rejection.

Now, is there a reason to question the power of evolution to explain our very existence?

Yes, there are many. Let us consider for instance the next point: the human genome contains more than 100 microbial genes, whose expression is made in the brain, and that do not appear in the genome of the chimp (Whitfield, [59] 2007). It is straightforward to propose that these genes were transposed directly from bacteria as well as it is also straightforward to ask: who apart from God could have done the very improbable right selection for the appropriate functional combination? It is here that we recognize the power of evolutionism as a philosophical regulative principle: facts with very low probability are declared to have occurred by mere chance and no god must be annoyed.

Anyway, evolutionism must battle for a honorable place in the mark of ideologies, where we also find the religious traditions that indwelt all cultures of the world: according to some of these traditions, a human being is much more than organized matter because his or her body is a recipient of a spirit, whose reality overcomes physical death. Or more explicitly, one can find people all over the world, very especially children under five, claiming that there are ghosts at their houses.

On the other hand, we have the tremendous impact of the enormous volume of information that any genome consists in. We join these two items, spirits + complexity, and we have God the Creator. We just have reinvented **creationism**, which claims that life was designed and synthesized by a spiritual entity.

Among creationists, Christian fundamentalists are especial because they have done their best to get hatred by scientists. On one hand, they have shot their mortal guns: they have tried to manipulate the legal power at courts to achieve that the book of Genesis could be taught as a scientific alternative to evolution. They try to do so in spite of the fact that they know that their academic works (Dembski, [14] 1998) have awaken fear but no applause (Melott, [34] 2002). On the other hand, they have administrated no wine or oil to bandage the wounded bodies of their enemies, who indeed have made a very instructive defense (Padian, [39] 2005).

Some scientists have paid special attention to the creationist movement and continuously deliver literature about the position of science. One of them is Richard Dawkins, whose work has world-wide impact. Why? It is because he is a master in dealing with those questions that worry every human mind: where are we from and where do we go? The answer given to these questions by modern science in the light of evolution are very simple: we appeared as a byproduct of evolution and we cannot go further than natural law allows us to go.

It happens that a clear explanation of the ideas of evolution has not been enough to illuminate the hearts of people, and some scientists have opted for a direct attack against religion. As a part of this reaction, Dawkins publishes *The God delusion* ([11] 2006), in which he claims that traditional forms or religion are irrational and harm prone while a completely atheist science is the true religion full in epics and

splendor. Now, if we look around, we must recognize that we live in the most glorious moment that science has ever seen. Just think of the possibility to predict the function of an enzyme from the corresponding DNA code (Hermann et al., [27] 2007). This is definitively a significant lap in the direction of ab initio synthetic dragonfly (Ball, [5] 1995) - a dream that is proper of the science of our time. But troubles begin when we notice that amidst so many splendors, we find that cancer and aging seem to play with us much as a cat plays with the little mouse it just killed. The tremendous advances of science to combat these killers (Finkel et al, [20] 2007) are very far from being satisfactory, a fact that one is not aware of until one sees the own father or daughter dying in the middle of the most severe tortures because of them.

But Dawkins has prepared since long ago an answer to the suffering of mankind: we are nothing apart from transient instruments of the egoism of genes that prompt them to survive at whatever cost. The election of very complex organisms, such as we are, is just one of their strategies to achieve their goal. So, our death has no meaning apart from allowing the existence of other forms of gene combinations with slightly mutated genomes. Henceforth, it is not dead that must be feared; instead it is life that must be appreciated and enjoyed.

This form of thought was popularized by Dawkins in his book *The selfish gene* ([9] 1976), an introduction to sociobiology, whose purpose is to trace down to genes the biological causes of our conduct even of those acts that seem to be the most altruistic.

To summarize, we find in evolution the dominant model of modern science: `nothing in modern science makes sense if not in the light of evolution`, mutatis mutandis, the slogan of Dobzhansky ([16] 1973). But more to the point, evolution together with its extensions to cover the origin of life and of the universe, defines a world view with answers for all questions, including those that concern the nature and fate of man. For this reason, evolution certainly has the power to compete with any religion. And, as it happens with all religions, not all things are clear: a rich ensemble of events of low probability, which are abundant in molecular biology, must be the subject of intense scientific study and original theories (Kauffman, [31] 1995) but not of fear driving into religion. So, evolutionists are encouraged to defend this vision with active propaganda (Editor of Nature, [17] 2008). By contrast, electromagnetism (Rodriguez, [44] 2008) needs neither propaganda whatsoever nor lawsuits at the court. This contrast shows that an all encompassing theory of evolution is still in the making but we must recognize that as a project it is sustained by the confidence, tenacity and endurance of very brilliant minds.

Let us abandon these captivating discussions and let us turn now to the formulation and study of very precise questions that can be enlightened with Java. To begin with, it will be straightforward for us to understand how Dawkins proposes the form as we can use evolution to explain the origin of complexity. Over this background, we propose in the next chapter another vision over evolution, one that has being developing during the last century and the one because of which present generation will be remembered.

2.2 Explaining complexity

Complexity is explained in *The Blind Watchmaker* (Dawkins, [10] 1986) as the result of either randomness or accumulation of small change. Let us see how they operate and why the second model is the preferred one. Let us begin with randomness.

10 ***Random assembling***

Randomness is invoked as follows: "I don't know who it was first pointed out that, given enough time, a monkey bashing away at random on a typewriter could produce all the works of Shakespeare. The operative phrase is, of course, given enough time." (pg 46). The Author next continues with a simulation in which he synthesizes a phrase of Hamlet: "Methinks it is like a weasel." (In reference to the similitude of a cloud to a weasel, a carnivorous mammal.) He finds that the phrase can indeed be synthesized by randomness but only after a long time.

Let us develop a program to verify this.

11 ***Simulating randomness***

From the work of Dawkins we know that evolution is better than randomness. So, let us investigate how well is random guessing to find a predefined goal. The program follows:

```
//Program RandomSearch,  B11
//A string of length k is declared as goal
//and it is guessed by random searching.
//The number of trials done to
//achieve the goal is reported.
```

```
package ejvol2p;



import java.util.Random;

public class RandomSearch
{
    //All string have the same length
    static int stringLength=0;
    //A particular string
    static String individual = "";
    //Boolean condition to continue else halt
    static boolean problem=true;
    //The string to b e guesses
    static String goal = "";


    // Turn on of the random generator
    static Random r  = new Random();

    //  This method generates random chars
    //  only lower case letters of the alphabet are used
    private  static  char  randomChar()
    {
     return  ( char  ) ( r.nextInt( 26 ) + 65  );
    }

    //  A string of stringLength characters is generated
    private static String generateString(int stringLength)
    {
        individual = "";
        //An individual is assembled char by char
        for(int j = 1; j<= stringLength; j++)
        {
            // We generate a random char
            char c =  randomChar();
            //System.out.println( "char = " + c);
            // char c is converted into a string s
```

```
            String s = ""+c;
            // s is concatenated to Individual[i]
            individual  = individual +s;
        }
        return individual;
    }


    //  The goal is defined
    private static void   defineProblem(int n)
    {
        goal = generateString(n);
        stringLength = n;
        System.out.println( "goal = " + goal);
    }

    //  The goal is achieved by guessing at random
    //Notice how easy is to simulate randomness
    private static void  randomSolution(int n)
    {
        int trial = 0;
        String a;
        while (problem)
        {
            trial = trial + 1;
            a=generateString(n);
            /*
                System.out.println("Try number = " +
                    trial   + " individual " + individual);
            */
            if (a.equals(goal))
            {
                System.out.println( "Task ended by trial = "
                        + trial );
                problem =false;
            }
        }
    }

    public static void main(String[] args)
```

```
    {
        for(int k = 1; k<=15; k++)
        {
            System.out.println("Length of goal = " + k);
            defineProblem(k);
            while (problem)
                randomSolution(k);
            problem = true;
            System.out.println();
        }
    }
}//End of main class RandomSearch,  B11
```

12 ***Exercise*** *Run the program and play with the code. To see what a method does, activate available printing options. Observe the code and verbalize its design and functioning. Let us recall that one can stop the running of a program by clicking at the red terminator square.*

13 ***Exercise*** *Make a mathematical calculation showing that the number of trials to achieve by mere randomness a goal with n letters is an exponential function in n. We can test this result with our program thanks to a regression analysis. To that aim, modify the previous code to produce an output to be pasted directly on Excel or Calc: it must produce just the length of the string and the number of trials that were necessary to achieve the goal. All other commentaries are unnecessary wording and shall not appear. Copy the output to the clipboard and paste it to Excel or Calc (use paste special) and make a graphic (in the XY scattering mode). Run a regression analysis to determine the nature of the function that describes the relation of the number of trials to the length of the goal.* *Answer*

14 ***Accumulating small change***

Dawkins next proposes to make a program in which evolution is simulated:

"[The program] again begins by choosing a random sequence of 28 letters, just as before:

WDLMNLT DTJBKWIRZREZLMQCO P

It now 'breeds from' this random phrase. It duplicates it repeatedly, but with a certain chance of random error -'mutation' - in the copying. The computer examines the mutant nonsense phrases, the 'progeny' of the original phrase, and chooses

the one which, *however slightly*, most resembles the target phrase, METHINKS IT IS LIKE A WEASEL".

The Author next explains what happens if the same process is repeated again and again, i.e., what is the fruit of evolution: in some few generations the target phrase is reached.

15 *Which is better?*

We have by now at least two models, randomness and evolution, to explain how specific information may appear. Thinking of the extreme complexity of living beings, it is rather abusive to claim that they appeared by one or the other method, but we can decide which method is more plausible just by one criterion: the preferred method produces the specified target using the minimum number of trials. For evolution this number is equal to population sizes times the number of generations. This criterion stems from the fact that available resources are always too poor as compared with needed resources to solve complex problems.

So, our question reads: is evolution better than randomness? To answer that question we must simulate evolution to see what it gives of itself.

16 *Evolution*

The next program implements a simulation of evolution to find a predetermined string. The program produces an initial population of popSize individuals generated at random, looks for the string that best resembles the goal, generates a new generation of clones of that individual and mutates a site in each new individual which are also recombined. All individuals are next assessed for resemblance to the goal and the best is chosen to be the father of new clones. And so on until the goal is achieved. The program follows:

```
package ejvol2p;


//Program WeaselResearch, B16
//A string of length k is declared as goal,
//which is guessed by evolution.
//The number of generations and trials done to
//achieve the goal  are reported.
```

```
import java.util.Random;

public class WeaselResearch
{

//Evolution of a population
//of strings.

//Strings are at the same time
//genotype and phenotype.
//They are like ribozymes.
//Selection is organismic, in the sense that
//the genome is assessed as a whole.

//Pre-declaration of global variables.
//They are used all throughout the whole class.

//Individuals are kept in the array
//Individual[]. It is an array of strings.
//The number of individuals is equal to popSize


private static final int popSize = 1024;
static String Individual[ ];//Predeclaration
static String Individual2[ ];
static String  individual ="";
static int stringLength=0;
//Order  is an array of integers.
static int Order[];//pre-declaration
static int gen;
static boolean problem=true;
static String goal = "";



//Turn on of the random generator
static Random r  = new Random();

//This method generates random chars
private  static  char  randomChar()
```

```
{
 return  ( char ) ( r.nextInt( 26 ) + 65  );
}


//A random string of length stringLength is generted
private static String generateString(int stringLength)
{
    individual = "";
    //An individual is assembled char by char
    for(int j = 0; j< stringLength; j++)
    {
     // We generate a random char
     char c =  randomChar();
     //System.out.println( "char = " + c);
     // Char c is converted into a string s
     String s =  "" + c;
     // s is concatenated to Individual[i]
     individual  = individual +s;
    }
     return individual;

}

// An initial population of strings is set  up.
private static void initialization( )
{
    //Formal declaration of our arrays.
    Individual= new String[popSize];
    Individual2= new String[popSize];
    //System.out.println("ORIGINAL POPULATION");
    for(int i = 0; i< popSize; i++)
    {
        Individual[i]=generateString(stringLength);
        Individual2[i]=Individual[i];
        //System.out.println("Individual " + i + "   " +  Individual[i
    }

    //Declarations and default initializations
     // of other arrays.
```

```
    Order= new int[popSize];
    for(int i = 0; i< popSize; i++)
       Order[i]=0;
}


//The goal is defined
private static void   defineProblem(int k)
{
    goal = generateString(k);
    System.out.println("goal = " + goal);
    stringLength = k;
    initialization();
}


//The more similar to the goal, the fitter.
//Method taken from weasel by MAx Scott
static public int fitness(String c)
{
    int errors = 0;
    int len = goal.length();
    for (int i = 0; i < len; ++i)
        if (c.charAt(i) != goal.charAt(i)) errors++;
    return errors;
}







//Individuals are sorted by fitness
private static void Sorting()
{
    int   Champ;
    // Default initialization
    int Fitness[];
    Fitness= new int[popSize];
```

```
    for(int i = 0; i< popSize; i++)
    {
        //System.out.println(" Individual i" +Individual[i]);
        Fitness[i] = fitness(Individual[i]);
        //Fitness 0 means no errors

    }
    //We sort individuals by fitness
    for(int i = 0; i< popSize;i++)
    {
        Champ = 0;
        for(int j = 0; j< popSize;j++)
            if (Fitness[j] < Fitness[Champ])   Champ = j;


        Order[i] = Champ;
        Fitness[Champ] = stringLength;
    }

    System.out.println("Champ by gen = " + gen  + ": " + Individual[Or
    + " -> " + fitness(Individual[Order[0]]));
    /*
    System.out.println("sorted arrray by gen = " + gen);
    for(int i = 0; i< popSize;i++)
    {
        System.out.println("i = " +  i + "   " + Individual[Order[i]]
            + "   " + fitness(Individual[Order[i]]) + " errors");
    }
    */
    if (fitness(Individual[Order[0]])  == 0)
    {
        problem = false;

        System.out.println( "Task ended by generation = " + gen);
        //In each generation, popSize individuals are checked out.
        System.out.println( "Number of trials = " + popSize*gen );
    }
}
```

```
//The new generation is made
private  static void newGeneration()
{
    int   i=0;
    //The fittest individuals are reproduced
    // numbCopies times
    int numbCopies = 10;
    int   k=0;  //rank, 0 is the highest

    while (i < popSize)
    {
        int counter = numbCopies;
        //Rank k is reproduced numbCopies times
        while ((counter > 0) & (i < popSize))
        {
            //System.out.println(i + "" + l);
            Individual2[i] = Individual[Order[k]];
            //System.out.println(i + " " + counter + " " + k);
            i++;
            counter= counter-1;
        }
        k++;
    }

    for(int j = 0; j < popSize; j++)
    {
        Individual[j] = Individual2[j];
        //System.out.println(j + " " + Individual[j]);
    }

}



private static void Reproduction()
{
    //Individuals are sorted by length
    Sorting();
    //The new generation is produced
```

```
    newGeneration();
}



//String a is mutated at a random place.
//Method taken from Weasel by Max Scott
private static String change(String a)
{
    int l = r.nextInt(stringLength);
    //System.out.println(l);
    String z = ((l <= 0)  ?  ""  :  a.substring(0, l)) +
        randomChar() +
        ((l >= (a.length() - 1))  ?  ""  :
                                a.substring(l + 1));


    return z;
}


// All individuals are mutated
private static void Mutation()
{
    for(int i = 0; i < popSize; i++)
    {
      Individual2[i] = change(Individual[i] );
      //System.out.println(Individual[i]  + " -> " + Individual2[i]);
    }
    for(int j = 0; j < popSize; j++)
    {
        Individual[j] = Individual2[j];
    }
}



//Recombination is enabled
private static String recInd(int l)
{
```

```
    String s = Individual[l];
    String  s1= "";
    int m= s.length();
    int m1 = 0;
    if (m>0)
    {
      m1 = r.nextInt(m) ;
     s1= s.substring(0,m1);
    }

    int k = r.nextInt(popSize);
    String t = Individual[k];
    int n= t.length();
    String  t2= t.substring(m1);

    String recombinant = s1 + t2;
    //System.out.println(s + "+" + t  + "->" + recombinant + "/");
    return recombinant;
}


// The definitive population  is a recombinant one.
private static void Recombination()
{

    //
    for(int l=0; l<popSize; l++)
       Individual2[l] = recInd(l );
    //Ind2 is transferred to Ind
    for(int i=0; i<popSize; i++)
       Individual[i] =  Individual2[i] ;
}
        //


//Given a goal, it is guessed by evolution
private static void  evolSolution()
{
    Reproduction();
    //The new population is subjected to mutation
```

```
    Recombination();
    Mutation();

    gen++;
}


public static void main(String[] args)
{
    for(int k = 6; k<=40; k++)
    {
        System.out.println("Length of goal = " + k);
        defineProblem(k);
        gen = 1;
        while (problem)
            evolSolution();
        problem = true;
        System.out.println();
    }
}
}//End of program WeaselResearch, B16
```

jose

17 ***Exercise and challenge.*** *Observe the code and try to understand its design. Run it and decide once and for all times whether or not our simulation of evolution is better than chance. If you like rigor, you can find the regression function under evolution to compare it with the regression function under randomness. Decide whether the regression function under randomness is above that under evolution.* *Answer*

18 ***Weasel by early professionals***

Dawkins wrote his simulations in Basic and Pascal. Very soon, various programs were written by different professional developers to implement his ideas, including a Java version, which we present right now.

The next program is a display of skills in the art of software design with the tool that were available at that time. The reader is invited to compare the style of this Author with ours. It is important to notice that software development is always accompanied with a style of programming. It is also wise to invert some time trying to scientifically define a style. We will find that task somewhere along our work.

We find **interfaces** for the first time in this program. An interface is the visible face of a piece of software whose details are not important because all that matters is its role or function. Nevertheless, those details must be somewhere specified and marked with the reserved word **implements**. Interfaces are important because they allow to rewrite the implementation as many times as desired and the User will not notice.

The program follows:

```
//The name of this class is Weasel, number B18
//Class Weasel: evolution reproduces Shakespeare.
/*
 * Weasel illustrates the ability of evolutionary techniques
 * to solve enormous problems rapidly.
 * Copyright (C) 1998  Scott Maxwell
 *
 * This program is free software; you can redistribute it
 * and/or modify it under the terms of the GNU General
 * Public License as published by the Free Software
 * Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even the
 * implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 * See the GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public
 * License along with this program; if not, write to the
 * Free Software Foundation, Inc., 59 Temple Place
 * - Suite 330, Boston, MA 02111-1307, USA.
 *
 * You can reach me at s-max@pacbell.net.
 * End of note from Scott MAxwell
 */

// This applet is mostly finished.  Things to do:

// (*) Validate the user's input (to make sure all
// characters of the
```

```
// given phrase are in Weasel's alphabet).

// (*) A better ScoreFn might mitigate the plateau problem.

//http://delrio.dcccd.edu/pnunley/P101/WEASEL.HTM

/* Note by Jose Rodriguez, 2016:
 * This is how programming of GUIs (graphic user interfaces)
 * looked like in 1998: Applets and AWT system.
 * Today we prefer JavaFX:
 * find a modern version of this program in
 * Shakespeare7, Vol XVI.
 *
 */
package ejvol2p;


import java.applet.*;
import java.awt.*;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;




// The simulation breeds (a derived type of) Critter.
interface Critter
{
    // Return a randomly created Critter.
    abstract public Critter spawn();


    // Breed this critter with the supplied one, returning
    // the offspring as a new Critter.
    abstract public Critter breedWith(Critter c);


    // Randomly mutate this critter.
    abstract public void mutate();
```

```
    // Return a short human-readable description of the
    //Critter.
    abstract public String describe();
}



// ScoreFn objects are used to score Critters.
// Typically, a ScoreFn
// is a closure that saves some state (such as a perfect
// Critter), which is used to compute the score.
interface ScoreFn
{
    // Return the supplied Critter's score.
// Scores are never
    // negative, and the best score is 0 (so the computed
    // score is a measure of "wrongness").
    abstract int score(Critter c);


    // Return the worst possible score a Critter may receive.
    abstract int worst();
}



// Implement a simple progress bar.  This fills left to
// right and prints the filled percentage in the middle
// of the bar.
class SimpleProgressBar extends Canvas
{

    private static final long serialVersionUID = 1L;
    int numer = 0; // Numerator of the filled-fraction.
    int denom = 1; // Denominator of the filled-fraction.
```

```
    // Create the progress bar.
public SimpleProgressBar()
{
    setForeground(Color.blue);
    setBackground(Color.white);
}


    // Fill n/d of the progress bar.
public void progress(int n, int d)
{
    numer = n;
    denom = d;

    Graphics g = getGraphics();
    if (g != null)
        paint(g);
}


// Paint callback.
    @Override
    public void paint(Graphics g)
{
    if (denom == 0)
        return;

    int w = getSize().width;
    int h = getSize().height;

    double fraction = (((double) numer) / denom);
    int intFraction = (int) (fraction * 100.0);
    int mid = (int) (w * fraction);

    // Draw the filled and un-filled parts.
    g.setPaintMode();
    g.fillRect(0, 0, mid, h);
    if (mid != w)
        g.clearRect(mid + 1, 0, w - mid + 1, h);
    g.drawRect(0, 0, w - 1, h - 1);
```

```
    // Draw the text in the middle of the bar
    // (draw it in XOR mode so that we can read it no
    //  matter how full the bar is).
    g.setXORMode(getBackground());
    String txt = "" + intFraction + "%";
    FontMetrics fm = g.getFontMetrics();
    g.drawString(txt,
        w / 2 - fm.stringWidth(txt) / 2,
        h / 2 + fm.getHeight() / 2);
}
}



// Handle displaying tracing information.
class Tracer
{
//  The TextArea in which we print the info.
private TextArea tracer;
private SimpleProgressBar bar; // Progress bar we can fill.


    // Ctor.
public Tracer(TextArea t, SimpleProgressBar b)
{
    tracer = t;
    bar = b;
}


    // Print this tracing text.
public void trace(String txt) {  tracer.append(txt);  }


    // Clear everything.
public void reset()
{
tracer.setText("");  progress(0, 1);
```

```
}


    // Set the progress bar to numer/denom full.
public void progress(int numer, int denom)
{
    bar.progress(numer, denom);
}
}




// Display Darwin's periodic Reports.
class Reporter
{
private Tracer tracer; // Where to display the report text.


    // Ctor.
public Reporter(Tracer t)  {  tracer = t;  }



    // Print a Report in human-readable format.
public void report(Report r)
{
    progress(r.getNumer(), r.getDenom());
    tracer.trace(
      "\n" +
      "Generation        " + r.getGeneration() + "\n" +
      "Tries <=          " + r.getTries() + "\n" +
      "Best Critter      " +r.getCritter().describe()+"\n"+
      "Score (0 is best) " + r.getHighScore() + "\n");
}


 // Just update the progress bar (make it numer/denom full).
public void progress(int numer, int denom)
{
```

```
    tracer.progress(numer, denom);
}


    // Print some straight text.
public void printText(String txt)
{
    tracer.trace(txt);
}
}


// A progress report periodically issued by Darwin.
class Report
{
private int generation; // # generations done so far.
private int tries; // Upper bound on # Critters examined.
private int highScore; //High score (non-negative,0 is best).
private Critter critter; // High-scoring critter.
private int numer; // Numerator of filled fraction.
private int denom; // Denominator of filled fraction.


    // Ctor.
public Report(int g, int t, int s, Critter c, int n, int d)
{
    generation = g;
    tries = t;
    highScore = s;
    critter = c;
    numer = n;
    denom = d;
}


    // Retrieve various aspects of the report.
public int getGeneration() {  return generation;  }
public int getTries() {  return tries;  }
public int getHighScore() {  return highScore;  }
public Critter getCritter() {  return critter;  }
```

```
public int getNumer() {  return numer;  }
public int getDenom() {  return denom;  }
}



// Evolve Critters toward a solution.
class Darwin implements Runnable
{
// Where to send my periodic progress reports.
    Reporter reporter;
//  How to score Critters.
    ScoreFn  scoreFn;
//  Breeding population of Critters.
    Critter  population[];
//  # Critters in population.
    int      nCritters;
//  How frequently to make a progress report.
    int      updateFreq;
    Thread   myThread;


// Return the index of a Critter randomly selected from the
// population.
private int randomCritter()
{
    return (int) (Math.random() * nCritters);
}


// Return the index of a Critter randomly selected from
// the population, but don't return the supplied index.
//  This assumes our population size is larger than 1.
private int randomCritterExcept(int not)
{
    int n = randomCritter();
    while (n == not)
        n = randomCritter();
    return n;
```

```
}


// Ctor.
public Darwin(Critter prototype, int n, int uf,
                  ScoreFn f, Reporter r)
{
    nCritters  = n;
    updateFreq = uf;
    reporter   = r;
    scoreFn    = f;

    // Use our prototype Critter to randomly create a
    // population of Critters.  Since this is often slow,
    //  show our progress  as we go.
    population = new Critter[nCritters];
    for (int i = 0; i < population.length; ++i)
    {
        population[i] = prototype.spawn();
        if ((i % 10) == 0)
            reporter.progress(i, population.length);
    }
    reporter.progress(1, 1);

    myThread = new Thread(this);

    // Since this thread is such a CPU hog, lower its priority.
    // Among other effects, this makes the GUI (specifically, the
    // Stop button) more responsive.  Under Netscape, this
    // generates an exception, but it still lets us get on with
    // life.
    // myThread.setPriority(myThread.getPriority() - 1);
}


// Do the breeding.
@Override
@SuppressWarnings("CallToThreadYield")
public void run()
{
```

```
    long start = System.currentTimeMillis();
    //Upper bound on different # Critters examined.
    int tries = nCritters;
    int worst = scoreFn.worst();

    int highScorer = 0; // Index of the high-scoring Critter.
    int highScore = -1; // Its score (set in the init loop).

    // Get the initial fitnesses of all Critters in the
    // population.  This determines the initial best critter
    // (which could even be a perfect solution).
    for (int i = 0; i < nCritters; ++i)
    {
        int score = scoreFn.score(population[i]);

        if ((i == 0) || (score < highScore))
        {
            highScore = score;
            highScorer = i;
        }
    }

    // Loop until we find a solution.
    for (int generation = 0; true; ++generation)
    {
Thread.yield(); // Allow any other threads to run.

// If we've found a solution or it's time to issue a
// report, issue a report.
if ((highScore == 0) || ((generation % updateFreq) == 0))
            reporter.report(new Report(generation, tries, highScore,
population[highScorer], (worst - highScore), worst));

if (highScore == 0)
        {
            long tm = (System.currentTimeMillis() - start) / 1000;
            if (tm != 0)
                    reporter.printText("\n\nChecked " + tries +
                        " critters in " + tm +
                        " seconds == " + (tries / tm) +
```

```
                        " tries/sec.\n");
            return;
        }

        // Breed the high scorer with some of the other Critters,
        // then mutate some.
        int some = nCritters / 8;
        for (int j = 0; j < some; ++j)
        {
            int n = randomCritterExcept(highScorer);
            population[n] =
                    population[highScorer].breedWith(population[n]);
            int score = scoreFn.score(population[n]);
            if (score < highScore)
            {
                highScore = score;
                highScorer = n;
            }
        }

        for (int j = 0; j < some; ++j)
        {
            // I was using randomCritter() here, but for
            // efficiency I no longer check the fitness of all
            // critters at each iteration -- hence, if the high
            // scorer is mutated, the code won't choose his
            // successor.  Fix ASAP.
            int n = randomCritterExcept(highScorer);
            population[n].mutate();
            int score = scoreFn.score(population[n]);
            if (score < highScore)
            {
                highScore = score;
                highScorer = n;
            }
        }

        tries += some * 2;
    }
}
```

```
public void stop()
{
    // I'd like to suspend()/resume() or wait()/notify() here,
    // but Netscape just whines about that.
    //Playing around
    // with ThreadGroups doesn't seem to make it happy,
    // either.
    if (myThread != null)
        myThread.interrupt();
}


public void start()
{
    if (myThread != null)
        myThread.start();
}
}




//A particular Critter for the purposes of the Weasel applet.
class WeaselCritter implements Critter
{
private static final String chars =
    "abcdefghijklmnopqrstuvwxyzABCDEF" +
    "GHIJKLMNOPQRSTUVWXYZ 0123456789,.?!'";
private static int nChars = chars.length();
private static char character [] = null;

//This Critter's guess at the solution.
private String rep;


// Return a randomly generated string of the given length.
private static String randomStringOfLength(int len)
{
```

```
    String s = "";
    for (int i = 0; i < len; ++i)
        s = s + randomChar();
    return s;
}


    // Return a character chosen randomly from our alphabet.
private static char randomChar()
{
    return character[(int) (Math.random() * nChars)];
}


// Return the index of a random character position
// within my guess.
private int randomPos()
{  return (int) (Math.random() * rep.length());  }


// Return a String that is the result of replacing the
// indicated substring of s1 with the corresponding
// substring of s2.  The substring's range is from
// index start to end, inclusive.
private static String repl(String s1, String s2,
                           int start, int end)
{
    if (start > end)
        return repl(s2, s1, end, start);

    String s = "";

    if (start > 0)
        s = s1.substring(0, start);
    while (start <= end)
        s = s + s2.charAt(start++);
    if (end < s1.length())
        s = s + s1.substring(end + 1);

    return s;
```

```
}


    // Initialize the class.
public static void init()
{
    if (character == null)
    {
        character = new char[chars.length()];
        for (int i = 0; i < chars.length(); ++i)
            character[i] = chars.charAt(i);
    }
}


    // Ctors.
public WeaselCritter(int len)
{
    init();
    rep = randomStringOfLength(len);
}

public WeaselCritter(String s)
{
    init();
    rep = s;
}


// Return a randomly generated (Weasel)Critter.
@Override
public Critter spawn()
{
    return new WeaselCritter(randomStringOfLength(
                                                    rep.length()));
}


// Return the result of breeding myself with the supplied
// (Weasel)Critter.  This is a simple crossover technique.
```

```
@Override
public Critter breedWith(Critter c_)
{
    WeaselCritter c = (WeaselCritter) c_;
    return new WeaselCritter(repl(rep, c.rep, randomPos(),
                                                    randomPos()));
}


    // Mutate myself.  This simply replaces one randomly
    // chosen character.
@Override
public void mutate()
{
    int pos = randomPos();
    rep = ((pos <= 0)  ?  ""  :  rep.substring(0, pos)) +
        randomChar() +
        ((pos >= (rep.length() - 1))  ?  ""  :
                                rep.substring(pos + 1));
}


// Return a short readable self-description.  For this
// critter, the job is trivial.
@Override
public String describe()  {  return rep;  }


// Return the character at index i in my representation.
public char charAt(int i)  {  return rep.charAt(i);  }
}




// Score WeaselCritters.
class WeaselScoreFn implements ScoreFn
{
    //  A WeaselCritter matching this string wins.
    String goal;
```

```
    // Ctor.
public WeaselScoreFn(String g)
{
    goal = g;
}


// Score the supplied (Weasel)Critter.
@Override
public int score(Critter c_)
{
    WeaselCritter c = (WeaselCritter) c_;

    int wrong = 0;
    int len = goal.length();
    for (int i = 0; i < len; ++i)
        if (c.charAt(i) != goal.charAt(i))
            ++wrong;

    return wrong;
}


// Return the worst possible score (all wrong).
@Override
public int worst()
{
    return goal.length();
}
}




// And now, the Applet itself.
public class Weasel extends Applet
{
```

```
private static final long serialVersionUID = 1L;

private boolean initialized = false;

//Where user types goal text.
private TextField goalTxt;
//Breed button.
private Button breedBtn;
//Stop button.
private Button stopBtn;
//Where to print tracing info.
private TextArea tracingTxt;
//Where user types # critters.
private TextField nCrittersTxt;
//Where user chooses upd8 freq.
private Choice genChc;
//Applet-wide progress bar.
private SimpleProgressBar progBar;
//Manages tracingTxt object.
private Tracer tracer;

private Darwin darwin = null;

public static final int wantW = 500;
public static final int wantH = 400;


    // Ctor.
public Weasel()  {}


// One-time applet initialization.
@Override
public void init()
{
    // I've heard that some browsers may (incorrectly)
    // invoke this method more than once.
    // So guard against that.
    if (initialized)
        return;
```

```
    initialized = true;

    Label lbl;

    // Set up the user interface.  This uses the default
    // FlowLayout, creating a full-width Panel for each row
    // of items.

    // First row.
    int rowHeight = 0;
    Panel p = new Panel();
    p.add(lbl = new Label("Goal Phrase:"));
    Dimension size = lbl.getSize();
    rowHeight = Math.max(rowHeight, size.height);

    goalTxt = new TextField("Methinks it is like a weasel",
                                                                    32);
    goalTxt.setSize(new Dimension(wantW - size.width,
        size.height));
    p.add(goalTxt);
    rowHeight = Math.max(rowHeight,goalTxt.getSize().height);

    p.setSize(new Dimension(wantW, rowHeight));
    add(p);

    // Second row.
    rowHeight = 0;
    p = new Panel();
    Label lbl2 = new Label("Population Size:");
    p.add(lbl2);
    rowHeight = Math.max(rowHeight, size.height);
    p.add(nCrittersTxt = new TextField("1024", 5));
    Dimension size2 = nCrittersTxt.getSize();
    Dimension size3 = size2;
    size2 = size3;
    rowHeight = Math.max(rowHeight, size2.height);

    Label lbl3 = new Label("Update Every");
    p.add(lbl3);
    rowHeight = Math.max(rowHeight, size.height);
```

```
genChc = new Choice();
String choices[] = {
    "1 Generation",
    "10 Generations",
    "25 Generations",
    "50 Generations",
    "100 Generations",
    "250 Generations",
    "1000 Generations"
};
for (int i = 0; i < choices.length; ++i)
    genChc.addItem(choices[i]);
genChc.select(1);
p.add(genChc);
rowHeight = Math.max(rowHeight, genChc.getSize().height);

p.setSize(new Dimension(wantW, rowHeight));
add(p);

// Third row.
rowHeight = 0;
p = new Panel();
p.add(breedBtn = new Button("Evolve A Solution"));
rowHeight = Math.max(rowHeight,
                                  breedBtn.getSize().height);

p.add(stopBtn = new Button("Stop Breeding!"));
rowHeight = Math.max(rowHeight,
                                   stopBtn.getSize().height);

p.setSize(new Dimension(wantW, rowHeight));
add(p);

// Fourth row.
rowHeight = 0;
p = new Panel();
p.add(progBar = new SimpleProgressBar());
progBar.setSize(wantW, 32);
rowHeight = Math.max(rowHeight,
                                   progBar.getSize().height);
```

```
    p.setSize(new Dimension(wantW, rowHeight));
    add(p);

    // Fifth row.
    rowHeight = 0;
    p = new Panel();
    p.add(tracingTxt = new TextArea("", 8, 64));
    tracingTxt.setFont(new Font("Courier", Font.PLAIN, 12));
    tracingTxt.setEditable(false);
    rowHeight = Math.max(rowHeight,
                                        tracingTxt.getSize().height);

    p.setSize(new Dimension(wantW, rowHeight));
    add(p);

    tracer = new Tracer(tracingTxt, progBar);
}


// Wait for the "Breed Solution" button to be clicked.
@Override
public boolean action(Event e, Object o)
{
    if (e.target.equals(breedBtn))
    {
        tracer.reset();
        String goal = goalTxt.getText();
        tracer.trace("Attempting to match `" + goal +
                                                "'\n" +
            "(Generating initial population now ....)\n");

        int n = 0;
        while (true)
        {
            n = Integer.parseInt(nCrittersTxt.getText());
            if (n <= 1)
                nCrittersTxt.setText("1024");
            else
                break;
```

```
        }

        String gTxt = genChc.getSelectedItem();

        darwin = new Darwin(new WeaselCritter(goal.length()),
            n,
            Integer.parseInt(gTxt.substring(0, gTxt.indexOf(' '))),
            new WeaselScoreFn(goal),
            new Reporter(tracer));
        darwin.start();

        return true; // Handled it myself.
    }
    else if (e.target.equals(stopBtn))
    {
        if (darwin == null)
            tracer.trace("Not evolving.\n");
        else
        {
            darwin.stop();
            darwin = null;
            tracer.trace("Stopped evolving.\n");
        }
    }
    return true;
    //return super.action(e, o);
}


//
@Override
public void stop()
{
    if (darwin == null)
        return; // We weren't doing anything anyway.

    darwin.stop();
    darwin = null;
    tracer.trace("\nThe browser stopped Weasel.\n");
}
```

```
@Override
public void start()
{
    if (darwin != null)
        darwin.start();
}


    // In case we're not running in in a Web browser.
public static void main(String args[])
{
    WeaselFrame frame = new WeaselFrame(wantW, wantH);

}
}



// Only used when the Weasel applet is not running in a
// Web browser.
class WeaselFrame extends Frame
{

private static final long serialVersionUID = 1L;


public WeaselFrame(int w, int h)
{
    super("Weasel");

    setSize(w, h); // Approximately; will change later.
    Weasel weasel = new Weasel();
    //add(weasel, BorderLayout.CENTER);
    add(weasel);
    weasel.init();
    weasel.start();
    setVisible(true);
```

```
    this.addWindowListener(new WindowAdapter()
        {
            @Override
            public void windowClosing(WindowEvent we)
            {
                dispose();
            }
        }
    );
}
}//End of main class Weasel, number B18
```

19 ***Exercise*** *For members of the evolJava community: run the code and play with it. Try to figure out the design of the program. For visitors: our book is a lab where one learns by direct experimentation. So, you are encouraged to run all programs, such as we learned in the first chapter. Let us recall that to run a program, one must have an open project. Next open a class (Weasel) and appends it to the project, copy the code of the program to the clipboard and paste it to the editor window and run it. Results appear in a special window that could be provided by the program else over the console (a special window for reporting outputs and errors). In the present case, the reporting window is provided by the program (but must be resized).*

20 ***Observations and Criticisms***

- All individuals are viable, i.e., there are no lethal mutations, but different individuals may have different fitness, so deleterious mutation may exist.
- A critter is just a string of symbols taken from an alphabet. There is no dissipation of free energy or life as we know it. The simulation is just an informational tracer of evolution restricted to DNA or RNA.
- Evolution is presented as preferable to chance because chance is too slow. Why do we need to worry about time? Are not 10^{10} years, the age of the Earth, or 10^{16} years, the age of our universe, enough to explain all genomes?
- Why other resources, such as memory requirement that simulates the number of atoms, were not included as fundamental limitations? Let us recall that we have some 10^{90} atoms to play with.
- Our simulation of evolution, of achieving a predefined goal, has nothing to do with biological evolution. Nevertheless, it is perfect for some tendencies

in religion in which God used natural evolution to synthesized what He already had planned. But our simulation is not good for science that has made a great effort to keep discussions in within the materialistic frame, in which there is no predefined goal. One cannot deny that this effort has propelled the progress of science. We will return to the terrible problem of achieving a theology free theory in its due moment.

2.3 Better than evolution

We can accept that evolution is much better than randomness. But, what about the following program?

21 ***Slashing the problem***

Too many professional engineers never have developed a genetic algorithm, a simulation of evolution. Nevertheless, they earn a life. Why? The reason is that they use their intelligence in forms that may sound as alien to evolution, as it is shown by the following program that guesses a predefined word by simple checking out its characters one by one. The code follows:

```
//Program CheckOut, B21
//A predefined word is guessed by direct
//checking against the chars of the alphabet;
//only lower case letters are used.


package ejvol2p;


public class CheckOut
{

static String goal = "methinks it is like a weasel";
static char alphabet[] = {'a', 'b', 'c','d','e','f',
                          'g','h','i','j','k','l','m',
                         'n','o','p','q','r','s','t',
                          'u','v','w','x','y','z',' '};


    public static void main(String[] args)
```

```
    {
        int n = goal.length();
        String guess = "";
        for(int i = 0; i<n;i++)
        {
            //the char at position i is copied
            char c = goal.charAt(i);
            for (int j = 0; j<n-1;j++)
            {
             //Each char of the goal is checked out
             //against the chars of the alphabet.
             if (c==alphabet[j]) guess = guess + alphabet[j];
            }
        }
        System.out.println(guess);
    }
}//End of program CheckOut, B21
```

22 ***Exercise*** *Run the previous code, observe how simple and direct it is. It is also very fast even so it might be made faster. Compare it with our two versions of weasel, in which evolution is used.*

23 ***Exercise*** *Try to answer to the following objection: If evolution can be superseded by ordinary programming, what it is worth and why it continues to be used in modern science?* Answer

24 ***What is evolution?***

Evolution arrived at science as a discover made by field biologists. But today, evolution is a tool that is widely used everywhere in science, technology, industry, art. Thus, evolution could be defined variously among diverse professions. For organismic biology, evolution is the process composed of differential reproduction, mutation and recombination. Nevertheless, our point of view is that of modern molecular biology: evolution is the result of text editing, it is everything that can be made with DNA (and RNA), and with texts in general. For the sake of operational clarity, let us take the next items as the **basic items of evolution**:

- We can synthesize strings at random. Thus, pure **randomness** is part of evolution. That is why we include random generators in our simulations.

- DNA can be replicated. So, given a string a, one can produce another string b with the same information. And given a string a, we also can copy any substring c of a. Thus, **copying substrings** is fundamental to evolution and it is equivalent to reproduction.

- A DNA string can be inserted at any place in within another string. Thus, **copy + paste** is at the heart of evolution.

- DNA replication could be imperfect. This produces mutation in the DNA content. **Mutation** leads to variability.

- A DNA string can fill in a function that is defined from the outside. Different strings may have different performance for the predefined function. This is called *selection*. We consider in biology organismic assessing, in which the string functions as a unity. For applied purposes, we also may consider modular assessing, in which substrings may be assessed separately.

- We can *compare* two strings to know whether or not they are equal. This is based on the fact that complementary DNA strands tend to match the stronger the more similar they are.

- Diverse operations can run at the same time on different places. Thus, evolution includes **parallelism**. This is usually ignored in our simulations but we need to come back later to study the implications of this fact.

Let us notice that deleting a portion of a string is equivalent to copying the rest from it. So, **deletion** is reducible to copying. By the same token, inserting a substring a at place $i-th$ of another string b can be reduced to copyPaste as follows: copy the substring of b to the left of place $i-th$, paste substring a, paste the part of string b at the left of its $i-th$ place. Instead of copyPaste, one can restore all the power of evolution with the function **replace** because copying string a is equivalent to replace the null string by a.

We see that at the DNA level, **evolution and text processing are synonymous** and that **copyPaste** seems to be enough to produce raw material for both text processing and evolution.

We also see that text processing produces variability which produces differential surviving + reproduction and so a escalate toward perfection might emerge in an evolutionary process. This is usually called **selection of the fittest**.

25 ***Exercise*** *Revise programs randomSearch (B11), and checkOut (B21) to see whether or not they could be classified as evolution based programs.* Answer

26 ***EVOLUTION EVERYWHERE***

As soon as we recognize that the basis of evolution is text processing, immediately we discover that evolution pervades all human verbal activities. That is why evolution is found everywhere where there are words.

2.4 A new weasel

We have seen how well evolution is to solve posed problem. Anyway, we feel challenged to devise a proficient version of weasel, a very efficient program that uses evolution to guess a predefined word or string.

27 ***A new try for a better weasel***

We see in the next code a new version of weasel specifically built to be proficient.

```
//Program NewWeasel, B27
//A string of length k is declared as bigGoal,
//which is  guesed by evolution.
//The number of generations and trials done to
//achieve the goal are reported.
//k is next increased by one.

//Proficiency is expected.
//To that aim, the bigGoal is partitioned
//in a ladder of subGoals of increasing length
//beginning with 1.

package ejvol2p;




import java.util.Random;

public class NewWeasel
{
```

```
//  Evolution of a population
//  of strings.

//  Strings are at the same time
//  genotype and phenotype.
//  They are like ribozymes.
//  Selection is organismic, in the sense that
//  the genome is assessed as a whole.

//  Individuals are kept in the array
//  Individual[]. It is an array of strings.
//  The number of individuals is equal to limit

//  The population has popSize members

//  For each i less than N,the program generates
//  a random string bigGoal of length i,
//  and then match it by evolution.
//  To that end, the program   defines and solves
//  a sucession of subgoals taking from bigGoal
//  the first k chars for 1<= k <= i.

static int maxLength;
static int popSize;

static String Individual[ ];//Predeclaration
static String Individual2[ ];
static String  individual ="";
static int stringLength=0;
//Order  is an array of integers.
static int Order[];//pre-declaration
static int gen;
static boolean problem=true;
static String bigGoal = "";
static String subGoal = "";
static String champString = "";
static int   Champ=0;
```

```
//Turn on of the random generator
static Random r  = new Random();

//This method generates random chars
private  static  char  randomChar()
{
    return  ( char ) ( r.nextInt( 26 ) + 65  );
}

// A random string of length stringLength is generated
private static String generateString(int stringLength)
{
    //System.out.println( "stringLength = " + stringLength);
    individual = "";
    //An individual is assembled char by char
    for(int j = 0; j < stringLength; j++)
    {
        // We generate a random char
        char c =  randomChar();
        //System.out.println( "char = " + c);
        // Char c is converted into a string s
        String s = ""+c;
        // s is concatenated to Individual[i]
        individual  = individual +s;
    }
    return individual;
}

// An initial population of strings is set up.
private static void createPop(int stringLength )
{
    //Formal declaration of our arrays.
    Individual= new String[popSize];
    Individual2= new String[popSize];
    //System.out.println("ORIGINAL POPULATION");
    for(int i = 0; i< popSize; i++)
    {
        Individual[i]=generateString(stringLength);
        Individual2[i]=Individual[i];
        //System.out.print("Individual " + i + " = ");
```

```
        //System.out.println( Individual[i]);
    }

    // Declarations and default initializations
    // of other arrays.
    Order= new int[popSize];
    for(int i = 0; i< popSize; i++)
            Order[i]=0;
}


//The more similar to the goal, the fitter.
//Method taken from weasel by Max Scott
static public int fitness(String subGoal, String indiv)
{
    //System.out.println(" indiv = " + indiv);
    int errors = 0;
    int len = subGoal.length();
    for (int i = 0; i < len; ++i)
        if (indiv.charAt(i) != subGoal.charAt(i)) ++errors;
    return errors;
}


//Individuals are sorted by fitness
private static boolean sorting(String subGoal)
{
    //System.out.println( "Subgoal = " + subGoal);

    //Default initialization
    int Fitness[];
    Fitness= new int[popSize];
    for(int i = 0; i< popSize; i++)
        Fitness[i] =popSize;
    for(int i = 0; i< popSize; i++)
    {
        //System.out.println(" Individual i" +Individual[i]);
        Fitness[i] = fitness(subGoal, Individual[i]);
        //Fitness 0 means no errors
        if (Fitness[i] == 0)
```

```
        {
           Champ = i;
           champString = Individual[i];
           problem =false;
        }
    }

    if (problem ==true)
    {
        //We sort individuals by fitness
        for(int i = 0; i< popSize;i++)
        {
            Champ = 0;
            for(int j = 0; j< popSize;j++)
                if (Fitness[j] < Fitness[Champ])   Champ = j;

            //The array Order classifies individuals by  fitness by
            //equal or decreasing order.
            Order[i] = Champ;
            Fitness[Champ] = popSize;
        }



    }
    // System.out.println("Gen = " +gen + " best solution = " +
Individual[Champ]);
    if (problem ==false)
    {
        System.out.println( "SubGoal = " + subGoal);
        System.out.println( "Task ended by generation = " + gen);
        //In each generation, popSize individuals are checked out.
        System.out.println( "Number of trials = " + gen*popSize);
    }
    return problem;
}

// The new generation is made
private  static void newGeneration()
{
```

```
    int   i=0;
    //The fittest individuals are reproduced
    // numbCopies times
    int numbCopies = 10;
    int   k=0;  //rank, 0 is the highest

    while (i<popSize)
    {
        int counter = numbCopies;
        while ((counter>0) & (i<popSize))
        {
            //System.out.println(i + "" + l);

            Individual2[i] = Individual[Order[k]];
            //System.out.println(i + " " + counter + " " + k);
            i++;
            counter= counter-1;
        }
        k++;
    }

    for(int j = 0; j<popSize; j++)
    {
        Individual[j] = Individual2[j];
        //System.out.println( j+ " length at newGen " +
        // "Pop1[j].length);
    }
}



//Recombination is enabled
private static String recInd(int l)
{
    String s = Individual[l];
    String  s1= "";
    int m= s.length();
    int m1 = 0;
    if (m>0)
    {
```

```
      m1 = r.nextInt(m) ;
     s1= s.substring(0,m1);
    }

    int k = r.nextInt(popSize);
    String t = Individual[k];
    int n= t.length();
    String  t2= t.substring(m1);

    String recombinant = s1 + t2;
    //System.out.println(s + "+" + t  + "->" + recombinant + "/");
    return recombinant;
}


// The definitive population  is a recombinant one.
private static void Recombination()
{

    //
    for(int l=0; l< popSize; l++)
       Individual2[l] = recInd(l );
    //Ind2 is transferred to Ind
    for(int i=0; i< popSize; i++)
       Individual[i] =  Individual2[i] ;
}




//String a is mutated at a random place.
private static String change(String a)
{
    if (a.length()==1)
    {

            return ""+randomChar();
    }
        else
    {
```

```
        //System.out.println("stringLength = " + stringLength);
        int l = r.nextInt(stringLength);
        //System.out.println("a in= " + a);
        //System.out.println("l = " + l);
        a = a.substring(0, l) + randomChar() +  a.substring(l + 1);
        //System.out.println("a out = " + a);
        return a;
    }
}

// Individuals are mutated.
//A copy of the champ is kept
private static void Mutation()
{
    for(int i=0; i<popSize; i++)
    {
        if (i != Champ)
            Individual[i] = change(Individual[i] );
    }
}

//The champ is cloned
//One random char is added to each individual
private static void  cloning()
{
    for (int i = 0; i<popSize ; i++)
    {
        Individual[i] = champString ;
        Individual[i] = Individual[i] + randomChar();
        //System.out.println(Individual[i]);
    }
}

//Given a bigGoal, it is guessed by evolution
private static void  evolSolution(int k)
{
    //A initial population is created
    stringLength = 1;
    createPop(stringLength);
    //Big problem is divided in a ladder of subproblems
```

```
    for(int i = 1; i<=k; i++)
    {
        stringLength = i;
        problem = true;
        subGoal = bigGoal.substring(0, i);
        //System.out.println( "subGoal " + i+ " = " +subGoal);
        //Individuals are sorted by length
        problem = sorting(subGoal);
        while (problem)
        {
            newGeneration();
            Recombination();
            //The new population is subjected to mutation
            Mutation();
            gen++;
            problem = sorting(subGoal);
        }
        if (i<k)
        {
            cloning();
            gen++;
        }
    }
}

public static void main(String[] args)
{
   // The population has popSize members
   popSize = 20;
   maxLength = 28;
   for(int k = 1; k<=maxLength; k++)
   {
        System.out.println("Length of bigGoal = " + k);
        bigGoal = generateString(k);
        System.out.println("bigGoal = " + bigGoal);
        gen = 1;
        while (problem)
          evolSolution(k);
        problem = true;
        System.out.println();
```

```
    }
  }
}//End of main class NewWeasel, B27
```

28 ***Exercise*** *Run the code and decide about its proficiency. Analyze the code to understand what the strategy that was followed to solve the problem is.*

29 ***Why do we use regression analyses?***

Our research versions of weasel were made to study the number of generations needed to synthesize a string equal to another that was given a priori. In principle, one can take a random string and do the job, and then one might consider that the problem has been solved. For us, that is not allowed. For suppose one can buy a house with N euros. Can we say that we can buy 1000 houses with 1000N euros?

Most probably not. The very first reason is that you shall be investigated for having so much money and you do not know what corrupted people can make against you. Next, because of so much money you shall invest a significant part of it to keep the rest safe along with you and with the persons you love.

So and as a rule, an enlarged problem is not solved by just enlarging the solution. That is precisely why we always want to know what happens to our programs if we enlarge problems just a bit (we are very far from being able of studying large enlargements).

On other hand, we use randomness at the core of all our simulations. But it happens that randomness can be included into evolution in many diverse ways, simulating myriads of diverse sources and we always want to compare some of them to see which the best is. The only way we have to decide that is by letting the diverse sources or distributions to solve problems that are more and more complicated: Our WeaselResearch is better than randomness, but Weaselresearch is not as proficient as our NewWeasel, which in its turn is superseeded by CheckOut. The natural way to proof these assertions is through regression analysis.

These considerations invite us to be on guard against a usual methodological error which consists in thinking that evolution explains something because an academic solution, made by high qualified experts, provides the explanation. Thus, *we must learn to distinguish between evolution as a wonderful human idea and evolution as a natural phenomena*. This implies that our community must work in union with other scientists, say with specialists in bioinformatics.

2.5 Conclusion

Randomness can explain everything. But a price must be paid: the number of trials to match a target string was an exponential function in its length. That is why

randomness is not the preferred explanation of complexity in a limited universe as ours. The preferred option of Science today is Evolution, which consists in the recursive application of 1) assessing the capability to fulfill a given function by the members of a parental population and 2) to confer imperfect reproduction of individuals in agreement with performance. In organismic evolution, the performance of an individual is made in toto, after the phenotype. But in simulations, an individual is a string and one can evaluate substrings directly. This freedom allows us to consider that evolution is equivalent to text editing as it happens in Molecular Biology with DNA. Once it is recognized that evolution and word processing is the same thing, one concludes that evolution pervades all human verbal activities. Additionally, we have witnessed how well is evolution to build complexity as accumulation of small change, a skill that was enough to solve the retro-engineering problem of matching a target string. In general, evolution is a universal tool to solve optimization problems but it not necessarily better than other forms of programming. Nevertheless, sometimes it could be desirable for us to program evolution to find the know-how of solving a problem instead of solving the problem by ourselves. That is why evolution is used in modern technology and industry albeit not too much. A question remains: is evolution almighty?

Chapter 3

The genome is software

`DNA contains messages for a programmable computer.`

30 ***Purpose*** *We prove that the genome is a set of verbal instructions for a programmable computer and so it is software. The theory of evolution now reads: evolution is the software developer responsible for the existence of the genome.*

3.1 General terminology

There are some usual terms that we specify below.

31 ♣ ***Definitions***

We, human beings, use languages to communicate one with another. A **language** is composed of verbal instructions, a code of interpretation and a semantic content. A **verbal instruction** is a string of sounds or letters that belong in an **alphabet**. Verbal instructions must be deciphered according to a **code of interpretation**, which appears, say, in a dictionary. The deciphered verbal instructions convey information about the world or about actions that must be undertaken and it is in this way that they acquire **semantic content**.

The Phoenicians are thought to have developed the first alphabet around 1400 BC, i.e., they partitioned the verbal sounds into syllables. Syriobabylonians used a cuneiform writing with around 500 symbols for their syllables. Next, vocal sounds were partitioned into elementary phonemes that could be represented by a letter, as it was done by Greeks and modern alphabets.

A **computer** is a devise that has the potential ability of executing verbal instructions. A **program** is a set of verbal instruction, written in a specific **programming**

language that can be interpreted and executed by a computer. **Code** is a set of instructions that eventually contain correctly written programs. **Software** is a set of programs to execute a task. **Hardware** is the wiring that converts programs into specific actions. The core of the hardware is the microprocessor that contains the central processing unit, which can understand verbal instructions that are written in machine language, which is composed of words whose alphabet is just $\{0, 1\}$. An **operating system** is a set of programs specifically targeted to serve the needs of communication between human users and the hardware: it contains the know-how of operating the computer to execute a verbal instruction that commands to, say, print a letter on the console; it also produces feedback from the hardware, say, it could communicate that there is no network connected to the computer. Simple microprocessors are not enabled to admit operating systems apart than the one contained in its language machine. A **robot** is a machine that has a computer that has been programmed to have certain independence.

A **designer** is a person that is committed to the design of software, i.e., of programs with a specific function, which is defined before beginning but that may be modified along the process. It is absolutely sure that the first trial of a code is inviable. But if one manages to produce a viable code, most surely it will not do what one was so eagerly expecting. The process of adjusting the code to the predefined purpose is called **debugging**.

In contrast to software designer, which is a person with a goal, we also have the concept of **software developer**, which is any process or entity that produces software. The point studied in this volume is that evolution is a software developer whose possibilities are worth being studied both to explain the genome and to be used in industrial software design.

We highlight the fact that the concept of bug has a meaning in software design but it lacks any meaning in software developing. The reason is that a bug is a fault against a goal, a project, a predefined function, while software developing may run without predefined goals by mere diffusion. Of course, the products of software developing may be used by exterior entities or processes to fill functions whose performance can be assessed and improved. This classifies as artificial selection. By contrast, natural selection as considered in the Evolutionary Theory has no target at all.

3.2 The ribosome is a programmable computer

The **genome** contains, apart from other things, a set of instructions to program ribosomes to build specific proteins. Portions of DNA are transcribed into RNA, whose codons are interpreted by t-RNA's and the ribosome to build proteins. A

good introduction to molecular biology ca be found in Wikibooks [55] (2012) . The whole ensemble functions like a programmable computer: one can order the synthesis of whatever protein one wants. One could also say that the ribosome with the t-RNA's form an operating system. Let us see that the genome is indeed software: it is based on an alphabet, it has code and obeys a given semantics.

32 ***The alphabet***

For DNA, the alphabet is $\{A,T,C,G\}$. For RNA, it is $\{A,U,C,G\}$. Anyway, there are other molecules that also count, say, the TT dinucleotide, which is formed by illuminating with ultraviolet rays the DNA. There are also molecules that can be inserted, maybe by van der Wans interactions, into a DNA string producing lethal consequences.

33 ***The code and the semantics***

The code that allows the deciphering of the verbal instructions contained in the genome is the **genetic code**, a dictionary that contains the correspondence among **codons** and amino acids. This dictionary is written in the t-RNAs: these molecules have in one extreme an anti- codon (complementary to a codon) and in the other the corresponding amino acid. Amino acids are assembled together by the ribosomes into proteins, which enable surviving in one of two ways: they may serve as structural items or as enzymes to catalyze chemical reactions. Thus, **surviving** is the feature that allows the genome to acquire a semantic content: a viable string has a meaning and unviable string does not. A **cell** is a robotic entity: it is directed by a software and has a lot of independence.

So, the genome is for us the natural example of software. The theory of evolution now reads: evolution is the software developer responsible for the existence of the genome. Thus, the evolution of the genome is a sub discipline of a more general matter: the study of the evolution of software in general. In fact, our study of **Java**, our platform of software design, rises many questions of biological interest and especially in relation with the genome.

The vision of the genome as software took some 80 years to be created and strengthened. That began with the speculation of the possible creation of computers by Alan Turing and John von Neumann in the 1830's and continued with all those technological achievements in computing science and in molecular biology that we see today and that ended with the design of DNA computers (Adami, [2] 2007).

Most probably, this vision of the genome as software and of evolution as software developer will appear in the history of science as the contribution of our generation to the understanding of fundamental biology. In consequence, let us invest some time and effort to understand it much better.

3.3 Software and dynamical systems

Let us see why computers are very different than the rest of physical systems.

34 ***Inertial vs free movement***

Large physical systems, stones and cars, have a tendency to follow in the same direction and with the same velocity unless external forces, say friction, exert some action over them. In general, the physical word is in first instance characterized for its tendency to behave just in the form it behaved in the past. By contrast to matter, we are living beings that, although subjected to physics, can manage situations to behave as free entities, to do what we plan to do. We are men and women that are so conscious of our freedom that in most cultures we are considered to be responsible for our words and deeds.

Many differences could exist among mater and life but we must highlight at this very moment the existence of a robotic facility possessed by living beings, which is a system of computation to calculate decisions plus an executive system to obey received commands. That facility enables free movement.

Robots are beginning to invade our lives, so we are completely sure that they have nothing else apart from highly organized matter. So, we ask: if a robot is only matter, how does it acquire freedom from the inertial behavior?

The answer is this: a robot never ceases to obey physical laws, it is precisely because of them that robots function. But the physical laws enable robotic freedom as follows. In first instance, some physical systems may behave erratically, say, the trajectory of particle that travels inside the water of a turbulent river cannot be predicted. The behavior of such a particle is so distinct than that of a stone that science invented a whole world for its study: **chaos**. The form as a software developer perceives the almost chaotic nature of the world he or she lives inside is through bugs: a minor difference may cause tremendous changes that lead to death, non viable code, else to a behavior very different than the one that is searched for. But we know that a chaotic personality is not desirable for a human being. The matter is that humans are free but controllable. So is software.

Thus, most humans and all living creatures and all robots and all computers enjoy two properties: they are almost chaotic and also are controllable. That is easy to say but it is very difficult to implement and that is why computers were only recently built. At the heart of computers, we find systems that may answer all or nothing and that can be as simple as a diode or as a transistor. Elements that answer all or nothing are example of **digital** systems. DNA bases are digital in the sense that they recognize their complementary partners and so they answer yes or not to an incoming molecule. Combining simple elements, one can achieve great levels of complexity, such as those we see in any programming language.

So, software is what makes the difference between stones and free will of action. Most probably, there is no higher exaltation of the power of software than that presented by Apostle John. Of course, he does not uses the word software, instead he uses *word*, the mean that human beings use to think, to make calculations, to express decisions and to teach, i.e. to program other people. He opens his Gospel with the next words:

$Εν\ αρχη\ ην\ ο\ λογος,\ και\ ο\ λογος\ ην\ προς\ τον\ Θεον,\ και\ Θεος\ ην\ ο\ λογος.$

A suggesting transcription could be:

En arkhe en o logos, kai o logos en pros ton Theon, kai Theos en o logos.

A suggested translation could be:

At the beginning was the Word, and the Word was with God, and God was the Word.

The official translation is:

In the beginning was the Word, and the Word was with God, and the Word was God (The King James Version, Cambridge 1769).

Now, let us apply this ideology to our mechanistic context: the word, the software, is so great that it converts a stone into a god, a silicon chip into a robot. But, software and evolution is the same because we have today DNA computers. So, according to ancient Christians, evolution is the brand of the Divinity. From that stand, it is insane to accept cheap explanations of the origin of evolution, as it is also to consider that evolution is unable of wonders. Two challenges to illuminate and inspire our lives.

3.4 Conclusion

Software is composed of a set of strings in an environment that contains a code of interpretation, an interpreter and a executer that can associate to each string a specific action whose fruits determine the semantics of the strings. The genome is the natural example of software: the strings are composed of DNA bases, the code of interpretation is the genetic code, the interpreter is the ensemble of t-RNA's, the primary executer is the ribosome, which converts DNA or RNA strings into protein chains. The function fulfilled by these chains in their capability of enabling or improving life in a given milieu defines the semantics of the DNA messages. These considerations prompt a challenge: if the genome is software, is evolution a software developer? Yes, we feel that it is. But, can we prove that?

Chapter 4

Evolution as software developer

```
Evolution programs a virtual robot (next three chapters).
```

35 ***Motivation.***

We have shown that **the genome is the natural example of software. The theory of evolution now reads: evolution is the software developer responsible for the existence of the genome**. We have clearly met a challenge: if evolution is the responsible of the existence of the genome, we must be able to tame the computing power of evolution to design our own software to produce programs with as excel quality as that we see in living creatures from bacteria to a dancing girl.

36 ***Purpose.*** *We run a simulation of evolution, whose object is to design an algorithm to teach a worm that appears on the screen to crawl and climb it. We begin with a design, with our own intelligence, of the algorithm that makes the worm to crawl. Next, we implement an evolutionary algorithm that must discover the know-how of crawling. We will proceed step by step. The whole work is contained in the next three chapters.*

4.1 The worm

37 ***The body.*** *Our worm is articulated: it consists of 12 links. All twelve parts or cells have one unit length. The 12 parts can move independently but the structure of an organism must be kept: movements cannot separate the organism which must remain in one piece. Every link has 8 possibilities of movement plus another one of making nothing. We are allowed to move over a reticulate.*

One shall check whether or not a given movement of a link disrupts the organism. For instance, if the worm is totally stretched, a forward movement of the head would disrupt the organism and so it is not allowed. In that case, allowed movements are sidewards.

Every part can move autonomously but the organism must be kept in one piece. We posit the worm at the bottom of our screen and its task is to arrive at the upper edge in the minimum number of moves. Since movements must be correlated to conserve the structure, movements are packed into a unit, the chromosome. The name of this unit is justified because we are committed to investigate in which form evolution can fill the chromosome in the correct information. The structure of the worm is shown in the next figure:

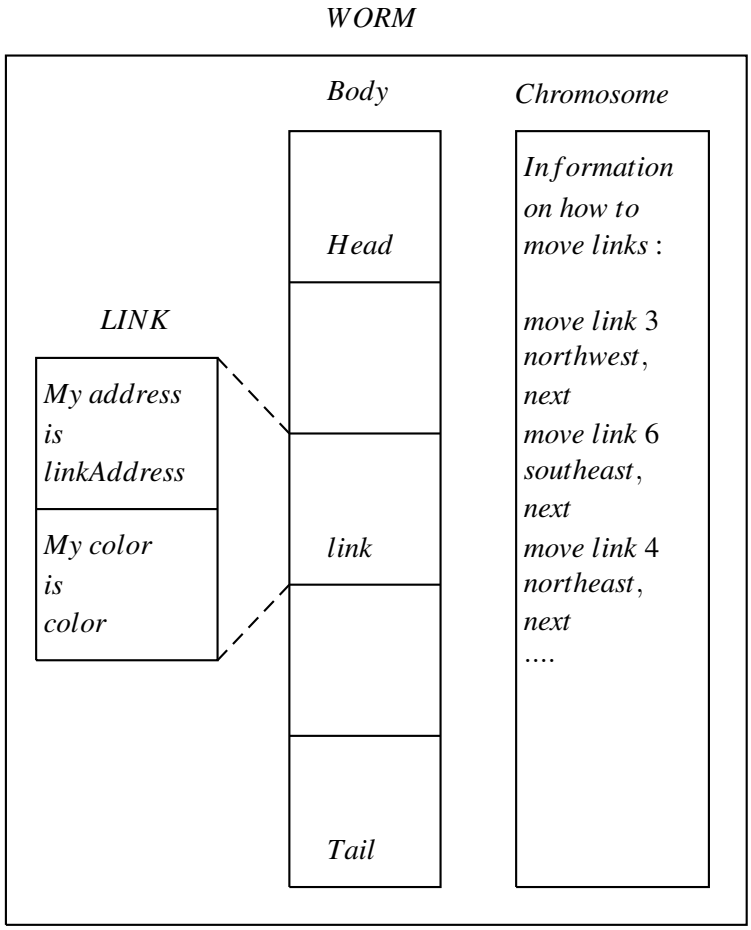


Figure 4.0. The structure of a worm: a body with various links and a chromosome that keeps the information on how to move links.

38 *Programming movement*

The next program presents a solution, devised by the Author, to the movement problem. The natural language used in the previous figure to describe the composition of a worm will be fully achieved with a posterior use of Object Oriented Programming. For now, we use ordinary technology.

```
//Program WormHumanDesign, B38

//A worm that crawls.
//Human design.

package ejvol2p;




import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Color;
import java.util.Random;

public class WormHumanDesign extends Applet
{

private static final long serialVersionUID = 1L;

//Turn on  the random generator
static Random r  = new Random();

static int bodyAddress[ ][ ] =new int[13][3] ;
static int colors[][];
static int initialRow;
static int   i, j,x,y;


private void    initialPosition()
{
```

```
    int k;
    //The worm has 12 links,
    //each part has its own coordinates on the screen,
    //which are kept in a 12x2 array.
    //This is the initial position of the head
    initialRow = 40;
    //links form a column
    for(int l = 1; l<=12;l++)
    {
        k = initialRow;
        bodyAddress[l][1] = 50;
        bodyAddress[l][2] = 12*l+k;
    }
}


private void drawBody(Graphics g)
 {
    //Paints the body
    // to init
    for(int l = 1; l<=12;l++)
    {
        x = bodyAddress[l][1];
        y = bodyAddress[l][2];
        //Each link has a color.
        //A color can be defined by three numbers,
        //each one from 0 to 255.
        //We has 12 links, so we need
        //a matrix to keep colors.

        int c1,c2,c3;
        c1 = r.nextInt(226);
        c2 = r.nextInt(226);
        c3 = r.nextInt(226);
        //Colors are memorized
        colors[l][1]= c1;
        colors[l][2]= c2;
        colors[l][3]= c3;
        g.setColor( new Color(c1,c2,c3));
```

```
        // A Rectangle is defined by
        // the coordinates of the
        // superior right corner,
        // width and hight.
        g.fillRect(x,y,4,12);
    }
}


private void delay(long l1,int s)
{
    while  (System.nanoTime() < l1 + s) {delay(l1,s);}
}


private  void  crawl(Graphics g)
{
    // The worm moves link by link
    // but no disruption to the body
    // could be done.
    // Hence, movement must begin from the tail.

    int   l, m, n;

    //To move a link:
    // the link is erased (painted in white),
    //and a new link is painted
    // in the new place


    for(l = 1; l<= 68; l++)//cycles of movement
    {
        //Moves in zig-zag
        int k =  (int) (Math.pow(-1,l ));

        for(m = 1; m<= 12; m++)//links
        {
            //The old link is erased
            n = 13 - m;
            i = bodyAddress[n][1];
```

```
            j = bodyAddress[n][2];
            g.setColor(Color.white);
            g.fillRect(i,j,4,12);

            // a new place is defined
            i = i + 4*k;
            j = j - 6;

            bodyAddress[n][1] = i;
            bodyAddress[n][2] = j;


            //Move=paint in the new place;

            g.setColor( new
            Color(colors[n][1],colors[n][2],colors[n][3]));
            g.fillRect(i,j,4,12);
            //If movement is too fast, nothing can be seen.
            //To attune the velocity of the worm,
            //augment or diminish the delay as   pleased
            long  l1 = System.nanoTime();
            int s = 500; //change this.
            //   change the upper bound of j
            for(int h=1; h<=2000000; h++)
                {delay(l1,s);}
        }
    }
}


@Override
public  void  paint(Graphics g)
{
    //Here is the start.
    colors = new int[13][4];
    initialPosition();
    drawBody(g);
    crawl(g);
}
}//End of main class WormHumanDesign, B38
```

39 ***Exercise*** *Run the program and play with the code: modify it to have the worm slide toward the left, to the right, backwards. Make a worm 7, 8, 11 links long. Make a worm with a double width.*

40 ***The space of moves***

If we use Cartesian coordinates and we imagine that the initial position of the link is at (0,0), then the nine possibilities of movement are:

(0,0); (0,1);(1,1);(1,0); (1,-1); (0,-1);(-1,-1);(-1,0);(-1,1);

Say, (1,1) means that the movement is in the northeast direction. Similarly, (0,-1) means that the movement is done in the south direction. And so on.

41 ***Exercise*** *Prove that the 9 listed allowed movements do not exhaust all possibilities: in some occasions one can move two units apart and nevertheless the organism does not get disrupted. Anyway, a movement of two units can be achieved using two moves in tandem, so we will ignore a movement two units long.*

42 ***Exercise*** *Prove else refute that the space of moves is both sufficient and not redundant, i.e., with the nine listed possibilities one can devise the solution to the proposed problem (to climb the screen) but none can be deleted.*

4.2 Evolution in perspective

Let us move now into evolution, which is today a method to solve any type of problems. Ours is to teach the worm to crawl as fast as possible, i.e., to provide the software to program the worm to move upwards without disrupting itself. We already know how to do that but we will search a way to begin from nothing and, with the only help of evolution, to end with a very wise worm.

43 ***Translation to evolutionary language***

Our first task is to translate the human solution to our problem into the language of evolution, which is understood here as composed of two items: 1) The assignation of a reward to any information set according to its performance. 2) The reward is the right to get possibly modified copies of the same information to be passed into the next generation.

More mechanistically: we pose a population of worms at the bottom of the screen, whose information set responsible for their movement has been assembled at random. Worms may move and reproduce but the number of offspring of each worm is proportional to the velocity of the mother to arrive to the top of the screen.

The offspring of each worm may have its set of instructions different than that of its mother, with mutations.

To implement this program, let us take some steps. The first is to rewrite the code of the algorithm designed by a human in the required form, in an evolutionary language. Our set of instructions is as follows:

Select the link of the tail of the worm and have it move $(\triangle x, \triangle y) = (1,1)$; do this for each link from tail to head, in that order. Repeat this over and over until finishing.

Our set of instructions is highly informative because 1) It does the task, 2) It does the task in the least possible time and 3) There are a lot of sets of instructions that has nothing to do with the task or with its optimization.

44 ***Chromosomes*** *Cellular chromosomes keep the genetic information, which intervene in the architecture and construction of the biological being and also enables the struggle for life. In regard with our worm, we define a chromosome as a set of instructions (genes) that says how links must be moved, in which order and in which direction, i.e., with which move among the nine possible. So, we may posit all instructions one after the other in a vector with 2 columns and 13 rows. In each row we keep an instruction: which link to move is kept in the first column and with which move in the second column. The software contained in the chromosome is decoded, interpreted and executed by a unit that we call ribosome and that plays the role of a CPU of a computer: the instructions in the chromosome are executed according to their position beginning with that in the first place. At present time, we need no more complexity. The following program contains the code:*

```
//Program WormRibosome, B44
//A worm that crawls.
//Human design
//in evolutionary language.


package ejvol2p;



import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Color;
import java.util.Random;
```

```
public class WormRibosome extends Applet
{
static int bodyAddress[ ][ ] =new int[13][3] ;
static int initialRow;
static int colors[][];
static int chromosome[ ][ ] =new int[13][3] ;
static int vectMoves[ ][ ] =new int[13][3] ;
static int side;
static int   i, j,x,y;

//Instead of four cardinal points,
//we have 8 + 1(nothing).
//They correspond to moves
//along any of the 9 cardinal directions.
//
//The 9 moves are kept in an array
//with 9 rows and 2 columns.
//The first column keeps the x direction
//The second column keeps the y direction.
static int vectorMoves[][]= {   {0,1},
                                {1,1},
                                {1,0},
                                {1,-1},
                                {0,-1},
                                {-1,-1},
                                {-1,0},
                                {-1,1},
                                {0,0}
                                  };


private void   rewrite()
{
    //Numeration is begun from 1
    for(int l = 1; l<=9;l++)
    for(int m = 1; m<=2;m++)
      vectMoves[l][m] = vectorMoves[l-1][m-1];
}
```

```
private static final long serialVersionUID = 1L;


//Turn on  the random generator
static Random r  = new Random();


private void    initialPosition()
{
    int k;
    //The worm has 12 links,
    //each part has its own coordinates on the screen,
    //which are kept in a 12x2 array.
    //This is the initial position of the head
    initialRow = 40;
    //links form a column
    for(int l = 1; l<=12;l++)
    {
        k = initialRow;
        bodyAddress[l][1] = 50;
        bodyAddress[l][2] = 12*l+k;
    }
}


private void drawBody(Graphics g)
{
    //Paints the body
    for(int l = 1; l<=12;l++)
    {
        x = bodyAddress[l][1];
        y = bodyAddress[l][2];
        // Each link must be colored
        int c1,c2,c3;
        c1 = r.nextInt(226);
        c2 = r.nextInt(226);
        c3 = r.nextInt(226);
        //colors are memorized.
        colors[l][1]= c1;
```

```
        colors[l][2]= c2;
        colors[l][3]= c3;
        //the link is painted
        g.setColor( new Color(c1,c2,c3));
        // A Rectangle is defined by
        // the coordinates of the
        // superior right corner,
        // width and hight.
        g.fillRect(x,y,4,12);
   }
}


private void initialChromosome(Graphics g)
 {
    // The instructions that choose what to do
    // are defined and kept in the chromosome.
    int n;
    for(int m = 1; m<=12;m++)
    {
        n = 13 - m;
        //Move the tail first and then the others links
        chromosome[m] [1]= n;
        //All links execute the same move:
        //Move northeast (or northwest)
        //Northeast: (1,1), whose number is 2,
        //Northwest: (-1,1), whose number is 8.
        //Which side is decided by the variable Side,
        //0 means  northwest,
        //1 is  northeast.
        //Initial direction: Northeast=2.
        //because Side = 0
        chromosome[m] [2] = 2 + 6 * side;
     }
 }


private  void  rightLeft(Graphics g)
{
    //Advance in zig-zag
```

```
    if (side == 0) side = 1;
       else side = 0;
    initialChromosome(g);
}


private void reproduction()
{
    //Our worm needs no reproduction
    //because it is perfect.
}


private void delay(long l1,int s)
{
    // to attune the velocity of movement
    // that it could be observed by a human.
    while  (System.nanoTime() < l1 + s) {delay(l1,s);}
}

private  void  ribosome(Graphics g)
{
    // The ribosome reads, interprets
    // and executes the instructions
    // packaged in the chromosome.
    // The chromosome is the software,
    // the ribosome is the CPU.
    //
    // The worm moves link by link
    // but no disruption to the body
    // could be done.
    // Hence, movement must begin from the tail.

    int   n;

    //To move a link:
    // the link is erased (painted in white),
    //and a new link is painted
    // in the new place
```

```
for(int l = 1; l<= 68; l++)//cycles of movement
{
    //Moves in zig-zag
    rightLeft(g);

    // All links must be moved
    for(int m = 1; m<= 12; m++)
    {

        //To move a link:
        //erase the rect in the old position
        //and draw a new rect in the new position
        //The former rect is erased
        n = 13 - m;
        i = bodyAddress[n][1];
        j = bodyAddress[n][2];
        g.setColor(Color.white);
        g.fillRect(i,j,4,12);

        // A new place is defined
        // The move to be done is read
        int   move = chromosome[m][2] ;
        i = i + 4*vectMoves[move][1] ;
        j = j - 6*vectMoves[move][2] ;
        // The new place is memorized
        bodyAddress[n][1]  = i;
        bodyAddress[n][2]  = j;


        //Move=paint in the new place;
        g.setColor( new Color(colors[n][1],colors[n][2],colors[n][3]));
        g.fillRect(i,j,4,12);
        //If movement is too fast, nothing can be seen.
        //To attune the velocity of the worm,
        //augment or diminish the delay as   pleased
        long  l1 = System.nanoTime();
        int s = 500; //change this.
        //   change the upper bound of j
        for(int f=1; f<=2000000; f++)
```

```
            {
                 delay(l1,s);
            }
            reproduction();
        }
    }
}



@Override
public  void  paint(Graphics g)
{
    //Here is the start.

    // A color can be defined by three numbers,
    // each one from 0 to 255. Each link is dressed
    //in its own color: we need a matrix to keep colors.
    //
    colors = new int[13][4];
    rewrite();
    initialPosition();
    drawBody(g);
    side = 0;
    initialChromosome(g);
    ribosome(g);
}

}//End of main class Program WormRibosome, B44
```

45 ***Exercise*** *Run the code, understand and verbalize it.*

46 ***Exercise*** *Modify the code to produce worms with various phenotypes.*

47 ***Back to OOP: worms at the start line*** *Let us rewrite our program using objects that are encoded in inner classes. In the next program we see how this approach allows us to immediately posit twenty worms at the start line of a race.*

```
//Program Worms20, B47
```

```
//Twenty worms are placed at the
//start line of a race.
//OOP version.


package ejvol2p;




//Needed tools
import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;
import java.util.Random;


public class Worms20 extends Applet

{
//Class Worms20 is an idea, a design.
//The next instruction says that
//one is allowed to
//construct a real stuff, an instance, on the
//image and likeness of the design.

public Worms20()// constructor
{}

//We command to make a real instance,
//worms20,
//in the image of the archetype, Worms20.
static  Worms20 worms20 = new Worms20();

//Turn on  the random generator

static Random r  = new Random();

private static final long serialVersionUID = 1L;
```

```
// We follow the next convention:
// Prototypes are written in upper case,
// instances are written in lower case

// In java, archetypes are called classes.
// It is not mandatory, but
// we define archetypes here as Inner classes,
// as classes in within another one.



// An archetype may have
// two types of information:
// one about things
// The other about tasks

private  class Move
{
    int x; // this is a thing
    int y;

    // The next is a method, a function
    // that  declares
    // how information is acquired.
    // It is a constructor method
    Move(int a, int b)
    {
        x= a;
        y = b;
    }
}

// An Exon keeps two types of information:
// which is the link of the worm to be moved
// and with which move.
private   class Exon
{
    int link ;
    Move move;
```

```
    // Exons can acquired information
    // in two ways: They have the same name
    // but the context says which manner
    // is to be used.
    // This is called overloading of methods

    Exon(int a, Move m)
    {
        link= a;
        move = m;
     }

    Exon(int a, int b, int c)
    {
        link= a;
        move.x = b;
        move.y = c;
    }
}


// The class LinkAddress keeps the actual
// position of a link.
private   class LinkAddress
{
    int x;
    int y;

    LinkAddress(int a, int b)
    {
        x= a;
        y = b;
    }


//  private int getX(){return x;}
//  private int getY(){return y;}
}

// Here we define the archetype of worm.
```

```
// It has a chromosome that keeps
// the information on how to move its links.
// Each chromosome has nExons instructions
// Each worm has its own color.
// Each worm keeps record of
// the actual position of each
// one of its links
private class Worm
{
    Exon chromosome[ ] ;
    int color[];
    LinkAddress  linkAddress[ ];

    public Worm()
    {
        chromosome = new Exon[20];
        color = new int[3];
        linkAddress = new LinkAddress[20];
    }
}


    //Global  Pre-definition=
    //intention  to work with them
      static Move vectMoves[];
    static Worm worm[];
    static Worm wormTwo[];
    static int colors[][];

    static int nWorms;   //Less than 20
    //static int  lifeSpan;
    static int initialRow;
    /*static int  order[];
     static int numbPointMutations;
    static int velocity;
    static int nOriginalexons, ncopies;*/
    static int nLinks, nExons ;
    //static int basalColor ;
    //static int nGen;
    //static boolean flag ;
```

```
    //static boolean  thereWasAdvance;

    private  void constructors()
    {
        System.out.println("constructors began");
        //Definition, assignment of memory.
        vectMoves = new Move[10];
        worm = new Worm[21];
        //wormTwo = new Worm[21];

        // a color is defined by three numbers,
        // and we have 20 worms
        colors = new int [21][4];
    }



static private void initialPosition(Graphics g, Worm worm[])
{
    int k, a,b;
    //The index W distinguishes each worm.
    //Each worm has its color
    //Each worm has nLinks links,
    //each part has its own coordinates on the screen,
    //which are kept in the linkAddress   array.
    //The worms are posited at the bottom of the screen.
    //Head are at row InitialRow
    for(int  W = 1; W<= nWorms; W++)
    {
        // We define an auxiliary worm
        Worm  worm1;
        worm1 = worms20.new Worm();
        //Each worm gets  colored
        int c1,c2,c3;
        c1 = r.nextInt(226);
        c2 = r.nextInt(226);
        c3 = r.nextInt(226);
        //colors are memorized.
        colors[W][1]= c1;
```

```
        colors[W][2]= c2;
        colors[W][3]= c3;

//All worms are posited at the same initial row
//in vertical position
for(int  l = 1; l <= nLinks; l++)
        {
            k = initialRow - nLinks;
            a = 20 + 20 * W;
            b = k + 12*l;
            LinkAddress   ad = worms20.new LinkAddress (a,b);
            worm1.linkAddress [l] = ad;

            //the link is painted
            g.setColor( new Color(c1,c2,c3));
            // A Rectangle is defined by
            // the coordinates of the
            // superior right corner,
            // width and hight.
            g.fillRect(a,b,4,12);
        }
        // Memorization
        worm[W] = worm1 ;
    }
}


@Override
public  void  paint(Graphics g)
{
    nWorms = 20;
    nLinks = 12;
    initialRow = 100;
    g.drawString("Please, resize the Applet", 20,20);
    constructors();

    initialPosition (g,worm);
}
}//End of Program worms20, B47
```

48 ***Exercise*** *Run the program and play with the code. Carefully study it.*

49 ***Exercise*** *In the next code we see the same program but without comments. Read the code and compose a comment on each line or segment. Return to the previous commented code to remind yourself of forgotten concepts.*

```
//Program Worms20b, B49

package ejvol2p;


import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;
import java.util.Random;


public class Worms20b extends Applet

{

private Worms20b()// constructor
{}

static  Worms20b worms20b = new Worms20b();

static Random r  = new Random();

private static final long serialVersionUID = 1L;

private  class Move
{
    int x; // this is a thing
    int y;

    Move(int a, int b)
    {
        x= a;
        y = b;
```

```
    }
}

private   class Exon
{
    int link ;
    Move move;

    Exon(int a, Move m)
    {
        link= a;
        move = m;
     }

    Exon(int a, int b, int c)
    {
        link= a;
        move.x = b;
        move.y = c;
    }
}


private   class LinkAddress
{
    int x;
    int y;

    LinkAddress(int a, int b)
    {
        x= a;
        y = b;
    }

}

private class Worm
{
    Exon chromosome[ ] ;
    int color[];
```

```
    LinkAddress  linkAddress[ ];

    private Worm()
    {
        chromosome = new Exon[20];
        color = new int[3];
        linkAddress = new LinkAddress[20];
    }
}


    static Move vectMoves[];
    static Worm worm[];
    static Worm wormTwo[];
    static int colors[][];

    static int nWorms;   //Less than 20
    static int initialRow;
    static int nLinks, nExons ;

    private  void constructors()
    {
        System.out.println("constructors began");
        vectMoves = new Move[10];
        worm = new Worm[21];
        colors = new int [21][4];
    }



static private void initialPosition(Graphics g, Worm worm[])
{
    int k, a,b;
    for(int  W = 1; W<= nWorms; W++)
    {
        // We define an auxiliary worm
        Worm  worm1;
        worm1 = worms20b.new Worm();
        int c1,c2,c3;
```

```
        c1 = r.nextInt(226);
        c2 = r.nextInt(226);
        c3 = r.nextInt(226);
        colors[W][1]= c1;
        colors[W][2]= c2;
        colors[W][3]= c3;

for(int  l = 1; l <= nLinks; l++)
        {
            k = initialRow - nLinks;
            a = 20 + 20 * W;
            b = k + 12*l;
            LinkAddress   ad = worms20b.new LinkAddress (a,b);
            worm1.linkAddress [l] = ad;

            g.setColor( new Color(c1,c2,c3));
            g.fillRect(a,b,4,12);
        }
        worm[W] = worm1 ;
    }
}


@Override
public  void  paint(Graphics g)
{
    nWorms = 20;
    nLinks = 12;
    initialRow = 100;
    g.drawString("Please, resize the Applet", 20,20);
    constructors();

    initialPosition (g,worm);
}
}//End of Program Worms20b, B49
```

50 ***Exercise*** *We have colored all links of the same worm with the same color. Make a suitable modification in the previous code that each link could have its individual color. Display the result on the screen.* Answer

51 ***Worms move: human design***

Let us program with our intelligence our OOP worms to climb the screen.

```
// Program WormsClimb, B51

// A population of worms
// is programmed
// to climb the screen.
// (Human design).
package ejvol2p;


// Needed tools
import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;
import java.util.Random;

public class WormsClimb extends Applet
{
// Class WormsClimb is an archetype.
// The next instruction says that
// one might
// construct a real stuff, an instance, on the
// image and resemblance of the archetype.

// The word archetype does not belongs
// in evolutionary biology,
// but it belongs in paleontology.
// We use here the word archetype as
// a synonym of a plan or design.
//
public WormsClimb()// constructor
        {}

// We command now to make a real instance
// in the image of the archetype, wormsClimb.
static  WormsClimb wormsClimb  = new WormsClimb();
```

```
//Turn on  the random generator

static Random r  = new Random();

private static final long serialVersionUID = 1L;

/*  We follow the next convention:
    Archetypes are written in upper case,
    instances are written in lower case

    In java, archetypes are called classes.
    It is not mandatory, but
    we define archetypes as  classes
    in within another one: as inner classes.

    Definition of the Move class:
    A Move says in which direction
    one must move:
    we have 8 cardinal points.
    Each point denoted by
    its x and y coordinates.

    An archetype may have
    two types of information:
    one about things
    The other about tasks */

    static private  class Move
    {
        int x; // this is a thing
        int y;

        // The next is a method, a function
        // that  declares
        // how information is acquired.
        // It is a constructor.
        // It is overload by two possibilities.
        // a) The input is a list
        Move(int a, int b)
        {
```

```
            x= a;
            y= b;
        }

        // b) The input is a structure
        Move(Move move)
        {
            x= move.x;
            y= move.y;
        }

    }


    // Definition of the class Mover:
    // A Mover keeps two types of information:
    // a) the link of the worm to be moved and
    // b) the direction or move to be chosen.
    private   class Mover
    {

        int link ;
        Move   move;

        //  Movers can acquired information
        // in two ways, they are  overloaded.
        // Form 1: by a list

        private Mover(int a, int b, int c)
        {
            link= a;
            move.x = b;
            move.y = c;
        }

        // Form 2, by structured data
        private Mover(int a, Move m)
        {
            link= a;
            move = m;
```

```
        move.x = m.x;
        move.y = m.y;
        }
    }

// The class LinkAddress keeps the actual
// position of a link.
private   class LinkAddress
{
    int x;
    int y;

    // LinkAddress is overloaded:
    // A list feeds it
    LinkAddress(int a, int b)
    {
        x= a;
        y = b;
    }

    //Structured data feed it
    LinkAddress(LinkAddress l)
    {
        x= l.x;
        y = l.y;
    }

}


/* Here we define the archetype of worm.
 A worm is composite: it has links.
 It has a chromosome that keeps
 the genetic information.
 The chromosome is the vector Mover[ ],
 which keeps the information
 on how to move its links:
 which  link and in which order
 (in the order that they appear in
 the vector Mover[]), and
```

```
 in which direction.
 Each worm has its own color,
 which is defined by three numbers.
 Each worm keeps record of
 the actual position of each
 one of its links,
 in linkAddress[]. */
private class Worm
{
    Mover mover[ ] ;
    int color[][];
    LinkAddress  linkAddress[ ];
    int side;

    private Worm( )
    {
        mover = new Mover[21];
        color = new int[21][4];
        linkAddress = new LinkAddress[21];
    }

    private Worm(Mover n[])
    {
        mover = new Mover[21];

        color = new int[21][4];
        linkAddress = new LinkAddress[21];

        mover  = n ;
    }

private Worm(Mover n[], int c[][], LinkAddress d[],int  f)
{
    mover = new Mover[21];// Information to move links

    color = new int[21][4]; // color  of each link
    linkAddress = new LinkAddress[21]; // Actual position

    mover  = n ; // acquisition of information
    color = c;
```

```
    linkAddress = d;
    side = f;
}
}


//Global definition and construction.
//Global definitions must be static


static Move  vectMoves[ ];
static Worm[] worm;
static Worm wormTwo[];


static int nWorms;   //Less than 20
static int  lifeSpan;
static int initialRow;
static int  order[];
static int numbPointMutations;
static int velocity;
static int nOriginalMovers, ncopies;
static int nLinks, nMovers ;
static int basalColor ;
static int nGen;
static boolean flag ;
static boolean  thereWasAdvance;

private  void constructors()
{
    System.out.println("constructors began");
    //Definition, assignment of memory.
    vectMoves = new Move[10];
    worm = new Worm[21];
}
```

```
static private void moveDef()
{
    // Here we define 9 cardinal points
    System.out.println("MoveDef began");
    final Move move1 =  new Move(0,1);
     vectMoves[1]   = move1 ;
    final Move move2 =  new Move(1,1);
     vectMoves[2]   = move2 ;
    final Move move3 =  new Move(1,0);
     vectMoves[3]   = move3 ;
    final Move move4 =  new Move(1,-1);
     vectMoves[4]   = move4 ;
    final Move move5 =  new Move(0,-1);
     vectMoves[5]   = move5 ;
    final Move move6 =  new Move(-1,-1);
     vectMoves[6]   = move6 ;
    final Move move7 =  new Move(-1,0);
     vectMoves[7]   = move7 ;
    final Move move8 =  new Move(-1,1);
     vectMoves[8]   = move8 ;
    final Move move9 =  new Move(0,0);
     vectMoves[9]   = move9 ;
    /* System.out.println(vectMoves[6].x);
     System.out.println(vectMoves[6].y);
     System.out.println(vectMoves[7].x);
     System.out.println(vectMoves[7].y); */
}


static private void initialChromosome(Worm worm1 )
{
    /*  The next human design says
        how the worm must move its links
        that it could  climb the screen:

        The instructions that choose what to do
        are defined and kept in the chromosome,
        which is  the vector mover[] .

        All links execute the same move:
```

```
        Move northeast (or northwest)
        Northeast: (1,1), whose number is 2,
        Northwest: (-1,1), whose number is 8.
        Which side is decided by the variable Side,
        0 means  northwest,
        1 is  northeast.
        Initial direction: Northeast=2.
        because Side = 0 */

    int n;
    for(int m = 1; m<=nLinks;m++)
    {
        n = nLinks +1 - m;
        //Move the tail first and then the others links.
        // Symbolic initialization
        Move  move1  =   new Move (1,1);
        Mover mover1  =   wormsClimb.new Mover(1,move1);
        // true initialization
        mover1.link = n;
        mover1.move = vectMoves[2 + 6 * worm1.side];
        worm1.mover[m]= mover1;
    }
}

static private void initialPosition(Graphics g, Worm worm[])
{
    int k, a,b;
    /*  System.out.println(vectMoves[6].x);
        System.out.println(vectMoves[6].y);
        System.out.println(vectMoves[7].x);
        System.out.println(vectMoves[7].y);*/

    for(int  W = 1; W<= nWorms; W++)
    {
        // We define an auxiliary worm
        Worm  worm1;
        worm1 = wormsClimb.new Worm();
        worm1.side = r.nextInt(2);

// All worms are posited at the same initial row
```

```
// in vertical position
for(int  l = 1; l <= nLinks; l++)
        {
            k = initialRow - nLinks;
            a = 20 + 20 * W;
            b = k + 12*l;
            LinkAddress   ad = wormsClimb.new LinkAddress (a,b);
            worm1.linkAddress [l] = ad;

            // Each link gets  colors
            int c1,c2,c3;
            c1 = r.nextInt(226);
            c2 = r.nextInt(226);
            c3 = r.nextInt(226);
            //colors are memorized.
            worm1.color[l][1]=c1;
            worm1.color[l][2]=c2;
            worm1.color[l][3]=c3;

            // The color is prepared
            g.setColor( new Color(c1,c2,c3));
            // A Rectangle is defined by
            // the coordinates of the
            // superior right corner,
            // width and height.
            //the link is painted
            g.fillRect(a,b,4,12);
            // The know-how to move
            initialChromosome(worm1);
        }
        //  Memorization
        worm[W]   = worm1;
    }
    /* System.out.println("asd = " +
worm[12].linkAddress[2].x );*/
}
```

```
private void delay(long l1,int s)
{
    // to attune the velocity of movement
    // that it could be observed by a human.
    while  (System.nanoTime() < l1 + s) {delay(l1,s);}
}


private void reproduction(Worm worm)
{
    //Our worms need  no reproduction
    //because they are  perfect.
}

private void actualChromosome( Worm worm)
{
    //Advance in zig-zag
    if ( worm.side == 0)  worm.side   = 1;
    else worm.side= 0;
    initialChromosome( worm  );
}

private void ribosome(Graphics g, Worm worm)
{
    // Reads and execute the genetic
    // information that resides in
    // the chromosome.
    int n, i, j;


    for (int m = 1; m<=nLinks ; m++)
    {
        System.out.println(m);
        //Reads the link to be moved
        n = worm.mover[m].link;

        //To move a link:
        //the cell occupied by  the link is cleared,
        //the cell to be occupied is painted.
```

```
        //The occupied cell is cleared
        i = worm.linkAddress[n].x;
        j = worm.linkAddress[n].y;

        g.setColor(Color.white);
        g.fillRect(i,j,4,12);

        // A new place is defined:
        //The move to be done is read

        Move  move1  =  worm.mover[m].move;

        i = i +4* move1.x;
        j = j - 6*move1.y;
        worm .linkAddress[n].x = i;
        worm .linkAddress[n].y = j;

        //Movement=repainting in the new place;

        g.setColor( new Color(
     worm.color[m][1],worm.color[m][2],worm.color[m][3] ));
        g.fillRect(i,j,4,12);
        //If movement is too fast, nothing can be seen.
        //To attune the velocity of the worm,
        //augment or diminish the delay as   pleased
        long  l1 = System.nanoTime();
        int s = 500; //change this.
        // change the upper bound of k
        for(int k=1; k<=2000000; k++)
        {
           delay(l1,s);
        }
        reproduction(worm);
    }//End of the m-loop
}



@Override
public  void  paint(Graphics g)
```

```
{
    nWorms = 8;
    nLinks = 7;
    initialRow = 85;
    g.drawString("Please, resize the Applet", 20,20);
    constructors();
    moveDef();
    initialPosition(g,worm );

    // Race
    for (int l = 1; l<=68; l++)
    {
        /* The index W distinguishes each worm.
        Each worm has its color
        Each worm has nLinks links,
        each part has its own coordinates on the screen,
        which are kept in the linkAddress   array.
        The worms are posited at the bottom of the screen.
        Head are at row InitialRow */

        for (int W = 1; W<=nWorms; W++)
        {
            Worm  worm1;
            worm1 = worm[W];
            actualChromosome(worm1);
            ribosome(g,worm1);
            reproduction(worm1);
            worm[W]=  worm1 ;
        }
    }
}
}//End of Program WormsClimb, B51
```

52 ***Exercise*** *Run the code and play with it making some reasonable changes in the parameters.*

53 ***Exercise*** *Digest the code. To that aim, study the comments. Next, make a copy of the code and save it in a word processor. Erase from that document any comment and then add your own comments that the program could be understandable. Please, notice that we are dealing with structuring code in an manner that could be easily understood by humans.*

54 ***Exercise*** *Our program has the following bug: the color of the links change when the worms begin to move. Make the necessary amendments that all worms could remember the color beginning with the very first movement.* Answer

4.3 Getting free

We already know how robotic all knowing worms behave. Our next task is to clean robotization by providing ways of being free. The price for freedom is ignorance.

55 ***Movement at random.*** *Let us see what random uncorrelated movements mean. To this aim, we fill the chromosome that contains the information on how to move links in random values. This is done in the method initialChromosome of the next program.*

```
// Program WormsRand, B55
// The links of each worm
// are programmed
// to  move at random.

package ejvol2p;



import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;
import java.util.Random;


public class WormsRand extends Applet
{
// Class WormsRand is an archetype.
// The next instruction says that
// one might
// construct a real stuff, an instance, on the
// image and resemblance of the archetype.

// The word archetype does not belongs
// in evolutionary biology,
```

```
// but it belongs in paleontology.
// We use here the word archetype as
// a synonym of a plan or design.
//
public WormsRand()// constructor
{}

// We command now to make a real instance
// in the image of the archetype, WormsRand.
static  WormsRand wormsRand  = new WormsRand();

// Turn on  the random generator

static Random r  = new Random();

private static final long serialVersionUID = 1L;

/*  We follow the next convention:
    Archetypes are written in upper case,
    instances are written in lower case

    In java, archetypes are called classes.
    It is not mandatory, but
    we define archetypes as  classes
    in within another one: as inner classes.

    Definition of the Move class:
    A Move says in which direction
    one must move:
    we have 8 cardinal points.
    Each point denoted by
    its x and y coordinates.

    An archetype may have
    two types of information:
    one about things
    The other about tasks */

static private  class Move
{
```

```
    int x; // this is a thing
    int y;

    // The next is a method, a function
    // that  declares
    // how information is acquired.
    // It is a constructor.
    // It is overload by two possibilities.
    // a) The input is a list
    Move(int a, int b)
     {
           x= a;
           y= b;
     }

    // b) The input is a structure
    Move(Move move)
     {
           x= move.x;
           y= move.y;
     }
}


// Definition of the class Mover:
// A Mover keeps two types of information:
// a) the link of the worm to be moved and
// b) the direction or move to be chosen.
private   class Mover
{

    int link ;
    Move   move;

    //  Movers can acquired information
    // in two ways, they are  overloaded.
    // Form 1: by a list

    private Mover(int a, int b, int c)
    {
```

```
        link= a;
        move.x = b;
        move.y = c;
    }

    // Form 2, by structured data
    private Mover(int a, Move m)
    {
        link= a;
        move = m;
        move.x = m.x;
        move.y = m.y;
    }
 }

// The class LinkAddress keeps the actual
// position of a link.
private   class LinkAddress
{
    int x;
    int y;

    // LinkAddress is overloaded:
    // A list feeds it
    LinkAddress(int a, int b)
    {
        x= a;
        y = b;
    }

    //Structured data feed it
    LinkAddress(LinkAddress l)
    {
        x= l.x;
        y = l.y;
    }

 }
```

```
/* Here we define the archetype of worm.
 A worm is composite: it has links.
 It has a chromosome that keeps
 the genetic information.
 The chromosome is the vector Mover[ ],
 which keeps the information
 on how to move its links:
 which  link and in which order
 (in the order that they appear in
 the vector Mover[]), and
 in which direction.
 Each worm has its own color,
 which is defined by three numbers.
 Each worm keeps record of
 the actual position of each
 one of its links,
 in linkAddress[]. */
private class Worm
 {

    Mover mover[ ] ;
    int color[][];
    LinkAddress  linkAddress[ ];
    int side;

    private Worm( )
    {
        mover = new Mover[21];
        color = new int[21][4];
        linkAddress = new LinkAddress[21];
    }

    private Worm(Mover n[])
    {
        mover = new Mover[21];
        color = new int[21][4];
        linkAddress = new LinkAddress[21];
        mover  = n ;
    }
```

```
    private Worm(Mover n[], int c[][], LinkAddress d[],int  f )
    {
        mover = new Mover[21]; // Information to move links

        color = new int[21][4]; // color  of each link
        linkAddress = new LinkAddress[21]; // Actual position

        mover  = n ; // acquisition of information
        color = c;
        linkAddress = d;
        side = f;
    }
 }


//Global definition and construction.
//Global definitions must be static


static Move  vectMoves[ ];
static Worm[] worm;
static Worm wormTwo[];


static int nWorms;   //Less than 20
static int  lifeSpan;
static int initialRow;
static int  order[];
static int numbPointMutations;
static int velocity;
static int nOriginalMovers, ncopies;
static int nLinks, nMovers ;
static int basalColor ;
static int nGen;
static boolean flag ;
static boolean  thereWasAdvance;

private  void constructors()
{
    System.out.println("constructors began");
```

```
    //Definition, assignment of memory.
    vectMoves = new Move[10];
    worm = new Worm[21];
}




static private void moveDef()
{
    // Here we define 9 cardinal points
    System.out.println("MoveDef began");
    final Move move1 =  new Move(0,1);
     vectMoves[1]   = move1 ;
    final Move move2 =  new Move(1,1);
     vectMoves[2]   = move2 ;
    final Move move3 =  new Move(1,0);
     vectMoves[3]   = move3 ;
    final Move move4 =  new Move(1,-1);
     vectMoves[4]   = move4 ;
    final Move move5 =  new Move(0,-1);
     vectMoves[5]   = move5 ;
    final Move move6 =  new Move(-1,-1);
     vectMoves[6]   = move6 ;
    final Move move7 =  new Move(-1,0);
     vectMoves[7]   = move7 ;
    final Move move8 =  new Move(-1,1);
     vectMoves[8]   = move8 ;
    final Move move9 =  new Move(0,0);
     vectMoves[9]   = move9 ;
    /* System.out.println(vectMoves[6].x);
     System.out.println(vectMoves[6].y);
     System.out.println(vectMoves[7].x);
     System.out.println(vectMoves[7].y); */
}



static private void initialChromosome(Worm worm1 )
```

```
{
    /*
        A link moves at random,
        without  purpose,
        without  restriction. */

    int n;
    for(int m = 1; m<=nLinks;m++)
    {
        //Select  which link to move:
        // Throw a number from 1 to 12

        n = r.nextInt(nLinks) +1;
        // Symbolic initialization
        Move  move1  =   new Move (1,1);
        Mover mover1  =   wormsRand.new Mover(1,move1);
        // true initialization
        mover1.link = n;

        //Select  which direction to move on
              //Throw a number from 1 to 9

        //Move the tail first and then the others links.

        mover1.move = vectMoves[r.nextInt(9) +1 ];

        worm1.mover[m]= mover1;
    }
}

static private void initialPosition(Graphics g, Worm worm[])
{
    int k, a,b;
    /* System.out.println(vectMoves[6].x);
     System.out.println(vectMoves[6].y);
     System.out.println(vectMoves[7].x);
     System.out.println(vectMoves[7].y);*/

    for(int  W = 1; W<= nWorms; W++)
    {
```

```
        // We define an auxiliary worm
        Worm  worm1;
        worm1 = wormsRand.new Worm();
        worm1.side = r.nextInt(2);

        // All worms are posited at the same initial row
        // in vertical position
        for(int  l = 1; l <= nLinks; l++)
        {
            k = initialRow - nLinks;
            a = 20 + 20 * W;
            b = k + 12*l;
            LinkAddress   ad = wormsRand.new LinkAddress (a,b);
            worm1.linkAddress [l] = ad;

            // Each link gets  colors
            int c1,c2,c3;
            c1 = r.nextInt(226);
            c2 = r.nextInt(226);
            c3 = r.nextInt(226);
            //colors are memorized.
            worm1.color[l][1]=c1;
            worm1.color[l][2]=c2;
            worm1.color[l][3]=c3;

            // The color is prepared
            g.setColor( new Color(c1,c2,c3));
            // A Rectangle is defined by
            // the coordinates of the
            // superior right corner,
            // width and hight.
            //the link is painted
            g.fillRect(a,b,4,12);
            // The know-how to move
            initialChromosome(worm1);
        }
        //  Memorization
        worm[W]   = worm1;
    }
    /* System.out.println("asd = " +
```

```
                        worm[12].linkAddress[2].x );*/
}



private void delay(long l1,int s)
{
    // to atune the velocity of movement
    // that it could be observed by a human.
    while  (System.nanoTime() < l1 + s) {delay(l1,s);}
}


 private void reproduction(Worm worm)
{
    //Our worms need  no reproduction
    //because they are  perfect.
}

private void actualChromosome( Worm worm)
{
    //Advance in zig-zag
   if ( worm.side == 0)  worm.side   = 1;
   else worm.side= 0;
   initialChromosome( worm  );
}

private void ribosome(Graphics g, Worm worm)
{
    // Reads and execute the genetic
    // information that resides in
    // the chromosome.
    int n, i, j;


    for (int m = 1; m<=nLinks ; m++)
    {
        System.out.println(m);
        //Reads the link to be moved
```

```
        n = worm.mover[m].link;

        //To move a link:
        //the cell occupied by  the link is cleared,
        //the cell to be occupied is painted.

        //The occupied cell is cleared
        i = worm.linkAddress[n].x;
        j = worm.linkAddress[n].y;

        g.setColor(Color.white);
        g.fillRect(i,j,4,12);

        // A new place is defined:
        //The move to be done is read

        Move  move1  = worm.mover[m].move;

        i = i +4* move1.x;
        j = j - 6*move1.y;
        worm .linkAddress[n].x = i;
        worm .linkAddress[n].y = j;

        //Movement=repainting in the new place;

        g.setColor( new Color(
        worm.color[n][1],worm.color[n][2],worm.color[n][3] ));
        g.fillRect(i,j,4,12);
        //If movement is too fast, nothing can be seen.
        //To attune the velocity of the worm,
        //augment or diminish the delay as   pleased
        long  l1 = System.nanoTime();
        int s = 500; //change this.
        // change the upper bound of k
        for(int k=1; k<=2000000; k++)
        {
           delay(l1,s);
        }
        reproduction(worm);
    }//End of the m-loop
```

```
  }


@Override
public  void  paint(Graphics g)
{
 nWorms = 8;
 nLinks = 7;
 initialRow = 85;
 setSize(500, 400) ;
 constructors();
 moveDef();
 initialPosition(g,worm );

// Race
 for (int l = 1; l<=68; l++)
 {
    /* The index W distinguishes each worm.
    Each worm has its color
    Each worm has nLinks links,
    each part has its own coordinates on the screen,
    which are kept in the linkAddress   array.
    The worms are posited at the bottom of the screen.
    Head are at row InitialRow */

    for (int W = 1; W<=nWorms; W++)
    {
     Worm  worm1;
     worm1  = worm[W];
     actualChromosome(worm1);
     ribosome(g,worm1);
     reproduction(worm1);
     worm[W]=  worm1 ;
    }
 }
}
}//End of Program WormsRand, B55
```

56 ***Exercise*** *Run the program and play with the code.*

57 ***Some tasks*** *Observing the output of the last code, we notice that links have no idea of what an organism is and moreover they can overwrite one another. To be fair to biology, all that must be remedied and will be done in the next program. Additionally, in our previous program, worms move sequentially, one after the other. This is clearly unnatural. So, we must simulate a movement, in which all worms move at the same time. Thus, we pass from sequential movement to parallel movement. At last, it is clear that larger worms move with more difficulty than shorter ones. So, we must make it possible to know whether or not to enlarge a worm that already knows how to move helps the new worm to find the way to move more rapidly.*

4.4 Enabling evolution

Randomness is the platform where evolution moves. We already know what random movements mean in the case of our worms. Now, we enable evolution: we accept and reinforce a random movement in the only case in which that movement is upwards.

58 ***Stepping into evolution***

We already know what a random movement means: it has no idea about the integrity of an organism and it ignores that two elements of matter cannot occupy the same place. We see also that random information does not recognize a preferred direction. In regard with the environment, it presents no feedback; there is no danger at all or any reward. The feedback of the environment is called **selection**.

Our job is to devise an evolutionary environment to teach these unwise links how to behave as an organism and how to advance toward a goal.

Thus, our task is to detect and forbid any disrupting proposal of movement. By the same token, we shall allow only those movements that do not overwrite links over links. With all those helps, evolution will easily find a solution to the posed problem: to teach worms to climb the display.

Those worms that by mere randomness happen to observe a tendency to rise along the display will have the right to reproduce and to get offspring in proportion to its efficacy. In that way, the knowledge of the population will increase from generation to generation. With our chosen selection, we program an easy rewarding procedure and at the end we get worms that know how to climb the screen.

To raise up the tail is the first task that is necessary to advance upwards: that is the beginning of an upward movement. It is clear that such a movement may be

done by mere randomness. So, our rewarding function must focus on the raising of the tail.

Let us introduce a flavor of the official jargon: the potentialities of random organization and movements are called **self-organization**. Thus, self-organization might cause a fluctuation that eventually could be used to trigger evolution.

What is then **evolution**?

It is the building of an evolutionary environment in which it is possible to link the potentialities of self organization to the reproductive cycle in such a way that those individuals that have certain property with more emphasis are allowed to be reproduced more abundantly than the rest. In that way, the selected property is extended over the population and is perfected to the extreme thanks to mutation and recombination.

Erratum: recombination was not included in the original version, in which mutation was all to evolution. Not, it is.

The next piece of code contains the central idea (this code is to be understood not to be run):

```
 public void dynamics(Graphics g)
 {

 // Worms are posited at the starting position,
 // their chromosomes are filled
 // in random information
 initialization();

 //Evolutive cycle
  for (int gen = 1; gen <=  nGen; gen++)
{
  System.out.println("Gen = " + gen);
  //The information in the
  //chromosomes is executed
  exoGenetics(g);

  //results are assessed
    assessFitness();
    // Reproduction is made,
    //mutations are enabled:
    //Gene amplification is included,
```

```
    //body enlargement is allowed.
    reproduction(g);
    reInitialization(g,worm);
   }
 }
```

59 ***Exercise*** *Verify that the program simulates evolution, specifying each one of its features.*

60 ***Selection***

Selection is defined by the method that declares which individuals leave more offspring. This purpose generates two main tasks: 1) to determine which worms have begun to rise up the tail, they are the fitter, and 2) to implement differential reproduction: more offspring for the fittest. We need moreover to include a mechanism of mutation to guarantee that the property of climbing could arise by mere randomness in the population.

61 ***Our mutations***

Molecular biology has unveiled several types of mutations, but we have implemented in our program only two: mutation at a base and gene amplification. Additionally, we will will include recombination in the next chapter. In nature, mutation at a base site is the most elementary mutation and one that cannot be stopped because it is enabled by quantum mechanics. So, simulation of mutation at a base is an inescapable duty. Next, we chose gene amplification because the form as a human solves the posed problem encloses the repetition in different places of the same instruction. Recombination is also implemented because it is a candidate to accelerate evolution, which in general is very demanding in every kind of resources, including time.

Caution: our scheme is called directed evolution or artificial selection. This has nothing to do with natural selection: here we pick up the best individual according to a predefined criterion. But nature is deprived of criteria.

62 ***The complete code*** *Let us see the code that enables the worms to acquire by an evolutionary process the know-how to climb the screen. When a worm appears that knows how to climb the screen, a new population of worms with one more link replaces the old one.*

```
// Program WormsE, B62
// An evolutionary environment
// is designed, in which
// a population of worms
// is programmed by evolution
// to climb the screen.
// Only mutation is implemented.
// Evolution functions as
// a software designer.

package ejvol2p;


//Needed tools
import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;
import java.util.Random;


public class WormsE extends Applet
{


private static final long serialVersionUID = 1L;

public WormsE()
{

}

// Class WormsE is an archetype.
// The next instruction says that
// one might
// construct a real stuff, an instance, on the
// image and resemblance of the archetype.

// Archetype means old-type, prototype.
// The word archetype does not belong
// in evolutionary biology,
```

```
// but it belongs in paleontology.
// We use here the word archetype as
// a synonym of a plan or design,
// of an idea.

static  WormsE wormsE  = new WormsE();


// Turn on  the random generator

Random r  = new Random();

// private static final long serialVersionUID = 1L;

/* We follow the next convention:
 Archetypes are written in upper case,
 instances are written in lower case.

 In java, archetypes are called classes.
 It is not mandatory, but
 we define archetypes as  classes
 in within another one: as inner classes.

 Definition of the Move class:
 A Move says in which direction
 one must move:
 we have 8 cardinal points.
 Each point denoted by
 its x and y coordinates.

 An archetype may have
 two types of information:
 one about things
 The other about tasks */

private  class Move
{
    int x; // this is a thing
  int y;
```

```
 // The next is a method, a function
 // that  declares
 // how information is acquired.
 // It is a constructor.
 // It is overload by two possibilities.
 // a) The input is a list
 Move(int a, int b)
{
    x= a;
    y= b;
}

// b) The input is a structure
Move(Move move)
{
    x= move.x;
    y= move.y;
}

}


// Definition of the class Mover:
// A Mover keeps two types of information:
// a) the link of the worm to be moved and
// b) the direction or move to be chosen.
private   class Mover
{

    int link ;
    Move   move;

    //  Movers can acquired information
    // in two ways, they are  overloaded.
    // Form 1: by a list
    private Mover(int a, int b, int c)
    {
           link= a;
     move.x = b;
     move.y = c;
```

```
    }

    // Form 2, by structured data
    private Mover(int a, Move m)
    {
           link= a;
           move = m;
     move.x = m.x;
     move.y = m.y;
    }
}


// The class LinkAddress keeps the actual
// position of a link.
private   class LinkAddress
{
    int x;
    int y;

    // LinkAddress is overloaded:
    // A list feeds it
    LinkAddress(int a, int b)
    {
        x= a;
        y = b;
    }

    //Structured data feed it
    LinkAddress(LinkAddress l)
    {
         x= l.x;
         y = l.y;
    }
}


/* Here we define the archetype of worm.
 A worm is composite: it has links.
 It has a chromosome that keeps
```

```
 the genetic information.
 The chromosome is the vector Mover[ ],
 which keeps the information
 on how to move its links:
 which  link and in which order
 (in the order that they appear in
 the vector Mover[]), and
 in which direction.
 Each worm has its own color,
 which is defined by three numbers.
 Each worm keeps record of
 the actual position of each
 one of its links,
 in linkAddress[]. */
private class Worm
{

    Mover mover[ ] ;
    int color[][];
    LinkAddress  linkAddress[ ];


    private Worm( )
    {
        mover = new Mover[41];
        color = new int[21][4];
        linkAddress = new LinkAddress[21];
    }

    private Worm(Mover n[])
    {
        mover = new Mover[21];
        color = new int[21][4];
        linkAddress = new LinkAddress[21];
        mover  = n ;
    }

    private Worm(Mover n[], int c[][], LinkAddress d[])
    {
        mover = new Mover[21]; // Information to move links
```

```
        color = new int[21][4]; // color  of each link
        linkAddress = new LinkAddress[21]; // Actual position

        for(int a = 1; a<= 20; a++)
            mover[a]  = n[a] ; // acquisition of information

        for(int a = 1; a<= 20; a++)
           for(int b = 1; b<= 3; b++)
                color[a][b]  = c[a][b] ;

        for(int a = 1; a<= 20; a++)
             linkAddress[a]   =d[a]  ;
    }

    private Worm(Worm worm1)
    {
        // Information to move links
        mover = new Mover[21];
        // color  of each link
        color = new int[21][4];
        // Actual position
        linkAddress = new LinkAddress[21];

        for(int a = 1; a<= 20; a++)
            // acquisition of information
            mover[a]  = worm1.mover[a] ;

        for(int a = 1; a<= 20; a++)
            for(int b = 1; b<= 3; b++)
              color[a][b]  = worm1.color[a][b] ;

        for(int a = 1; a<= 20; a++)
              linkAddress[a]   =worm1.linkAddress[a]  ;
    }
}


//Global definition.
//Global definitions must be static
```

```
 static Move  vectMoves[ ];
 static Worm[] worm;
 static Worm[] wormTwo;


 static int nWorms;   //Less than 20
 static int width;
 static int  lifeSpan;
 static int initialRow;
 static int  order[];
 static int numbPointMutations;
 static int velocity;
 static int nOriginalMovers, nCopies;
 static int nLinks, nMovers, factor ;
 static int basalColor ;
 static int nGen;

 static int  thereWasAdvance;
 static int randInitio;
 static int xWin, yWin;

 static boolean occupied[][];
 static  int fitness[];

//Automatic ignition
@Override
public void init()
{
    //repaint();
    //Measures of the Applet
    xWin = 500;
    yWin = 400;
    setSize(xWin,yWin);
}

//Definition, assignment of memory.
private  void constructors( )
{
```

```
    // System.out.println("constructors   ");
    vectMoves = new Move[10];
    worm = new Worm[21];
    wormTwo= new Worm[21];
    order = new int[21];
    occupied = new boolean[201][101];
    fitness = new int[21];
}



// To avoid overwriting of worms,
// an occupation record is kept:
// The screen is divided in cells 4x4.
// At the beginning every cell is empty=false.
// Occupancy is recorded in occupied[][],
// which is a boolean matrix.
// A boolean  variable can be
// true= occupied,  else false= not occupied.
private void  occupationMatrix()
{
    //System.out.println("occupationMatrix");
    for(int i = 1; i<=200; i++)
    {
        for(int j = 1; j<=100; j++)
            occupied[i][j]= false;
    }
}


 //Here we define 9 cardinal points
 private void moveDef( )
{
    // System.out.println("MoveDef   ");
    final Move move1 =  new Move(0,1);
     vectMoves[1]   = move1 ;
    final Move move2 =  new Move(1,1);
     vectMoves[2]   = move2 ;
    final Move move3 =  new Move(1,0);
     vectMoves[3]   = move3 ;
```

```
    final Move move4 =  new Move(1,-1);
     vectMoves[4]   = move4 ;
    final Move move5 =  new Move(0,-1);
     vectMoves[5]   = move5 ;
    final Move move6 =  new Move(-1,-1);
     vectMoves[6]   = move6 ;
    final Move move7 =  new Move(-1,0);
     vectMoves[7]   = move7 ;
    final Move move8 =  new Move(-1,1);
     vectMoves[8]   = move8 ;
    final Move move9 =  new Move(0,0);
     vectMoves[9]   = move9 ;
    /* System.out.println(vectMoves[6].x);
     System.out.println(vectMoves[6].y);
     System.out.println(vectMoves[7].x);
     System.out.println(vectMoves[7].y); */
}


//Initialization of parameters
private void setParameters()
{
    /*  Each worm has its color
 Each worm has nlinks links,
 each link has its own coordinates on the screen,
 which are kept in the linkAddress.
 The worms are posited at the bottom of the screen.
 Their head are at row InitialRow  */
 //InitialRow is the place in the screen
   //where the race begins
   initialRow = 350;

    // Nworms is the number of worms,
    // Maximum allowed value 20.
    nWorms = 8;
    //Width of the worm
    width = 4;
    // NGen is the number of generations
    nGen = 2000000000;
```

```
// Each worm has nlinks number of links.
//nlinks shall be smaller than 20.
//Initial number of links
nLinks = 2;

//Each chromosome has NMovers
//number of instructions.
//NMovers shall be smaller than 40
factor = 2;
nMovers = factor * nLinks;

// LifeSpan is the number of   days
// in a life.
// Life ends with reproduction and dead.
lifeSpan = 4 * nMovers;

//NumbPointMutations  is the number of
//point mutations of the offspring
numbPointMutations = 2;

//Gene amplification follows:
//NOriginalMovers is the number of different Movers
//to be copied.
nOriginalMovers = 2;

//Ncopies is the number of new copies
//of each copied Mover.
nCopies = 2;

//Velocity is among 0 and 10000,
// it regulates  the velocity of the
//movements in the display.
velocity = 500;

/*To generate a repeatable sequence
 of random numbers, activate the next
 two lines:

long seedValue = 40;
r.setSeed( seedValue) ;*/
```

```
}

/*
Movement at random,
without any purpose,
without any restriction. */
private   void initialChromosome(Worm worm1 )
{
    int n;
    for(int m = 1; m<=nMovers;m++)
    {
        //Select  which link to move:
        // Throw a number from 1 to nLinks

        n = r.nextInt(nLinks) +1;
        // Symbolic initialization
        Move  move1  =   new Move (1,1);
        Mover mover1  =   wormsE.new Mover(1,move1);
        // true initialization
        mover1.link = n;

        //Select  which direction to move on
        //Throw a number from 1 to 9

        //Move the tail first and then the others links.

        mover1.move = vectMoves[r.nextInt(9) +1 ];

        worm1.mover[m]= mover1;
    }
}

// All worms are posited at the same initial row
// in vertical position
private void initialPosition(Graphics g, Worm worm[])
{
    int k, a,b;

    for(int  W = 1; W<= nWorms; W++)
    {
```

```
        // We define an auxiliary worm
        Worm  worm1;
        worm1 = wormsE.new Worm();


        for(int  l = 1; l <= nLinks; l++)
        {
            //Here is the head
            k = initialRow - width*nLinks;
            // The distance among worms is 40.
            // width is the width of  a link
            a =   40 * W;
                b = k + width*l;
            LinkAddress   ad = wormsE.new LinkAddress (a,b);
            worm1.linkAddress [l] = ad;

            // The first generation is initialized
            // by randomness, others by inheritance.
            // symbolic initialization
            int c1=1,c2=1,c3=1;
            if (randInitio == 1)
            {
                // a color is assigned to each link
                // but different than white
                c1 = r.nextInt(256);
                c2 = r.nextInt(256);
                c3 = r.nextInt(256);
                if (c1+c2+c3==765) c3=1;
                //colors are memorized.
                worm1.color[l][1]=c1;
                worm1.color[l][2]=c2;
                worm1.color[l][3]=c3;
            }

            // The color is prepared

            g.setColor( new Color(c1,c2,c3));
            // A Rectangle is defined by
    // the coordinates of the
    // superior right corner,
```

```
    // width and hight.
            // The link is painted
            g.fillRect(a,b,width, width);

            int m =  a/width;
            int n =   b/width;
            // System.out.println("occupied = " + m + "  " + n );
            occupied[m][n]= true;
        }
        //The know-how to move.
        if (randInitio == 1) initialChromosome(worm1);
        //Memorization
worm[W] = worm1;
    }
    /* System.out.println("asd = " +
worm[12].linkAddress[2].x );*/
}


// Parameters are set and worms are posited at the start
private void setBeginning(Graphics g)
{
    setParameters();

    // Initialization at random
    randInitio = 1;
    initialPosition( g,  worm);
    randInitio = 0;
}



//Beginning from the bottom
//with new genes
private void reInitialization(Graphics g, Worm worm[])
{
    randInitio = 1;
    occupationMatrix();
    initialPosition( g, worm);
}
```

```
// A worm is and must remain a unity.
private int disruptive(int m, Worm worm )
{
    //  System.out.println("disruptive");
    // A move is disruptive if the target
    // address is farther than 1.5 units
    //  from the neighboring links.
    int  n;
    int  x,y, x1, y1, x2, x3, y2, y3;
    double d1;
    double d2 = 0;

    //Reads the link to be moved
    n = worm.mover[m].link;
    //Reads the   address of the link;
    x  = worm .linkAddress [n].x;
    y  = worm .linkAddress [n].y;
    //The   move to be done is read
    Move move1  =  worm .mover[m].move;

    //The target address is calculated
    x1 =   x + width*move1.x;
    y1 = y - width*move1.y;

    // The addresses of the
    // two neighboring links is read
    // The disruption is calculated

    if ((1 < n) & (n < nLinks))
    {
        x2 = worm .linkAddress [n-1].x;
        y2 = worm .linkAddress [n-1].y;
        d1 =   Math.pow(  (Math.pow( (x2 - x1),2)
                    + Math.pow((y2 - y1),2)),0.5);
        x3 = worm .linkAddress [n + 1].x;
        y3 = worm .linkAddress [n + 1].y;
```

```
        d2 =  Math.pow(  (Math.pow( (x3 - x1),2)
                  + Math.pow((y3 - y1),2)),0.5);


        // Here we decide whether or not
        // a move is disruptive

        if ((d1 > 1.4*width) || (d2 > 1.4*width))
           return 1;
        else
           return 0;
    }


    if (n == 1)
    {
        x2 = worm .linkAddress [2].x;
        y2 = worm .linkAddress [2].y;
        d1 = Math.pow(  (Math.pow( (x2 - x1),2)
                  + Math.pow((y2 - y1),2)),0.5);
    if( d1 > 1.1*width) return 1; else return   0;
    }

    if( n == nLinks )
    {
        x2 = worm .linkAddress [nLinks - 1].x;
        y2 = worm .linkAddress [nLinks - 1].y;
        d2 = Math.pow(  (Math.pow( (x2 - x1),2)
                  + Math.pow((y2 - y1),2)),0.5);
    }
    if ( d2 > 1.1*width ) return    1;
        else return 0;
}


// The velocity of movement
// shall be observable  by a human.
private void delay(long l1 )
{
```

```
    while  (System.nanoTime() < l1  ) {delay(l1 );}
}



//Worms begin to move
private void moveOn(Graphics g, int m, Worm worm )
{
    int n, i,j;

    //  System.out.println("moveOn");

    //Read  the link to be moved
    n = worm.mover[m].link;

    //To move a link:
    //the cell occupied by  the link is cleared,
    //the cell to be occupied is painted.

    //The occupied cell is cleared
    i = worm.linkAddress[n].x;
    j = worm.linkAddress[n].y;
    int k =  i/width;
    int l =   j/width;

    /*System.out.println("TTT" + occupied[k][l]);
    System.out.println("occupied = " + k + "  " + l );*/
    occupied[k][l] = false;
    // System.out.println(i+j+n);
    // the white color is prepared
    g.setColor(Color.white);
    g.fillRect(i,j,width, width);


    //The move to be done is read

    Move  move1  =  worm.mover[m].move;

    i = i +width* move1.x;
    j = j - width*move1.y;
```

```
    worm.linkAddress[n].x = i;
    worm.linkAddress[n].y = j;

    //Movement=repainting in the new place;

    g.setColor( new Color(
              worm.color[n][1],worm.color[n][2],worm.color[n][3] ));
    g.fillRect(i,j,width, width);
    //Cell is occupied
    k = k+move1.x;
    l = l-move1.y;
    occupied[k][l] = true;

    //If movement is too fast, nothing can be seen.
    //To attune the velocity of the worm,
    //augment or diminish the delay as   pleased
    //change the upper bound of h

    long  l1 = System.nanoTime();

    //change the upper bound of k
    for(int h=1; h<=(10000-velocity) * 20  ; h++)
    {
       delay(l1 );
    }

}


// The instructions that are in the Chromosome,
// are executed.
private void ribosome(Graphics g, int m, Worm worm)
{
    // System.out.println("ribosome");
    int  n, i, j;

    Move move ;
    int x, y, x1, y1;

    //Reads the link  to be moved
```

```
    // System.out.println("m = " + m);
    //The   move to be done is read
    move = worm .mover[m].move;
    //The   link to be moved is read
    n = worm.mover[m].link;
    //Reads the   address of the link;
    x = worm .linkAddress [n].x;
    y = worm .linkAddress [n].y;


    //The target address is calculated
    x1 = x + width*move.x;
    y1 = y - width*move.y;

    //Movement must remain in within barriers
    if ((x1 >= 1) & (y1 >= 1) & (x1 < 500) & (y1 < 400))
    {

        //Is the target address occupied?
        // Is the move disruptive?

        i =   x1 / width;
        j =   y1 / width;

        //   System.out.print("i= " + i + "  j= " + j);
        //   System.out.println("occupied = " + occupied[i][j] );

        if ((occupied[i][j]== false ) &
                (disruptive(m,worm) == 0))
                   moveOn( g, m,   worm );
    }

}



// Every worm is assigned a fitness
private void assessFitness()
{
    //  System.out.println("Fitness");
```

```
    int champ;
    int max, man ;


    // The nearer to the top, the fitter

    for (int W = 1; W<= nWorms; W++)
    {
        // We catch the farthest link
        max = 1 ;
        for (int m = 1; m<= nLinks; m++)
        {
            man= worm[W].linkAddress[m].y;
            if ( man > max)   max = man;
        }
        fitness[W] = 1000 - max;
        //  System.out.println( W + " " + fitness[W] );
    }

    //   Now, we sort worms by fitness
    //   and select the best four,
    //   whose indexes are recorded in Order[].
    for (int W = 1; W<=4; W++)
    {
        champ = 1;
        for (int B = 1; B <= nWorms; B++)
            if (fitness[B] > fitness[champ])
                champ = B;
        order[W] = champ;
        fitness[champ] = 0;
    }
}


//Blind point mutations
private void pointMutation()
{
    int   m, n;
    //Let us try out blind point mutations
    //in specific Movers,
```

```
    //in the  link  to be moved
    //and in the direction of movement.
    for (int W = 1; W<= nWorms; W++)
    {
        //Each offsprings receives numbmutation mutations
        for (int k = 1; k<= numbPointMutations; k++)
        {
            //Select at random  which Mover to mutate:
            //throw a number from 1 to NMovers
            m =  r.nextInt(nMovers ) + 1;
            //Select at random which link  to move:
            //throw a number from 1 to Nlinks.
            n = r.nextInt( nLinks) + 1;
            worm[W].mover[m].link  = n;
            //Select at random which direction to move on.
            //Throw a number from 1 to 9
            worm[W].mover[m].move = vectMoves[r.nextInt(9) + 1];
        }
    }
}


// Some movers are overwritten with others
// in within the same worm
private void geneAmplification()
{
    //  System.out.println("geneAmplification");
    int    e, n,   ee;

    Worm  worm1;


    for (int W = 1; W<= nWorms; W++)
    {
        worm1 = wormTwo[W];
        initialChromosome(worm1);
        for (int m = 1; m<= nOriginalMovers; m++)
        {
            //Select  which Mover to be amplified:
            //throw a number from 1 to NMovers
```

```
            e = r.nextInt(nMovers)+1  ;

            Move move1  = worm1.mover[e].move;

            wormTwo[W].mover[e].move = move1;
            n = wormTwo[W].mover[e].link;

            move1 = worm[W].mover[e].move;
            //The Mover is amplified
            for (int k = 1; k<= nCopies; k++)
            {
                //Select at random where to be copied:
                //throw a number from 1 to NMovers.
                ee = r.nextInt(nMovers) + 1;
                worm[W].mover[ee].link = n;
                worm[W].mover[ee].move = move1;
            }
        }
    }
}

// Old worms are erased.
private void death(Graphics g  )
{
    // The screen is cleared
    g.setColor(Color.white);
    g.fillRect(0,0, xWin, yWin);
    //The worm array is overwritten with
    //the wormTwo array.
    worm  = wormTwo ;
}


// Point mutation and gene amplification
private void mutation(   )
{
    //Point mutation dominates over
    //gene amplification.
    geneAmplification( );
    pointMutation();
```

```
}


//A new link  is added to all worms.
private void oneMoreLink(  )
{
    // System.out.print("oneMoreLink: ");
    int   lim1;
    int   n;
    // The link  is added at the tail
    lim1 = nMovers + 1;
    nLinks = nLinks + 1;
    // System.out.println(nLinks);


    nMovers = factor * nLinks;
    Worm  worm1;
    for (int W = 1; W<= nWorms; W++)
    {
        worm1 = worm[W] ;
        //New movers are filled in information
        for (int m = lim1; m<=nMovers; m++)
        {

            //Select at random which link to move:
                  // Throw a number from 1 to nLinks

            n = r.nextInt(nLinks) +1;
            // Symbolic initialization
            Move  move1  =   new Move (1,1);
                  Mover mover1  =   wormsE.new Mover(1,move1);
                  // true initialization
                  mover1.link = n;

            //Select  which direction to move on
                  //Throw a number from 1 to 9

            //Move the tail first and then the others links.

            mover1.move = vectMoves[r.nextInt(9) +1 ];
```

```
                worm1.mover[m]= mover1;


            worm[W].mover[m] = worm1.mover[m];
        }
    }
}

// Is there any progress?
private   int thereWasAdvance()
{
    // System.out.println("thereWasAdvance");
    // Our criterion for   advance is
    // that the head has surpassed
    // its original vertical position.
    int headPosition, advance ;
    advance = 0;

    for (int W = 1; W<=nWorms; W++)
    {
        Worm  worm1 ;
        worm1  = worm[W];
        headPosition = worm1.linkAddress[1].y;
        if (headPosition < initialRow - width*nLinks)
            advance = 1;
    }
    /* if (advance == 1) System.out.println("Advance");*/
    return advance;
}


// If at least one worm   advanced even a bit,
// the next population is enlarged by one link:
private void growth()
{
    //  System.out.println("growth");
    if (thereWasAdvance() ==1)   oneMoreLink();
 else;
}
```

```
//* Winners reproduce others not
private void reproduction(Graphics g)
{
//   Our reproductive scheme is as follows:
//   50% offspring for  the champion,
//   25% offspring for  the second,
//   12.5% for   the third,
//   12.5% for   the fourth.
//   We place children on order.
//   Children are clones of the fathers
//   that are copied to the wormTwo array.

//   Call InitialPosition(8)
    int champ = order[1];
    Worm  worm1;
        worm1 = wormsE.new Worm(worm[champ].mover,
    worm[champ].color,worm[champ].linkAddress );
    /*  System.out.println("Address1 = "
                              + worm1.linkAddress[2].x); */

    int half =   nWorms/2;
    int quarter =   nWorms/4;
    // System.out.println("Champ = " + order[1] );


    for (int W = 1; W<= half; W++)
        wormTwo[W]=worm1;

    int limf = half + quarter;
    // System.out.println("limf = " + limf);

    champ = order[2];
    worm1 = wormsE.new Worm(worm[champ].mover,
    worm[champ].color,worm[champ].linkAddress );

    for (int W =half + 1; W<= limf; W++)
        wormTwo[W]=worm1;

    int limi = limf;
```

```
    limf = limf + quarter / 2;

    champ = order[3];
    worm1 = wormsE.new Worm(worm[champ].mover,
                    worm[champ].color,worm[champ].linkAddress );
    for (int W =  limi +1; W<=limf ; W++)
        wormTwo[W]=worm1;

    champ = order[4];
    worm1 = wormsE.new Worm(worm[champ].mover,
                    worm[champ].color,worm[champ].linkAddress );
    for (int W = limf+1; W<= nWorms; W++)
    {
        wormTwo[W]=worm1;
    }
    // System.out.println("limf = " + limf);
    /*   System.out.println("Address1= " +
    worm[order[1]].linkAddress[2].x ); */
    /*  System.out.println("Address2 = " +
  wormTwo[order[1]].linkAddress[2].x);*/
    death(g );
    mutation();
    growth();
}


//
private void exoGenetics(Graphics g)
{
    // Number of generations to be run

    // This is a life
    // l is the counter of days of life.
    // m is the number of the mover;
    // W is the index of the worm.

    for (int l = 1; l<=  lifeSpan ; l++)
    {
        for (int m  = 1 ; m <= nMovers; m ++)
        {
```

```
            for  (int W = 1; W<=nWorms; W++)
            {
                //Auxiliary worm: worm1
                Worm  worm1;
                worm1  = worm[W];
                /* System.out.println("W = " +W + " m = "
                + m + " l= " + l);  */
                ribosome(g,m,worm1);
                worm[W]=  worm1 ;
                }
            }
        }
    }


// Evolution is called to function
private void dynamics(Graphics g)
{
    System.out.println("Initial length = " + nLinks );
    System.out.println("Link number + waiting time");
    int time1 = 0;
    int n1 = nLinks;
    int waitingTime ;
    for (int gen = 1; gen <=  nGen; gen++)
    {
        if (nLinks>n1)
        {
            n1 = nLinks;
            waitingTime = gen - time1;
            time1 = gen;
            System.out.println( nLinks + "\t " +
            waitingTime);

        }
        g.drawString("Gen = " + gen, 20,20);
        //System.out.println("Gen = " + gen);
        exoGenetics(g);
        assessFitness();
        //Gene amplification is included.
        //Body enlargement is allowed.
```

```
        reproduction(g);
        reInitialization(g,worm);
    } //end of gen loop
}


//This is the main method.
@Override
public  void  paint(Graphics g)
{
    //System.out.println("Applet");
    constructors( );
    occupationMatrix();
    moveDef(  );
    setBeginning(g);
    dynamics(g);
}
}//End of Program WormsE, B62
```

63 ***Exercise*** *Run the program as a Java Applet. Please, regulate the velocity of the movement with the parameter "velocity", which can take values on the interval (0,10000). Notice that once a worm succeeds in taking an upwards direction, a new link is added to the worms of the next generation. The velocity must be regulated that these movements could be observed.*

64 ***Exercise*** *The program outputs the link index and the waiting time. The waiting time is the number of generations that were necessary for the appearing of a new link (whose index is reported) since the appearing of the previous one. Using the output of the console, make a graphic of the waiting time against the link index. With a suitable transformation, decide whether or not the graphic represents an exponential function.*

65 ***Exercise*** *Search for the method "setParameters" and play with the parameters. For instance, begin with worms with three links (NLinks= 3) and during some 20 generations. Verify that successful mutants arise after some 13 generations. For worms with 4 links, use the following setting: Nlinks=4; LifeSpan = 8; NGen = 500. For another run, augment the number of exons from 8 to 12. Next, augment the number of point mutations from 1 to 2 (per chromosome, per generation). Make some scans with worms of 5 , 6 or 7 links to see what happens.*

66 ***Exercise*** *Study the code and the comments that clarify it. Next, make a copy of the program in a word processor and delete all comments. Next, fill the program*

in your own comments. Return back to the original commented program for new lights when problems are met.

67 ***Exercise*** *Attempt to understand the code: get aware of details as well as of the architecture. Help yourself with the breakpoint + debug facility of NetBeans and with the activation of tracer printing instructions.*

4.5 Conclusion

A restricted form of evolution was used to synthesize the software that enables a virtual worm to climb the screen. A very simple programming language, consisting of 9 moves, was sufficient and the resultant software was packed into a string that is called chromosome. That information was decoded and converted in movement by another method that play the role of the ribosome in the cell and of a CPU in a computer. In short, the chromosome contains the software while the ribosome is the central processing unit. So, **evolution has been promoted to a software developer**.

Chapter 5

Evolution with crossing over

`The power of cooperation.`

68 ***Motivation and purpose*** *We saw in the last chapter how evolution designed software that enabled a population of worms to climb the screen. But, each worm defined a pure evolutionary line that had nothing to do with the others. So, discoveries made by different lines were not joined together to try to accelerate evolution, which is very slow. In this chapter, we implement* ***crossing over*** *or* ***recombination*** *in the population and investigate its effect on the velocity of evolution. In that way, we can discuss somehow the terrible question of why sex has not been banned by evolution.*

5.1 Recombinant operators

We saw in Vol I at least one rudimentary form of implementing recombination: one takes two strings and makes a new one taking a part from the first string and another part from the second. Now, we are going to define recombination in a more general form.

69 ***Definition***

Let us consider two strings, $s1$ and $s2$. They could be DNA strings of the form $TTAACGTATGCA$ but one can also imagine more complex strings, such as the chromosomes of our worms of the previous chapter, for which we will later implement recombination. The two strings are assumed of the same length, say, n. A **recombinant operator** R is encoded by a binary string of the same length and whose chars can be zero or one. When R operates over the ordered pair $(s1, s2)$, it produces another string $s3$ as follows: $s3$ has in the $i-th$ place the corresponding

letter from $s1$ if R has a one in that place, otherwise $s3$ takes for that place the corresponding letter from $s2$.

70 ***Example*** *Recombinant operator R recombines s_1 and s_2:*
$s1 = abcdefg$
$s2 = hijklmn$
$R = 0110101$
$s3 = hbckemg = R(s_1, s_2)$

71 ***Implementation***

A recombinant operator is generated at random, which acts over two worms that are chosen at random from the population. The resultant recombinant individual enters to form the raw material of the next generation. Recombination is executed as many times as there are individuals in the population. In that way the censal population remains constant. The code follows:

```
// Program WormsRec, B71
// A population of worms
// is programmed by evolution
// to climb the screen.
// Recombination is implemented.

// Automatic format: Source -> Format

//Bug:
//This programs sometimes draws the worms others not.
//Unknown reason.

package ejvol2p;

// Needed tools
import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;
import java.util.Random;
import java.util.logging.Level;
import java.util.logging.Logger;

public class WormsRec extends Applet {
```

```
    private static final long serialVersionUID = 1L;

    public WormsRec() {

    }

    // Class WormsRec is an archetype.
    // The next instruction says that
    // one might
    // construct a real stuff, an instance, on the
    // image and resemblance of the archetype.
    // Archetype means old-type, prototype.
    // The word archetype does not belong
    // in evolutionary biology,
    // but it belongs in paleontology.
    // We use here the word archetype as
    // a synonym of a plan or design,
    // of an idea.
    static WormsRec wormsRec = new WormsRec();

    // Turn on  the random generator
    Random r = new Random();

    // private static final long serialVersionUID = 1L;

    /* We follow the next convention:
        Archetypes are written in upper case,
        instances are written in lower case

        In java, archetypes are called classes.
It is not mandatory, but
        we define archetypes as  classes
        in within another one: as inner classes.

        Definition of the Move class:
        A Move says in which direction
one must move:
we have 8 cardinal points.
        Each point denoted by
```

```
its x and y coordinates.

        An archetype may have
        two types of information:
        one about things
        The other about tasks */
    private class Move {

        int x;
        int y;

        // The next is a method, a function
        // that  declares
        // how information is acquired.
        // It is a constructor.
        // It is overload by two possibilities.
        // a) The input is a list
        Move(int a, int b) {
            x = a;
            y = b;
        }

        // b) The input is a structure
        Move(Move move) {
            x = move.x;
            y = move.y;
        }

    }

    // Definition of the class Mover:
    // A Mover keeps two types of information:
    // a) the link of the worm to be moved and
    // b) the direction or move to be chosen.
    private class Mover {

        int link;
        Move move;

        //  Movers can acquired information
```

```
        // in two ways, they are  overloaded.
        // Form 1: by a list
        private Mover(int a, int b, int c) {
            link = a;
            move.x = b;
            move.y = c;
        }

        // Form 2, by structured data
        private Mover(int a, Move m) {
            link = a;
            move = m;
            move.x = m.x;
            move.y = m.y;
        }
    }

    // The class LinkAddress keeps the actual
    // position of a link.
    private class LinkAddress {

        int x;
        int y;

        // LinkAddress is overloaded:
        // A list feeds it
        LinkAddress(int a, int b) {
            x = a;
            y = b;
        }

        //Structured data feed it
        LinkAddress(LinkAddress l) {
            x = l.x;
            y = l.y;
        }

    }

    /* Here we define the archetype of worm.
```

```
A worm is composite: it has links.
It has a chromosome that keeps
the genetic information.
The chromosome is the vector Mover[ ],
which keeps the information
on how to move its links:
which  link and in which order
(in the order that they appear in
the vector Mover[]), and
in which direction.
Each worm has its own color,
which is defined by three numbers.
Each worm keeps record of
the actual position of each
one of its links,
in linkAddress[]. */
   private class Worm {

       Mover mover[];
       int color[][];
       LinkAddress linkAddress[];

       private Worm() {
           mover = new Mover[41];
           color = new int[21][4];
           linkAddress = new LinkAddress[21];
       }

       private Worm(Mover n[]) {
           mover = new Mover[21];
           color = new int[21][4];
           linkAddress = new LinkAddress[21];
           mover = n;
       }

       private Worm(Mover n[], int c[][], LinkAddress d[]) {
           mover = new Mover[21];// Information to move links

           color = new int[21][4]; // color  of each link
           linkAddress = new LinkAddress[21]; // Actual position
```

```
            for (int a = 1; a <= 20; a++) {
                mover[a] = n[a]; // acquisition of information
            }
            for (int a = 1; a <= 20; a++) {
                for (int b = 1; b <= 3; b++) {
                    color[a][b] = c[a][b];
                }
            }

            for (int a = 1; a <= 20; a++) {
                linkAddress[a] = d[a];
            }
        }

        private Worm(Worm worm1) {
            // Information to move links
            mover = new Mover[21];
            // color  of each link
            color = new int[21][4];
            // Actual position
            linkAddress = new LinkAddress[21];

            // acquisition of information
            for (int a = 1; a <= 20; a++) {
                mover[a] = worm1.mover[a];
            }

            for (int a = 1; a <= 20; a++) {
                for (int b = 1; b <= 3; b++) {
                    color[a][b] = worm1.color[a][b];
                }
            }

            for (int a = 1; a <= 20; a++) {
                linkAddress[a] = worm1.linkAddress[a];
            }
        }
    }
```

```
//Global definition.
//Global definitions must be static
static Move vectMoves[];
static Worm[] worm;
static Worm[] wormTwo;

static int nWorms;   //Less than 20
static int width;
static int lifeSpan;
static int initialRow;
static int order[];
static int numbPointMutations;
static int velocity;
static int nOriginalMovers, nCopies;
static int nLinks, nMovers, factor;
static int basalColor;
static int nGen;

static int thereWasAdvance;
static int randInitio;
static int xWin, yWin;

static boolean occupied[][];
static int fitness[];

//Automatic ignition
@Override
public void init() {
    // repaint();
    // Measures of the Applet
    xWin = 500;
    yWin = 400;
    setSize(xWin, yWin);
}

//Definition, assignment of memory.
private void constructors() {
    // System.out.println("constructors   ");
    vectMoves = new Move[10];
    worm = new Worm[21];
```

```
        wormTwo = new Worm[21];
        order = new int[21];
        occupied = new boolean[201][101];
        fitness = new int[21];
    }

    // To avoid overwriting of worms,
    // an occupation record is kept:
    // The screen is divided in cells 4x4.
    // At the beginning every cell is empty=false.
    // Occupancy is recorded in occupied[][],
    // which is a boolean matrix.
    // A boolean  variable can be
    // true= occupied,  else false= not occupied.
    private void occupationMatrix() {
        // System.out.println("occupationMatrix");
        for (int i = 1; i <= 200; i++) {
            for (int j = 1; j <= 100; j++) {
                occupied[i][j] = false;
            }
        }
    }

    //Here we define 9 cardinal points
    //that define 9 moves.
    //These moves form the
    //programming language for the worm.
    private void moveDef() {
        // System.out.println("MoveDef   ");
        final Move move1 = new Move(0, 1);
        vectMoves[1] = move1;
        final Move move2 = new Move(1, 1);
        vectMoves[2] = move2;
        final Move move3 = new Move(1, 0);
        vectMoves[3] = move3;
        final Move move4 = new Move(1, -1);
        vectMoves[4] = move4;
        final Move move5 = new Move(0, -1);
        vectMoves[5] = move5;
        final Move move6 = new Move(-1, -1);
```

```
        vectMoves[6] = move6;
        final Move move7 = new Move(-1, 0);
        vectMoves[7] = move7;
        final Move move8 = new Move(-1, 1);
        vectMoves[8] = move8;
        final Move move9 = new Move(0, 0);
        vectMoves[9] = move9;
        /* System.out.println(vectMoves[6].x);
System.out.println(vectMoves[6].y);
System.out.println(vectMoves[7].x);
System.out.println(vectMoves[7].y); */
    }

    //Initialization of parameters
    private void setParameters() {
        /*  Each worm has its color
Each worm has nlinks links,
each link has its own coordinates on the screen,
which are kept in the linkAddress.
The worms are posited at the bottom of the screen.
They head are at row InitialRow  */
        //InitialRow is the place in the screen
        //where the race begins
        initialRow = 350;

        // Nworms is the number of worms,
        // Maximum allowed value 20.
        nWorms = 8;
        //Width of the worm
        width = 4;
        // NGen is the number of generations
        nGen = 2000000000;

        // Each worm has nlinks number of links.
        //nlinks shall be smaller than 20.
        //Initial number of links
        nLinks = 2;

        //Each chromosome has NMovers
        //number of instructions.
```

```
        //NMovers shall be smaller than 40
        factor = 2;
        nMovers = factor * nLinks;

        // LifeSpan is the number of   days
        // in a life.
        // Life ends with reproduction and dead.
        lifeSpan = 4 * nMovers;

        //NumbPointMutations  is the number of
        //point mutations of the offspring
        numbPointMutations = 2;

        //Gene amplification follows:
        //NOriginalMovers is the number of different Movers
        //to be copied.
        nOriginalMovers = 2;

        //Ncopies is the number of new copies
        //of each copied Mover.
        nCopies = 2;

        //Velocity is among 0 and 10000,
        //it regulates  the velocity of the
        //movements in the display.
        velocity = 10000;

        /*To generate a repeatable sequence
        of random numbers, activate the next
        two lines:

       long seedValue = 40;
       r.setSeed( seedValue) ;*/
    }

    /*
  Movement at random,
    without any purpose,
    without any restriction. */
    private void initialChromosome(Worm worm1) {
```

```
        int n;
        for (int m = 1; m <= nMovers; m++) {
            //Select  which link to move:
            // Throw a number from 1 to nLinks

            n = r.nextInt(nLinks) + 1;
            // Symbolic initialization
            Move move1 = new Move(1, 1);
            Mover mover1 = wormsRec.new Mover(1, move1);
            // true initialization
            mover1.link = n;

            //Select  which direction to move on
            //Throw a number from 1 to 9
            //Move the tail first
            //and then the others links.
            mover1.move = vectMoves[r.nextInt(9) + 1];

            worm1.mover[m] = mover1;
        }
    }

    // All worms are posited at the same initial row
    // in vertical position
    private void initialPosition(Graphics g, Worm worm[]) {
        int k, a, b;

        for (int W = 1; W <= nWorms; W++) {
            // We define an auxiliary worm
            Worm worm1  = wormsRec.new Worm();

            for (int l = 1; l <= nLinks; l++) {
                //Here is the head
                k = initialRow - width * nLinks;
                // The distance among worms is 40.
                // width is the width of  a link
                a = 40 * W;
                b = k + width * l;
                LinkAddress ad = wormsRec.new LinkAddress(a, b);
                worm1.linkAddress[l] = ad;
```

```
                // The first generation is initialized
                // by randomness, others by inheritance.
                int c1 = 1, c2 = 1, c3 = 1; //symbolic initialization
                if (randInitio == 1) {
                    // a color is assigned to each link
                    // but different than white
                    c1 = r.nextInt(256);
                    c2 = r.nextInt(256);
                    c3 = r.nextInt(256);
                    if (c1 + c2 + c3 == 765) {
                        c3 = 1;
                    }
                    //colors are memorized.
                    worm1.color[l][1] = c1;
                    worm1.color[l][2] = c2;
                    worm1.color[l][3] = c3;
                }

                // The color is prepared
                g.setColor(new Color(c1, c2, c3));
                // A Rectangle is defined by
                // the coordinates of the
                // superior right corner,
                // width and hight.
                // The link is painted
                g.fillRect(a, b, width, width);

                int m =   a / width;
                int n =   b / width;
                /* System.out.println
("occupied = " + m + "  " + n );*/
                occupied[m][n] = true;
            }
            //The know-how to move.
            if (randInitio == 1) {
                initialChromosome(worm1);
            }
            // Memorization
            worm[W] = worm1;
```

```
        }
        /* System.out.println("asd = " +
worm[12].linkAddress[2].x );*/
    }

    // Parameters are set and
    // worms are posited at the start line
    private void setBeginning(Graphics g) {
        setParameters();
        // Initialization at random
        randInitio = 1;
        initialPosition(g, worm);
        randInitio = 0;
    }

    // Beginning from the bottom
    // with new genes
    private void reInitialization(Graphics g, Worm worm[]) {
        randInitio = 1;
        occupationMatrix();
        initialPosition(g, worm);
    }

    // A worm is and must remain a unity.
    private int disruptive(int m, Worm worm) {
        // System.out.println("disruptive");
        // A move is disruptive if the target address
        // is farther than 1.5 units from
        //  the neighboring links.
        int n;
        int x, y, x1, y1, x2, x3, y2, y3;
        double d1, d2;
        d2 = 0;

        //Reads the link to be moved
        n = worm.mover[m].link;
        //Reads the   address of the link;
        x = worm.linkAddress[n].x;
        y = worm.linkAddress[n].y;
        //The   move to be done is read
```

```
Move move1 = worm.mover[m].move;

//The target address is calculated
x1 = x + width * move1.x;
y1 = y - width * move1.y;

// The addresses of the two neighboring links is read
// The disruption is calculated
if ((1 < n) & (n < nLinks)) {
    x2 = worm.linkAddress[n - 1].x;
    y2 = worm.linkAddress[n - 1].y;
    d1 = Math.pow((Math.pow((x2 - x1), 2)
            + Math.pow((y2 - y1), 2)), 0.5);
    x3 = worm.linkAddress[n + 1].x;
    y3 = worm.linkAddress[n + 1].y;

    d2 = Math.pow((Math.pow((x3 - x1), 2)
            + Math.pow((y3 - y1), 2)), 0.5);

    // Here we decide whether or not
    // a move is disruptive
    if ((d1 > 1.4 * width) || (d2 > 1.4 * width)) {
        return 1;
    } else {
        return 0;
    }
}

if (n == 1) {
    x2 = worm.linkAddress[2].x;
    y2 = worm.linkAddress[2].y;
    d1 = Math.pow((Math.pow((x2 - x1), 2)
            + Math.pow((y2 - y1), 2)), 0.5);
    if (d1 > 1.1 * width) {
        return 1;
    } else {
        return 0;
    }
}
```

```
        if (n == nLinks) {
            x2 = worm.linkAddress[nLinks - 1].x;
            y2 = worm.linkAddress[nLinks - 1].y;
            d2 = Math.pow((Math.pow((x2 - x1), 2)
                    + Math.pow((y2 - y1), 2)), 0.5);
        }
        if (d2 > 1.1 * width) {
            return 1;
        } else {
            return 0;
        }
    }



    //Worms begin to move
    private void moveOn(Graphics g, int m, Worm worm) throws
InterruptedException {
        int n, i, j;

        // System.out.println("moveOn");
        // Read  the link to be moved
        n = worm.mover[m].link;

        //To move a link:
        //the cell occupied by  the link is cleared,
        //the cell to be occupied is painted.
        //The occupied cell is cleared
        i = worm.linkAddress[n].x;
        j = worm.linkAddress[n].y;
        int k =   i / width;
        int l =   j / width;

        /*System.out.println("TTT" + occupied[k][l]);
   System.out.println("occupied = " + k + "   "
     + l );*/
        occupied[k][l] = false;
        // System.out.println(i+j+n);
        // the white color is prepared
        g.setColor(Color.white);
```

```
        g.fillRect(i, j, width, width);

        //The move to be done is read
        Move move1 = worm.mover[m].move;

        i = i + width * move1.x;
        j = j - width * move1.y;
        worm.linkAddress[n].x = i;
        worm.linkAddress[n].y = j;

        //Movement=repainting in the new place;
        g.setColor(new Color(
                worm.color[n][1], worm.color[n][2], worm.color[n][3]));
        g.fillRect(i, j, width, width);
        //Cell is occupied
        k = k + move1.x;
        l = l - move1.y;
        occupied[k][l] = true;

        //If movement is too fast, nothing can be seen.
        //To attune the velocity of the worm,
        //augment or diminish the delay as pleased:
        Thread.sleep(1);
    }

    // The instructions about link m
    // are executed.
    //This method is equivalent to the cell ribosome
    //and to the CPU of a computer.
    private void ribosome(Graphics g, int m, Worm worm)
            throws InterruptedException {
        // System.out.println("ribosome");
        int n, i, j;

        Move move;
        int x, y, x1, y1;

        //Reads the link  to be moved
        // System.out.println("m = " + m);
        //The   move to be done is read
```

```
        move = worm.mover[m].move;
        //The   link to be moved is read
        n = worm.mover[m].link;
        //Reads the   address of the link;
        x = worm.linkAddress[n].x;
        y = worm.linkAddress[n].y;

        //The target address is calculated
        x1 = x + width * move.x;
        y1 = y - width * move.y;

        //Movement must remain in within barriers
        if ((x1 >= 1) & (y1 >= 1) & (x1 < 500)
                & (y1 < 400)) {

            //Is the target address occupied?
            // Is the move disruptive?
            i =   x1 / width;
            j =   y1 / width;

            //  System.out.print("i= " + i + "  j= " + j);
            /*   System.out.println("occupied = " +
           occupied[i][j] );*/

            //Warning: bad moves are not selected against,
            //instead they are filtered.
            //This is done for the sake of hurried observers
            if ((occupied[i][j] == false)
                    & (disruptive(m, worm) == 0)) {
                moveOn(g, m, worm);
            }
        }
    }

    // Every worm is assigned a fitness
    private void assessFitness() {
        // System.out.println("Fitness");
        int champ;
        int max, man;
```

```
        // The nearer to the top, the fitter
        for (int W = 1; W <= nWorms; W++) {
            // We catch the farthest link
            max = 1;
            for (int m = 1; m <= nLinks; m++) {
                man = worm[W].linkAddress[m].y;
                if (man > max) {
                    max = man;
                }
            }
            fitness[W] = 1000 - max;
            //System.out.println( W + " " + fitness[W] );
        }

        //   Now, we sort worms by fitness
        //   and select the best four,
        //   whose indexes are recorded in Order.
        for (int W = 1; W <= 4; W++) {
            champ = 1;
            for (int B = 1; B <= nWorms; B++) {
                if (fitness[B] > fitness[champ]) {
                    champ = B;
                }
            }
            order[W] = champ;
            fitness[champ] = 0;
        }
    }

    //Blind point mutations
    private void pointMutation() {
        int m, n;
        //Let us try out blind point mutations
        //in specific Movers,
        //in the  link  to be moved
        //and in the direction of movement.
        for (int W = 1; W <= nWorms; W++) {//Each offsprings receives
numbmutation mutations
            for (int k = 1; k <= numbPointMutations; k++) {//Select at random
which Mover to mutate:
```

```
                //throw a number from 1 to NMovers
                m = r.nextInt(nMovers) + 1;
                //Select at random which link  to move:
                //throw a number from 1 to Nlinks.
                n = r.nextInt(nLinks) + 1;
                worm[W].mover[m].link = n;
                //Select at random which direction to move on.
                //Throw a number from 1 to 9
                worm[W].mover[m].move = vectMoves[r.nextInt(9) + 1];
            }
        }
    }

    // Some movers are overwritten with others
    // in within the same worm
    private void geneAmplification() {
        //  System.out.println("geneAmplification");
        int e, n, ee;

        Worm worm1;


        for (int W = 1; W <= nWorms; W++) {
            worm1 = wormTwo[W];
            initialChromosome(worm1);
            for (int m = 1; m <= nOriginalMovers; m++) {//Select  whic
to
be amplified:
                //throw a number from 1 to NMovers
                e = r.nextInt(nMovers) + 1;

                Move move1 = worm1.mover[e].move;

                wormTwo[W].mover[e].move = move1;
                n = wormTwo[W].mover[e].link;

                move1 = worm[W].mover[e].move;
                //The Mover is amplified
                for (int k = 1; k <= nCopies; k++) {//Select at random
be copied:
```

```
                    //throw a number from 1 to NMovers.
                    ee = r.nextInt(nMovers) + 1;
                    worm[W].mover[ee].link = n;
                    worm[W].mover[ee].move = move1;
                }
            }
        }
    }

    // Old worms are erased.
    private void death(Graphics g) {
        // The screen is cleared
        g.setColor(Color.white);
        g.fillRect(0, 0, xWin, yWin);
        // The worm array is overwritten with
        // the wormTwo array.
        worm = wormTwo;
    }

    // Point mutation and gene amplification
    private void mutation() {
        // Point mutation dominates over
        // gene amplification.
        geneAmplification();
        pointMutation();
    }

    // A new link  is added to all worms.
    private void oneMoreLink() {
        // System.out.print("oneMoreLink: ");
        int lim1;
        int n;
        // The link  is added at the tail
        lim1 = nMovers + 1;
        nLinks = nLinks + 1;
        // System.out.println(nLinks);

        nMovers = factor * nLinks;
        Worm worm1;
        for (int W = 1; W <= nWorms; W++) {
```

```
            worm1 = worm[W];
            //New movers are filled in information
            for (int m = lim1; m <= nMovers; m++) {

                //Select at random which link to move:
                //Throw a number from 1 to nLinks
                n = r.nextInt(nLinks) + 1;
                // Symbolic initialization
                Move move1 = new Move(1, 1);
                Mover mover1 = wormsRec.new Mover(1, move1);
                // true initialization
                mover1.link = n;

                //Select  which direction to move on
                //Throw a number from 1 to 9
                //Move the tail first and then the others links.
                mover1.move = vectMoves[r.nextInt(9) + 1];

                worm1.mover[m] = mover1;

                worm[W].mover[m] = worm1.mover[m];
            }
        }
    }

    // Is there any progress?
    private int thereWasAdvance() {
        //System.out.println("thereWasAdvance");
        //Our criterion for   advance is
        //that the head has surpassed
        //its original vertical position.
        int headPosition, advance;
        advance = 0;

        for (int W = 1; W <= nWorms; W++) {
            Worm worm1;
            worm1 = worm[W];
            headPosition = worm1.linkAddress[1].y;
            if (headPosition < initialRow - width * nLinks) {
                advance = 1;
```

```
            }
        }
        /* if (advance == 1)
      System.out.println("Advance");*/
        return advance;
    }

    //If at least one worm   advanced even a bit,
    //  the next population is enlarged by one link:
    private void growth() {
        //  System.out.println("growth");
        if (thereWasAdvance() == 1) {
            oneMoreLink();
        } else;
    }

    //* Winners reproduce others not
    private void reproduction(Graphics g) {
        //Our reproductive scheme is as follows:
        //50% offspring for  the champion,
        //25% offspring for  the second,
        //12.5% for   the third,
        //12.5% for   the fourth.
        //We place children on order.
        //Children are clones of the fathers
        //that are copied to the wormTwo array.

        //Call InitialPosition(8)
        int champ = order[1];
        Worm worm1;
        worm1 = wormsRec.new Worm(worm[champ].mover,
                worm[champ].color, worm[champ].linkAddress);
        /*  System.out.println("Address1 = "
            + worm1.linkAddress[2].x); */

        int half =   nWorms / 2;
        int quarter =   nWorms / 4;
        // System.out.println("Champ = " + order[1] );

        for (int W = 1; W <= half; W++) {
```

```
            wormTwo[W] = worm1;
        }

        int limf = half + quarter;
        // System.out.println("limf = " + limf);

        champ = order[2];
        worm1 = wormsRec.new Worm(worm[champ].mover,
                worm[champ].color, worm[champ].linkAddress);

        for (int W = half + 1; W <= limf; W++) {
            wormTwo[W] = worm1;
        }

        int limi = limf;
        limf = limf + quarter / 2;

        champ = order[3];
        worm1 = wormsRec.new Worm(worm[champ].mover,
                worm[champ].color, worm[champ].linkAddress);
        for (int W = limi + 1; W <= limf; W++) {
            wormTwo[W] = worm1;
        }

        champ = order[4];
        worm1 = wormsRec.new Worm(worm[champ].mover,
                worm[champ].color, worm[champ].linkAddress);
        for (int W = limf + 1; W <= nWorms; W++) {
            wormTwo[W] = worm1;
        }
        // System.out.println("limf = " + limf);
        /*   System.out.println("Address1= " +
    worm[order[1]].linkAddress[2].x ); */
        /*  System.out.println("Address2 = " +
     wormTwo[order[1]].linkAddress[2].x);*/
        death(g);
        mutation();
        growth();
    }
```

```
//Recombination is implemented
private void recombination() {
    //Every individual is replaced by a recombinant
    for (int i = 1; i <= nWorms; i++) {
        // A  recombinant operator is assembled at random
        int rec[] = new int[21];
        for (int j = 1; j <= nLinks; j++) {
            rec[j] = r.nextInt(2);
        }

        // Two worms are chosen at random
        int m = r.nextInt(nWorms) + 1;
        int n = r.nextInt(nWorms) + 1;

        Worm worm1;
        worm1 = wormsRec.new Worm(worm[m].mover,
                worm[m].color, worm[m].linkAddress);

        Worm worm2;
        worm2 = wormsRec.new Worm(worm[n].mover,
                worm[n].color, worm[n].linkAddress);

        // Recombination is executed.
        Worm worm3;
        worm3 = wormsRec.new Worm(worm[1].mover,
                worm[1].color, worm[1].linkAddress);

        for (int j = 1; j <= nLinks; j++) {
            if (rec[j] == 1) {
                worm3.mover = worm1.mover;
            } else {
                worm3.mover = worm2.mover;
            }
        }

        // Recombinant is memorized
        wormTwo[i] = worm1;
    }
    // Old population is replaced by
```

```
        // the recombinant one

        worm = wormTwo;
    }


    //A life for all worms is calculated
    private void oneLife(Graphics g) throws InterruptedException {


        // l is the counter of days of life.
        // m is the number of the mover;
        // W is the index of the worm.
        for (int l = 1; l <= lifeSpan; l++) {
            for (int m = 1; m <= nMovers; m++) {
                for (int W = 1; W <= nWorms; W++) {

                    //Auxiliary worm: worm1
                    Worm worm1;
                    worm1 = worm[W];
                    /* System.out.println("W = " +W +
           " m = " + m + " l= " + l); */
                    ribosome(g, m, worm1);
                    worm[W] = worm1;
                }
            }
        }
    }

    // Evolution is called to function
    private void dynamics(Graphics g) throws InterruptedException {
        System.out.println("Initial length = " + nLinks);
        System.out.println("Link number + waiting time");
        int time1 = 0;
        int n1 = nLinks;
        int waitingTime;
        for (int gen = 1; gen <= nGen; gen++) {
            if (nLinks > n1) {
                n1 = nLinks;
                waitingTime = gen - time1;
```

```
                time1 = gen;
                System.out.println(nLinks + "\t "
                        + waitingTime);

            }
            g.drawString("Gen = " + gen, 20, 20);
            // System.out.println("Gen = " + gen);
            oneLife(g);
            assessFitness();
            //Gene amplification is included.
            //Body enlargement is allowed.
            reproduction(g);
            recombination();
            reInitialization(g, worm);
        } //end of gen loop
    }

    //This is the main method.
    @Override
    public void paint(Graphics g) {
        //System.out.println("Applet");
        constructors();
        occupationMatrix();
        moveDef();
        setBeginning(g);

        try {
            dynamics(g);
        } catch (InterruptedException ex) {
            Logger.getLogger(WormsRec.class.getName()).log(Level.SEVERE, null,
ex);
        }
    }

}//End of Program WormsRec, B71
```

72 ***Exercise*** *Run the program, play with the different parameters. Does recombination accelerate evolution?* Answer

5.2 Conclusion

Cross over or recombination causes many troubles for the developer but nevertheless we found no significant diminishing of arrival time to the solution. Thus, our simulation sheds no light about the reason of the existence and permanence of sexual reproduction in living earth. The possible cause of our failure is that we have simulated a type of evolution in which the environment offers no problem for surviving, thus it is too simple: heavy duty machinery is of no help to solve simple problems. One also may consider that there is a bug in the programming that avoids to see the role of recombination. Or maybe, we have a misconception: Is the type of implemented recombination too chaotic and so useless?

Chapter 6

Accelerating evolution

```
Evolution functions but it is too slow.  Is that irremediable?
```

73 ***Motivation*** *The evolution displayed by the programs of the previous chapter is slow, so much that it gets a burden for the developer. Can evolution somehow be accelerated?*

74 ***Purpose.*** *We implement in this chapter a* ***variable mutation rate*** *to try to accelerate evolution.*

6.1 Local maxims

Evolutionary problems can be reformulated as optimization problems, say, our worms must find the *optimal* way to climb the display. The problem is that evolution is blind and never can make sure that its proposed solution is optimal or nearly optimal. The reason is that there are **false evolutionary solutions**: solutions that resists improvement but that are far from a **global maximum**. False solutions coincide with **local maxims**: Any local maximum leads to stagnation at a low quality solution.

This situation is best described by the next drawing:

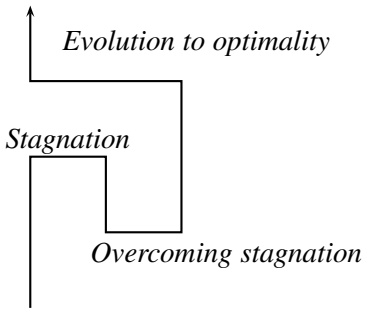


Figura 6.0. A complex world is plagued with local maxims or false solutions that must be overcome.

What can we do to escape a local maximum?

If enough time is given, any non zero mutation rate suffices to escape a local maximum in a very big population. But in this chapter, we try to economize time keeping population size in within tolerable bounds. How? If we detect stagnation, we memorize the achieved solution but we turn on a high level of mutation. This is a particular case of **regulated mutation rate.**

75 ***Research*** *Are there genes that in molecular biology warms the mutation rate? If they do not exist, explain their absence. Explain how one can engineer one of them.* Answer

6.2 Implementation

Let us make two experiments. In the first one, we retake a previous program in which we found a zero of a fifth degree polynomial and in which we observed a false solution. In the second experiment, we try to accelerate the evolutionary rate of our worms to improve on the learning capacity to climb the screen.

76 ***Exercise*** *Run the following program whose aim is to find a root of a polynomial. Verify that under low resources (small population size, say 100, and some few generations, say, 6), one arrives to a local minimum, a false solution. To scape from the local optimum one must invest more resources, say augment population size to 300 and run the program during some 500 generations. When population size is too small, one needs hundreds of thousands of generations to reach at a root.*

```
// Program polynomial, B76
//This s a clone of program A235 Equation2 of Vol I.
// Evolution of a population
// of strings that finds a root
// of a polynomial equation.

package ejvol2v2p;


import java.util.Random;

public class Polynomial
{
    // Strings are at the same time
    // genotype and phenotype.

    // Pre-declaration of global variables.
    // They are used all throughout the whole class.

    // Individuals are kept in the array
    // Individual[]. It is an array of strings.
    // The number of individuals must be
    // less than limit

    private static final int POPSIZE = 100;
    static private double Fitness[];
    static private String Individual[ ];//Predeclaration
    // Order  is an array of integers.
    static private int Order[];//pre-declaration
    static private int Generation;
    static private int  nGen;
    static private int ReportMin[], ReportMax[];
    private  static int numberOfClones ;
    private static final double MUTATIONRATE = 0.2;
    private  static int Champ;

    // Turn on of the random generator
    private static final  Random RAND = new Random();
    private static final boolean  PRINT = false;
```

```
// This method generates random chars
// that represent digits
private  static  Character  randomChar()
{
    return  ( char ) ( RAND.nextInt( 10 ) + 48  );
}


private static void Initialization( )
{
    /* We generate POPSIZE individuals (strings)
    Sequences are completely random */

    //Formal declaration of our array.
    Individual= new String[POPSIZE];
    //System.out.PRINTln("ORIGINAL POPULATION");
    for(int i = 0; i< POPSIZE; i++)
    {
        //Initialization  in blank.
        Individual[i]="";

        //An individual is assembled char by char
        for(int j = 0; j<  9; j++)
        {
            // We generate a random numeric char
            Character c =  randomChar();
            //System.out.PRINTln( "char = " + c);
            // Char c is converted into a string s
            String s = c.toString();
            // s is concatenated to Individual[i]
            Individual[i] = Individual[i]+s;
        }
        // We add the place for a point.
        Character n =  randomChar();
        String s = n.toString();
        Individual[i] = s+Individual[i] ;

        //We add a sign: zero or one.
        n = (char) ( RAND.nextInt( 2 ) + 48  );
        s = n.toString();
```

```
            Individual[i] = s+Individual[i] ;
            //System.out.PRINT("Individual " +  i + "\t");
            // System.out.PRINTln( Individual[i]);
        }


        //Declarations and default initializations
// of other arrays.

        Order= new int[POPSIZE];
        ReportMin= new int[POPSIZE];
        ReportMax= new int[POPSIZE];
        Fitness= new double[POPSIZE ];

        for(int i = 0; i< POPSIZE; i++)
        {
            Order[i]=0;
            ReportMin[i]=0;
            ReportMax[i]=0;
        }
    }

    //The encoded number is withdrawn
    private static double decoder(String s)
    {
        // We get the sign:
        char s1  =   s.charAt(0);
        int sign = Character.getNumericValue(s1);
        // We get the place of the point,
        // which implies multiples of ten.
        s1  =   s.charAt(1);
        int  point = Character.getNumericValue(s1);
        // We build the number with the other chars
        double number = 0;
        for(int j=2;j  <  11;j++ )
        {
            char s3  =   s.charAt(j);
            int  l = Character.getNumericValue(s3);
            int k = j-1;
            int power = point-k;
```

```
            //number1 = l * 10^power
            double number1 =  l * Math.pow(10,power);
            number = number + number1;
        }
        if (sign == 0) return (-1) *number;
            else return  number;
    }

    //The method encodes the equation = 0 to solve.
    //Perfect solution must ne a root and output zero.
    //Other  numbers represents an error
    private static double functionValue(double b)
    {
        //Root = 0.75;
        return 4*b+3;
        //Root = root = -4.71817809;
        //return b*b*b*b*b + 4*b*b*b*b - 3*b*b*b + 2*b*b +b +1;
        //Root = 1;
        //return b*b - 2*b + 1;
    }



    //The fitness of each individual is calculated
    private static void fitness()
    {
       for(int i = 0; i< POPSIZE; i++)
        {
            // Individual[i] is a string,
            // it is a genotype.
            // b is a number: it is the phenotype
            double b = decoder(Individual[i]);
            double  fV = functionValue(b);
            //Fitness(root) = 1
            //Others have Fitness less than 1
            Fitness[i] =    1/(1+fV * fV);
        }
    }
```

```
//Strings are sorted by fitness
private static void Sorting()
{

    fitness();

    for(int i = 0; i< POPSIZE;i++)
    {
        int champ = 0;
        for(int j = 0; j< POPSIZE; j++)
            if (Fitness[j] >= Fitness[champ])   champ = j;
        //The array Order classifies individuals by fitness.
        //The fittest, with less errors, go to the 0-th entry.
        Order[i] = champ;
        Fitness[champ] = 0;
    }


    String s = Individual[Order[0]];
    double b = decoder(s);
    double p = functionValue(b);
    System.out.println("Gen = " + Generation + " best solution  = "
    + b + " Error = " + p );
}




//The top   fill the world
private  static void Cloning()
{
    //Array to host cloned individuals
String Individual2[ ] = new String[100];
int   i=0;
int   k=0;         //rank, 0 is the highest
//Cloning
while (i<POPSIZE)
{
    int counter = numberOfClones;
```

```
        while ((counter>0) & (i<POPSIZE))
        {
            Individual2[i] = Individual[Order[k]];
            i++;
            counter= counter-1;
        }
        k++;
    }


    //Cloned individuals become the official population
    for(int j = 0; j < POPSIZE; j++)
        Individual[j] = Individual2[j];
    if (PRINT) printPopulation();
    }

    private static void Reproduction()
    {
        //Individuals are sorted by fitness
        Sorting();
        //for(int i = 0; i<  10;i++)
        //System.out.PRINTln(  Individual(Order(i))

        //Fitest individuals are cloned
        Cloning();
    }


private static  String change(String a)
{
    for(int l = 0; l < a.length(); l++)
    {
        double s = RAND.nextDouble();
        if (s < MUTATIONRATE)
        {
            if (a.length()==1) return ""+randomChar();
            else
            {
                //In case of a ghost
                if ( (l<0) || (l > a.length()-1))
```

```
                {

                    return a;
                }
                a = a.substring(0, l) + randomChar()
                    +  a.substring(l + 1);
            }
        }
    }
    return a;
}



    private static void Mutation()
    {
        for(int i=0; i < POPSIZE; i++)
       if (i != Champ) Individual[i] = change(Individual[i] );

    }




private static String recombinant(int i)
{
    //A partner for i is found
    int j = RAND.nextInt(POPSIZE);
    //A place for recombination
    int lengthI =  Individual[i].length();
    int lengthJ =  Individual[j].length();
    int minLength = Math.min(lengthI,lengthJ);
    int place = RAND.nextInt(minLength);
    String recombinant = Individual[i].substring(0, place)
            + Individual[j].substring(place);
    /*
    System.out.PRINTln( "Inputs = " + Individual[i]
            + "    " + Individual[j]
    + " Recombinant = " + recombinant);
```

```
    */
    return recombinant;
}


//Two strings are taken as input
//and one recombinant is produced
private static void Recombination()
{


    //Array with recombinant individuals
    String Individual2[ ] = new String[100];
    for(int i = 0; i< POPSIZE; i++)
    {
        Individual2[i] = recombinant(i);
    }
    //Recombinants become the official population
    for(int i = 0; i< POPSIZE; i++)
    {
        Individual[i] = Individual2[i];
    }

}


    private static void Dynamics()
    {
        //The top ten are allowed to  reproduce:
        //a child occupies the place of the shortest individual
        Reproduction();
        //The new population is subjected to mutation
        Mutation();
        Recombination();
    }


    private static void printPopulation()
    {
        //Cloned individuals become the official population
```

```
    for(int j = 0; j < POPSIZE; j++)
    {

        double b = decoder(Individual[j]);
        double  fV = functionValue(b);
        double    myFitness =    1/(1+fV*fV);
        System.out.println( j + "  ->  "  + Individual[j] + "  ->  " +
myFitness);

    }
    System.out.println();
    }


    public static void main(String[] args)
    {

        Initialization( ) ;
        nGen = 15;
        numberOfClones  = 10;
        //System.out.println("Root of 4*x + 3 = 0,  x = 0.75");
        for(int Gen = 0; Gen < nGen; Gen++)
        {
            Generation = Gen;
            Dynamics();
        }
    }
}//End of Program polynomial, B76
```

77 ***Exercise*** *Device a polynomial that causes the program studied in the previous exercise to get trapped in an inescapable false solution. This shows that when one uses evolution to solve problems one must always take care of the* ***completeness problem****: Is the global solution of the problem in the span of the searching machinery of the algorithm? At the same time we find the next intriguing question: In which sense is natural evolution complete and in which sense not?* Answer

78 ***Regulating mutation rate***

Everything is possible for evolution but a price must be paid in terms of resources, population size and number of generations that the program must run. It

happens that some problems are immediately solved but others seem to demand more resources than available. That is why everyone is trying to invent methods to accelerate the performance of genetic algorithms.

To get the flavor of those investigations, let us play a bit with varying mutation rates. Recombination is not included. Mutation can vary as in time as in space assigning different levels for different encoding values. Our general purpose is to detect stagnation and if that happens to occur, a high level of mutation is turned on to see what happens. In that way we device a **varying mutation rate.** Warning: since a high mutation is equivalent to chaos, one must program a large population size to shield its destructive role.

To detect stagnation, we follow the changes in fitness: if during 100 generations it does not change enough, stagnation is declared and the mutation rate is doubled. The code follows:

```
// Program varMutation, B78
// Evolution of a population
// of strings that finds a root
// of a fifth degree polynomial.
// Mutation rate is variable in the hope of
// escaping from local minima.
package ejvol2v2p;

import java.util.Random;

public class Polynomial2 {

    // Strings are at the same time
    // genotype and phenotype.
    // They are like ribozymes,
    // but more advanced.
    // Pre-declaration of global variables.
    // They are used all throughout the whole class.
    // Individuals are kept in the array
    // Individual[]. It is an array of strings.
    // The number of individuals must be
    // less than limit
    private static final int POPSIZE = 100;
    static double Fitness[];
    static String Individual[];//Predeclaration
    static String PatternParasite, PatternFood;
```

```
    // Order  is an array of integers.
    static int Order[];//pre-declaration
    private static int Champ;
    static String b;
    static int generation;
    private  static int numberOfClones ;
    static int nGen;
    static int ReportMin[], ReportMax[];
    static double mutationRate;
    static double oldError, newError;
    static double deltaError;
    private static final boolean  PRINT = false;

    //  Turn on of the random generator
    static Random r = new Random();

    private static char randomChar() //  This method generates random chars
    // that represent digits
    {
        return (char) (r.nextInt(10) + 48);
    }

    private static void Initialization() {
        /* We generate POPSIZE individuals (strings)
15 characters long.
Sequences are completely random */

        //Formal declaration of our array.
        Individual = new String[POPSIZE];
        //  System.out.println("ORIGINAL POPULATION");
        for (int i = 0; i < POPSIZE; i++) {
            //Initialization in blank.
            Individual[i] = "";

            //An individual is assembled char by char
            for (int j = 1; j <= 9; j++) {
                // We generate a random numeric char
                char c = randomChar();
                //System.out.println( "char = " + c);
                // Char c is converted into a string s
```

```
                String s = "" + c;
                // s is concatenated to Individual[i]
                Individual[i] = Individual[i] + s;
            }
            //We add the place for a point.
            char n = randomChar();
            String s = "" + n;
            Individual[i] = s + Individual[i];

            // We add a sign: zero or one.
            n = (char) (r.nextInt(2) + 48);
            s = "" + n;
            Individual[i] = s + Individual[i];
            //System.out.print("Individual " +  i + "\t");
            // System.out.println( Individual[i]);
        }

        // Declarations and default initializations
        // of other arrays.
        Order = new int[POPSIZE];
        ReportMin = new int[POPSIZE];
        ReportMax = new int[POPSIZE];
        Fitness = new double[POPSIZE];
        for (int i = 0; i < POPSIZE; i++) {
            Order[i] = 0;
            ReportMin[i] = 0;
            ReportMax[i] = 0;
            Fitness = new double[POPSIZE];
        }
    }

    private static double decoder(String s) {
        // This method transforms our strings in numbers
        // We get the sign:
        char s1 = s.charAt(0);
        int sign = Character.getNumericValue(s1);
        // We get the place of the point,
        // which implies multiples of ten.
        s1 = s.charAt(1);
        int point = Character.getNumericValue(s1);
```

```
        // We build the number with the other chars
        double number = 0;
        for (int j = 2; j < 11; j++) {
            char s3 = s.charAt(j);
            int l = Character.getNumericValue(s3);
            int k = j - 1;
            double number1 = (int) l * Math.pow(10, point - k);
            number = number + number1;
        }
        if (sign == 0) {
            return (-1) * number;
        } else {
            return number;
        }
    }

    //The method encodes the equation = 0 to solve.
    //Perfect solution must ne a root and output zero.
    //Other  numbers represents an error
    private static double functionValue(double b)
    {
        //Root = 0.75;
        //return 4*b+3;
        //Root = root = -4.71817809;
        return b*b*b*b*b + 4*b*b*b*b - 3*b*b*b + 2*b*b +b +1;
        //Root = 1;
        //return b*b - 2*b + 1;
    }


    //The fitness of each individual is calculated
    private static void fitness()
    {
       for(int i = 0; i< POPSIZE; i++)
        {
            // Individual[i] is a string,
            // it is a genotype.
            // b is a number: it is the phenotype
            double b = decoder(Individual[i]);
            double  fV = functionValue(b);
```

```
            //Fitness(root) = 1
            //Others have Fitness less than 1
            Fitness[i] =    1/(1 + fV * fV);
        }
    }


    //Strings are sorted by fitness
    private static void Sorting()
    {

        fitness();

        for(int i = 0; i < POPSIZE;i++)
        {
            int champ = 0;
            for(int j = 0; j< POPSIZE; j++)
                if (Fitness[j] >= Fitness[champ])   champ = j;
            //The array Order classifies individuals by fitness.
            //The fittest, with less errors, go to the 0-th entry.
            Order[i] = champ;
            Fitness[champ] = 0;
        }

        Champ = Order[0];
        String s = Individual[Champ];
        double b = decoder(s);
        double p = functionValue(b);
        System.out.println("Gen = " + generation + " best solution  = "
        + b + " Error = " + p );
    }




    //The top   fill the world
    private  static void Cloning()
    {
        //Array to host cloned individuals
```

```
String Individual2[ ] = new String[POPSIZE];
int   i=0;
int   k=0;         //rank, 0 is the highest
//Cloning
while (i < POPSIZE)
{
    int counter = numberOfClones;
    while ((counter > 0) & (i < POPSIZE))
    {
        Individual2[i] = Individual[Order[k]];
        i++;
        counter= counter-1;
    }
    k++;
}


//Cloned individuals become the official population
for(int j = 0; j < POPSIZE; j++)
    Individual[j] = Individual2[j];
if (PRINT) report(generation );
}

private static void Reproduction()
{
    //Individuals are sorted by fitness
    Sorting();
    //for(int i = 0; i<  10;i++)
    //System.out.PRINTln(  Individual(Order(i))

    //Fitest individuals are cloned
    Cloning();
}


private static void report(int gen) {
    for(int j = 0; j< POPSIZE; j++)
    {
        System.out.print("gen " + gen + " Ind " + j );
        System.out.println(" " +  Individual[j]);
```

```
        }
        String c = Individual[POPSIZE]; //champ reported
        double d = decoder(c);
        double error = functionValue(d);
        System.out.println("genotype = " + c);
        System.out.println("Phenotype" + b);
        System.out.println(" Error = " + error);
    }


    private static void Mutation() {
        //We consider only one type of mutation:
        //substitution of one char by another one.


        char c, n;
        String s;
        String sign;
        String place;
        //All the individuals mutate except the champ = 0;
        for (int j = 1; j < POPSIZE; j++) {
            //Mutation of digits
            // w = mutated individual.

            String w = "";

            for (int k = 2; k <= 10; k++) {
                //Change digit k?
                double p = r.nextDouble();
                if (p < mutationRate) {
                    c = randomChar();
                } else { //no mutation
                    c = Individual[j].charAt(k);
                }
                // Char c is converted into a string s
                s = "" + c;
                // s is concatenated to w
                w = w + s;
            }
```

```
            // Change place of point
            double p = r.nextDouble();
            if (p < (3 / 3) * mutationRate) //mutation
            {
                // place redefined
                n = randomChar();
            } //no mutation
            else {
                n = Individual[j].charAt(1);
            }
            place = "" + n;
            w = place + w;
            // Change sign?
            p = r.nextDouble();
            if (p < (1 / 3) * mutationRate) //mutation
            {
                // Sign redefined
                n = (char) (r.nextInt(2) + 48);
            } else { //no mutation
                n = Individual[j].charAt(0);
            }
            sign = "" + n;
            w = sign + w;
            Individual[j] = w;
            /*System.out.println(j + "\t "
    +v+"\t "+ Individual[j]); */
        }
    }

    private static void mutationRedef(int gen) {
        String c = Individual[0];
        double d = decoder(c);
        double error = Math.abs(d);
       System.out.println("gen " + gen +
  " Error " + error);
        /* System.out.println("mutation rate = " +
    mutationRate);*/
        newError = error;
        deltaError = Math.abs(newError - oldError);
        System.out.println("Error= " + error +
```

```
" deltaError= " + deltaError);
        oldError = newError;
        if (deltaError < 0.001) {
            mutationRate = 1.1 * mutationRate;
            if (mutationRate > 1) {
                mutationRate = 1;
            }

        System.out.println("gen " + gen
                    + " mutation was redefined");
        } else;
    }

    private static void dynamics(int gen) {
        generation = gen;
        //The top ten are allowed to  reproduce:
        //a child occupies the place
        //of the shortest individual
        Reproduction( );
        //every 100 generations, stagnation is revised
        if (gen % 100 == 0) {
            mutationRedef(gen);
        }
        //The new population is subjected to mutation
        if (gen == nGen) {
            report(gen);
        }
        Mutation();

    }



    public static void main(String[] args) {
        //To modify POPSIZE,
        // the method Copying()
        // must also be modified.
        System.out.println("Running ");
        Initialization();
```

```
        nGen = 200000;
        oldError = 1000;
        mutationRate = 0.1;
        numberOfClones  = 10;
        for (int gen = 0; gen <  nGen; gen++) {
            dynamics(gen);
        }
    }
}//End of Program varMutation, B78
```

79 ***Exercise*** *Run the program and play with the code. Verify that it escapes the local maximum and approaches a global one (put nGen = 300000;). In relation with the concrete problem, this means that a true root is found in spite of blind alleys that deceive the evolutionary process.*

80 ***Exercise*** *Add recombination to mutation in the previous program to see what happens.* Answer

81 ***Challenge*** *We have used the expression 'evolutionary rate' as a normal expression of the jargon that acquires a concrete meaning from the context we have given to it. Would you explain exactly what it consists in?* Answer

82 ***Challenge*** *We expect that an increasing mutation rate accelerates the evolutionary rate. The reason is that when one is trapped in a local optimum and is far from the solution to a problem it could be very useful to try out many variants. But then a problem arises: if we approach at a high mutation rate a global maximum, most surely, we will pass by and then we will try to return back and so on. The solution is to implement a* ***self regulated mutation rate****: if a maximum is declared satisfactory, the mutation rate shall diminish close to zero so that the maximum could be approached softly and surely. Try this out.*

6.3 Hurrying worms

Let us tinker about to try to accelerate the evolutionary process that teaches the worms of chapter IV to climb the screen.

We have implemented three types of mutation: gene amplification, site mutation and recombination. Site mutation is regulated by the integer *numbPointMutations*, gene amplification depends on the integer *nOriginalMovers* that regulates the number of Movers to be amplified a number *nCopies* of times. To accelerate the evolution rate, we choose to increase the number of point mutations once stagnation has been detected.

83 ***Gedanken worms***

The graphic displaying of worms was good because its main role was to ease the debugging process: one can detect a bug visually at once. Since we made already sure that the program is free from big bugs, we can silence displaying operations and consequently do not appear in the next program.

The code follows:

```
// Program WormsVaryingMutRate, B83
// A population of worms
// is programmed by evolution
// to climb the screen.
// Increased mutation rate
// Needed tools

package ejvol2v2p;

import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;
import java.util.Random;

public class WormsVaryingMutRate extends Applet {

    private static final long serialVersionUID = 1L;

    public WormsVaryingMutRate() {

    }

// Class WormsVaryingMutRate is an archetype.
// The next instruction says that
// one might
// construct a real stuff, an instance, on the
// image and resemblance of the archetype.
// Archetype means old-type, prototype.
// The word archetype does not belong
// in evolutionary biology,
// but it belongs in paleontology.
```

```
// We use here the word archetype as
// a synonym of a plan or design,
// of an idea.
    static WormsVaryingMutRate wormsMutRate = new WormsVaryingMutRate();

// Turn on  the random generator
    Random r = new Random();

// private static final long serialVersionUID = 1L;

    /* We follow the next convention:
    Archetypes are written in upper case,
    instances are written in lower case

    In java, archetypes are called classes.
    It is not mandatory, but
    we define archetypes as  classes
    in within another one: as inner classes.

    Definition of the Move class:
    A Move says in which direction
    one must move:
    we have 8 cardinal points.
    Each point denoted by
    its x and y coordinates.

    An archetype may have
    two types of information:
    one about things
    The other about tasks */
    public class Move {

        int x; // this is a thing
        int y;

        // The next is a method, a function
        // that  declares
        // how information is acquired.
        // It is a constructor.
        // It is overload by two possibilities.
```

```
    // a) The input is a list
    Move(int a, int b) {
        x = a;
        y = b;
    }

    // b) The input is a structure
    Move(Move move) {
        x = move.x;
        y = move.y;
    }

}

// Definition of the class Mover:
// A Mover keeps two types of information:
// a) the link of the worm to be moved and
// b) the direction or move to be chosen.
public class Mover {

    int link;
    Move move;

    //  Movers can acquired information
    // in two ways, they are  overloaded.
    // Form 1: by a list
    public Mover(int a, int b, int c) {
        link = a;
        move.x = b;
        move.y = c;
    }

    // Form 2, by structured data
    public Mover(int a, Move m) {
        link = a;
        move = m;
        move.x = m.x;
        move.y = m.y;
    }
}
```

```
// The class LinkAddress keeps the actual
// position of a link.
public class LinkAddress {

    int x;
    int y;

    // LinkAddress is overloaded:
    // A list feeds it
    LinkAddress(int a, int b) {
        x = a;
        y = b;
    }

    //Structured data feed it
    LinkAddress(LinkAddress l) {
        x = l.x;
        y = l.y;
    }

}

/* Here we define the archetype of worm.
    A worm is composite: it has links.
    It has a chromosome that keeps
    the genetic information.
    The chromosome is the vector Mover[ ],
    which keeps the information
    on how to move its links:
    which  link and in which order
    (in the order that they appear in
    the vector Mover[]), and
    in which direction.
    Each worm has its own color,
    which is defined by three numbers.
    Each worm keeps record of
    the actual position of each
    one of its links,
    in linkAddress[]. */
```

```
public class Worm {

    Mover mover[];
    int color[][];
    LinkAddress linkAddress[];

    public Worm() {
        mover = new Mover[41];
        color = new int[21][4];
        linkAddress = new LinkAddress[21];
    }

    public Worm(Mover n[]) {
        mover = new Mover[21];
        color = new int[21][4];
        linkAddress = new LinkAddress[21];
        mover = n;
    }

    public Worm(Mover n[], int c[][], LinkAddress d[]) {
        // Information to move links
        mover = new Mover[21];
        // color  of each link
        color = new int[21][4];
        // Actual position
        linkAddress = new LinkAddress[21];

        for (int a = 1; a <= 20; a++) {
            mover[a] = n[a]; // acquisition of information
        }
        for (int a = 1; a <= 20; a++) {
            for (int b = 1; b <= 3; b++) {
                color[a][b] = c[a][b];
            }
        }

        for (int a = 1; a <= 20; a++) {
            linkAddress[a] = d[a];
        }
    }
```

```
    public Worm(Worm worm1) {
        // Information to move links
        mover = new Mover[21];
        // color  of each link
        color = new int[21][4];
        // Actual position
        linkAddress = new LinkAddress[21];
        // How to acquire information
        for (int a = 1; a <= 20; a++)
        {
            mover[a] = worm1.mover[a];
        }

        for (int a = 1; a <= 20; a++) {
            for (int b = 1; b <= 3; b++) {
                color[a][b] = worm1.color[a][b];
            }
        }

        for (int a = 1; a <= 20; a++) {
            linkAddress[a] = worm1.linkAddress[a];
        }
    }
}

//Global definition.
//Global definitions must be static
static Move vectMoves[];
static Worm[] worm;
static Worm[] wormTwo;

static int nWorms;   //Less than 20
static int width;
static int lifeSpan;
static int initialRow;
static int order[];
static int numbPointMutations;
static int velocity;
static int nOriginalMovers, nCopies;
```

```
static int nLinks, nMovers, factor;
static int basalColor;
static int nGen;

static int thereWasAdvance;
static int randInitio;
static int xWin, yWin;
static double oldFitness, newFitness;
static double deltaFitness;

static boolean occupied[][];
static int fitness[];

//Automatic ignition
@Override
public void init() {
    // repaint();
    //  Measures of the Applet
    xWin = 500;
    yWin = 400;
    setSize(xWin, yWin);
}

//Definition, assignment of memory.
public void constructors() {
    // System.out.println("constructors   ");
    vectMoves = new Move[10];
    worm = new Worm[21];
    wormTwo = new Worm[21];
    order = new int[21];
    occupied = new boolean[201][101];
    fitness = new int[21];
}

// To avoid overwriting of worms,
// an occupation record is kept:
// The screen is divided in cells 4x4.
// At the beginning every cell is empty=false.
// Occupacy is recorded in occupied[][],
// which is a boolean matrix.
```

```
    // A boolean  variable can be
    // true= occupied,  else false= not occupied.
    public void occupationMatrix() {
        //System.out.println("occupationMatrix");
        for (int i = 1; i <= 200; i++) {
            for (int j = 1; j <= 100; j++) {
                occupied[i][j] = false;
            }
        }
    }

//Here we define 9 cardinal points
    public void moveDef() {
        // System.out.println("MoveDef   ");
        final Move move1 = new Move(0, 1);
        vectMoves[1] = move1;
        final Move move2 = new Move(1, 1);
        vectMoves[2] = move2;
        final Move move3 = new Move(1, 0);
        vectMoves[3] = move3;
        final Move move4 = new Move(1, -1);
        vectMoves[4] = move4;
        final Move move5 = new Move(0, -1);
        vectMoves[5] = move5;
        final Move move6 = new Move(-1, -1);
        vectMoves[6] = move6;
        final Move move7 = new Move(-1, 0);
        vectMoves[7] = move7;
        final Move move8 = new Move(-1, 1);
        vectMoves[8] = move8;
        final Move move9 = new Move(0, 0);
        vectMoves[9] = move9;
        /* System.out.println(vectMoves[6].x);
 System.out.println(vectMoves[6].y);
 System.out.println(vectMoves[7].x);
 System.out.println(vectMoves[7].y); */
    }

    //Initialization of parameters
    public void setParameters() {
```

```
       /*  Each worm has its color
Each worm has nlinks links,
each link has its own coordinates on the screen,
which are kept in the linkAddress.
The worms are posited at the bottom of the screen.
They head are at row InitialRow  */
       //InitialRow is the place in the screen
       //where the race begins
       initialRow = 350;

       // Nworms is the number of worms,
       // Maximum allowed value 20.
       nWorms = 8;
       //Width of the worm
       width = 4;
       // NGen is the number of generations
       nGen = 2000000000;

       // Each worm has nlinks number of links.
       //nlinks shall be smaller than 20.
       //Initial number of links
       nLinks = 2;

       //Each chromosome has NMovers
       //number of instructions.
       //NMovers shall be smaller than 40
       factor = 2;
       nMovers = factor * nLinks;

       // LifeSpan is the number of   days
       // in a life.
       // Life ends with reproduction and dead.
       lifeSpan = 4 * nMovers;

       //NumbPointMutations  is the number of
       //point mutations of the offspring
       numbPointMutations = 2;

       //Gene amplification follows:
       //NOriginalMovers is the number of different Movers
```

```
        //to be copied.
        nOriginalMovers = 2;

        //Ncopies is the number of new copies
        //of each copied Mover in
        // the gene amplification method.
        nCopies = 2;

        //Velocity is among 0 and 10000,
        // it regulates  the velocity of the
        //movements in the display.
        velocity = 10000;

        /*To generate a repeatable sequence
        of random numbers, activate the next
        two lines:*/

        /* long seedValue = 400;
         r.setSeed( seedValue) ;  */
    }

    /*
    Movement at random,
    without any purpose,
    without any restriction. */
    public void initialChromosome(Worm worm1) {
        int n;
        for (int m = 1; m <= nMovers; m++) {
            //Select  which link to move:
            // Throw a number from 1 to nLinks

            n = r.nextInt(nLinks) + 1;
            // Symbolic initialization
            Move move1 = new Move(1, 1);
            Mover mover1 = wormsMutRate.new Mover(1, move1);
            // true initialization
            mover1.link = n;

            //Select  which direction to move on
            //Throw a number from 1 to 9
```

```
            //Move the tail first and then the others links.
            mover1.move = vectMoves[r.nextInt(9) + 1];

            worm1.mover[m] = mover1;
        }
    }

    // All worms are posited at the same initial row
    // in vertical position
    public void initialPosition(Graphics g, Worm worm[]) {
        int k, a, b;

        for (int W = 1; W <= nWorms; W++) {
            // We define an auxiliary worm
            Worm worm1;
            worm1 = wormsMutRate.new Worm();

            for (int l = 1; l <= nLinks; l++) {
                //Here is the head
                k = initialRow - width * nLinks;
                // The distance among worms is 40.
                // width is the width of  a link
                a = 40 * W;
                b = k + width * l;
                LinkAddress ad = wormsMutRate.new LinkAddress(a, b);
                worm1.linkAddress[l] = ad;

                // The first generation is initialized
                // by randomness, others by inheritance.
                // symbolic initialization
                /*  int c1=1,c2=1,c3=1;
if (randInitio == 1)
 {

  // a color is assigned to each link
  // but different than white
  c1 = r.nextInt(256);
  c2 = r.nextInt(256);
  c3 = r.nextInt(256);
  if (c1+c2+c3==765) c3=1;
```

```
  //colors are memorized.
  worm1.color[l][1]=c1;
  worm1.color[l][2]=c2;
  worm1.color[l][3]=c3;
 }

 // The color is prepared

  g.setColor( new Color(c1,c2,c3));
  // A Rectangle is defined by
    // the coordinates of the
    // superior right corner,
    // width and hight.
  // The link is painted
g.fillRect(a,b,width, width); */
                int m = (int) a / width;
                int n = (int) b / width;
                /* System.out.println("occupied = " +
      m + "  " + n );*/
                occupied[m][n] = true;
            }
            //  The know-how to move.
            if (randInitio == 1) {
                initialChromosome(worm1);
            }
            //  Memorization
            worm[W] = worm1;
        }
        /* System.out.println("asd = " +
worm[12].linkAddress[2].x );*/
    }

    // Parameters are set and worms are posited at the start
    public void setBeginning(Graphics g) {
        setParameters();
        // Initialization at random
        randInitio = 1;
        initialPosition(g, worm);
        randInitio = 0;
    }
```

```
// Beginning from the bottom
// with new genes
public void reInitialization(Graphics g, Worm worm[]) {
    randInitio = 1;
    occupationMatrix();
    initialPosition(g, worm);
}

// A worm is and must remain a unity.
public int disruptive(int m, Worm worm) {
    //  System.out.println("disruptive");
    // A move is disruptive if the target address
    // is farther than 1.5 units
    // from the neighboring links.
    int n;
    int x, y, x1, y1, x2, x3, y2, y3;
    double d1, d2;
    d2 = 0;

    //Reads the link to be moved
    n = worm.mover[m].link;
    //Reads the   address of the link;
    x = worm.linkAddress[n].x;
    y = worm.linkAddress[n].y;
    //The   move to be done is read
    Move move1 = worm.mover[m].move;

    //The target address is calculated
    x1 = x + width * move1.x;
    y1 = y - width * move1.y;

    //The addresses of the two neighboring links is read
    //The disruption is calculated
    if ((1 < n) & (n < nLinks)) {
        x2 = worm.linkAddress[n - 1].x;
        y2 = worm.linkAddress[n - 1].y;
        d1 = Math.pow((Math.pow((x2 - x1), 2)
                + Math.pow((y2 - y1), 2)), 0.5);
        x3 = worm.linkAddress[n + 1].x;
```

```
            y3 = worm.linkAddress[n + 1].y;

            d2 = Math.pow((Math.pow((x3 - x1), 2)
                    + Math.pow((y3 - y1), 2)), 0.5);

            // Here we decide whether or not
            // a move is disruptive
            if ((d1 > 1.4 * width) || (d2 > 1.4 * width)) {
                return 1;
            } else {
                return 0;
            }
        }

        if (n == 1) {
            x2 = worm.linkAddress[2].x;
            y2 = worm.linkAddress[2].y;
            d1 = Math.pow((Math.pow((x2 - x1), 2)
                    + Math.pow((y2 - y1), 2)), 0.5);
            if (d1 > 1.1 * width) {
                return 1;
            } else {
                return 0;
            }
        }

        if (n == nLinks) {
            x2 = worm.linkAddress[nLinks - 1].x;
            y2 = worm.linkAddress[nLinks - 1].y;
            d2 = Math.pow((Math.pow((x2 - x1), 2)
                    + Math.pow((y2 - y1), 2)), 0.5);
        }
        if (d2 > 1.1 * width) {
            return 1;
        } else {
            return 0;
        }
    }

    // The velocity of movement
```

```
// shall be observable  by a human.
public void delay(long l1) {
    while (System.nanoTime() < l1) {
        delay(l1);
    }
}

//Worms begin to move
public void moveOn(Graphics g, int m, Worm worm) {
    int n, i, j;

    //  System.out.println("moveOn");
    //Read  the link to be moved
    n = worm.mover[m].link;

    //To move a link:
    //the cell occupied by  the link is cleared,
    //the cell to be occupied is painted.
    //The occupied cell is cleared
    i = worm.linkAddress[n].x;
    j = worm.linkAddress[n].y;
    int k = (int) i / width;
    int l = (int) j / width;

    /* System.out.println("TTT" + occupied[k][l]);
 System.out.println("occupied = " + k +
   "  " + l );*/
    occupied[k][l] = false;
    // System.out.println(i+j+n);
    // the white color is prepared
    /* g.setColor(Color.white);
g.fillRect(i,j,width, width);*/

    //The move to be done is read
    Move move1 = worm.mover[m].move;

    i = i + width * move1.x;
    j = j - width * move1.y;
    worm.linkAddress[n].x = i;
    worm.linkAddress[n].y = j;
```

```
        //Movement=repainting in the new place;
        /* g.setColor( new Color(
    worm.color[n][1],worm.color[n][2],
    worm.color[n][3] ));
  g.fillRect(i,j,width, width);*/
        //Cell is occupied
        k = k + move1.x;
        l = l - move1.y;
        occupied[k][l] = true;

        //If movement is too fast, nothing can be seen.
        //To attune the velocity of the worm,
        //augment or diminish the delay as   pleased
        //change the upper bound of h
        long l1 = System.nanoTime();

        //change the upper bound of k
        for (int h = 1; h <= (10000 - velocity) * 20; h++) {
            delay(l1);
        }

    }

    // The instructions that are in the Chromosome,
    // are executed.
    public void ribosome(Graphics g, int m, Worm worm) {
        // System.out.println("ribosome");
        int n, i, j;

        Move move;
        int x, y, x1, y1;

        //Reads the link  to be moved
        // System.out.println("m = " + m);
        //The   move to be done is read
        move = worm.mover[m].move;
        //The   link to be moved is read
        n = worm.mover[m].link;
        //Reads the   address of the link;
```

```
        x = worm.linkAddress[n].x;
        y = worm.linkAddress[n].y;

        //The target address is calculated
        x1 = x + width * move.x;
        y1 = y - width * move.y;

        //Movement must remain in within barriers
        if ((x1 >= 1) & (y1 >= 1)
                & (x1 < 500) & (y1 < 400)) {

            //Is the target address occupied?
            // Is the move disruptive?
            i = (int) x1 / width;
            j = (int) y1 / width;

//  System.out.print("i= " + i + "  j= " + j);
//   System.out.println("occupied = " + occupied[i][j] );
            if ((occupied[i][j] == false)
                    & (disruptive(m, worm) == 0)) {
                moveOn(g, m, worm);
            } else;
        } else;
    }

    // Every worm is assigned a fitness
    //Slowest case reinforcement
    public void assessFitness() {
        //  System.out.println("Fitness");

        int max, man;

        // The nearer to the top, the fitter
        for (int W = 1; W <= nWorms; W++) {
            // We catch the farthest link
            max = 1;
            for (int m = 1; m <= nLinks; m++) {
                man = worm[W].linkAddress[m].y;
                if (man > max) {
                    max = man;
```

```
                }
            }
            fitness[W] = 1000 - max;
            // System.out.println( W + " " + fitness[W] );
        }

        // Now, we sort worms by fitness
        // and select the best four,
        // whose indexes are recorded in Order.
        int champ;
        for (int W = 1; W <= 4; W++) {
            champ = 1;
            for (int B = 1; B <= nWorms; B++) {
                if (fitness[B] > fitness[champ]) {
                    champ = B;
                }
            }
            order[W] = champ;
            fitness[champ] = 0;
        }
    }

    //Blind point mutations
    public void pointMutation() {
        int m, n;
        //Let us try out blind point mutations
        //in specific Movers,
        //in the  link  to be moved
        //and in the direction of movement.
        for (int W = 1; W <= nWorms; W++) {
            //Each offspring receives
            //numbPointMutation mutations
            //Select at random  which Mover to mutate:
            for (int k = 1; k <= numbPointMutations; k++) {
                //throw a number from 1 to NMovers
                m = r.nextInt(nMovers) + 1;
                //Select at random which link  to move:
                //throw a number from 1 to Nlinks.
                n = r.nextInt(nLinks) + 1;
                worm[W].mover[m].link = n;
```

```
                //Select at random which direction to move on.
                //Throw a number from 1 to 9
                worm[W].mover[m].move = vectMoves[r.nextInt(9) + 1];
            }
        }
    }

    // Some movers are overwritten with others
    // in within the same worm
    public void geneAmplification() {
        //  System.out.println("geneAmplification");
        int e, n, ee;

        Worm worm1;

        //All worms are scanned
        for (int W = 1; W <= nWorms; W++) {
            //wormTwo and worm coincide.
            worm1 = wormTwo[W];
            //Symbolic initialization
            initialChromosome(worm1);
            //nOriginalMovers are amplified
            //Select  which Mover to be amplified:
            for (int m = 1; m <= nOriginalMovers; m++) {
                //throw a number from 1 to NMovers
                e = r.nextInt(nMovers) + 1;

                //Symbolic initialization
                Move move1 = worm1.mover[e].move;
                n = worm1.mover[e].link;

                //The Mover is amplified
                //Select at random where to be copied:
                for (int k = 1; k <= nCopies; k++) {
                    //throw a number from 1 to nMovers.
                    ee = r.nextInt(nMovers) + 1;
                    worm[W].mover[ee].link = n;
                    worm[W].mover[ee].move = move1;
                }
            }
```

```
    }
}

// Old worms are erased.
public void death(Graphics g) {
    // The screen is cleared
    g.setColor(Color.white);
    g.fillRect(0, 0, xWin, yWin);
    // The worm array is overwritten with
    // the wormTwo array.
    worm = wormTwo;
}

// Point mutation and gene amplification
public void mutation() {
    // Point mutation dominates over
    // gene amplification.
    geneAmplification();
    pointMutation();
}

// A new link  is added to all worms.
public void oneMoreLink() {
    // System.out.print("oneMoreLink: ");
    int lim1;
    int n;
    //   The link  is added at the tail
    lim1 = nMovers + 1;
    nLinks = nLinks + 1;
    // System.out.println(nLinks);

    nMovers = factor * nLinks;
    Worm worm1;
    for (int W = 1; W <= nWorms; W++) {
        worm1 = worm[W];
        //New movers are filled in information
        for (int m = lim1; m <= nMovers; m++) {

            //Select at random which link to move:
            // Throw a number from 1 to nLinks
```

```
                n = r.nextInt(nLinks) + 1;
                // Symbolic initialization
                Move move1 = new Move(1, 1);
                Mover mover1 = wormsMutRate.new Mover(1, move1);
                // true initialization
                mover1.link = n;

                //Select  which direction to move on
                //Throw a number from 1 to 9
                //Move the tail first and then the others links.
                mover1.move = vectMoves[r.nextInt(9) + 1];

                worm1.mover[m] = mover1;

                worm[W].mover[m] = worm1.mover[m];
            }
        }
    }

    // Is there any progress?
    public int thereWasAdvance() {
        //  System.out.println("thereWasAdvance");
        //Our criterion for   advance is
        //that the head has surpassed
        //its original vertical position.
        int headPosition, advance;
        advance = 0;

        for (int W = 1; W <= nWorms; W++) {
            Worm worm1;

            worm1 = worm[W];
            headPosition = worm1.linkAddress[1].y;
            if (headPosition < initialRow - width * nLinks) {
                advance = 1;
            }
        }
        /* if (advance == 1)
       System.out.println("Advance");*/
        return advance;
```

```
    }

    // If at least one worm   advanced even a bit,
    // the next population is enlarged by one link:
    public void growth() {
        //  System.out.println("growth");
        if (thereWasAdvance() == 1) {
            oneMoreLink();
        } else;
    }

    //* Winners reproduce, others not
    public void reproduction(Graphics g) {
        //   Our reproductive scheme is as follows:
        //   50% offspring for  the champion,
        //   25% offspring for  the second,
        //   12.5% for   the third,
        //   12.5% for   the fourth.
        //   We place children in order.
        //   Children are clones of the fathers
        //   that are copied to the wormTwo array.

        //   Call InitialPosition(8)
        int champ = order[1];
        /* System.out.println("champ = "
     + champ);   */
        Worm worm1;
        worm1 = wormsMutRate.new Worm(worm[champ].mover,
                worm[champ].color, worm[champ].linkAddress);
        /* System.out.println("Address1 = "
+ worm1.linkAddress[2].x);  */

        int half = (int) nWorms / 2;
        int quarter = (int) nWorms / 4;
        // System.out.println("Champ = " + order[1] );

        for (int W = 1; W <= half; W++) {
            wormTwo[W] = worm1;
        }
```

```
        int limf = half + quarter;
        // System.out.println("limf = " + limf);

        champ = order[2];
        worm1 = wormsMutRate.new Worm(worm[champ].mover,
                worm[champ].color, worm[champ].linkAddress);

        for (int W = half + 1; W <= limf; W++) {
            wormTwo[W] = worm1;
        }

        int limi = limf;
        limf = limf + quarter / 2;

        champ = order[3];
        worm1 = wormsMutRate.new Worm(worm[champ].mover,
                worm[champ].color, worm[champ].linkAddress);
        for (int W = limi + 1; W <= limf; W++) {
            wormTwo[W] = worm1;
        }

        champ = order[4];
        worm1 = wormsMutRate.new Worm(worm[champ].mover,
                worm[champ].color, worm[champ].linkAddress);
        for (int W = limf + 1; W <= nWorms; W++) {
            wormTwo[W] = worm1;
        }
        // System.out.println("limf = " + limf);
        /*   System.out.println("Address1= " +
      worm[order[1]].linkAddress[2].x ); */
/*  System.out.println("Address2 = " +
     wormTwo[order[1]].linkAddress[2].x); */
        death(g);
        mutation();
        growth();
    }

    // Recombination is implemented
    public void recombination() {
        //Every individual is replaced by a recombinant
```

```
for (int i = 1; i <= nWorms; i++) {
    //  A  recombinant operator
    // is assembled at random
    int rec[] = new int[21];
    for (int j = 1; j <= nMovers; j++) {
        rec[j] = r.nextInt(2);
    }

    // Two worms are chosen at random
    int m = r.nextInt(nWorms) + 1;
    int n = r.nextInt(nWorms) + 1;

    Worm worm1;
    worm1 = wormsMutRate.new Worm(worm[m].mover,
            worm[m].color, worm[m].linkAddress);

    Worm worm2;
    worm2 = wormsMutRate.new Worm(worm[n].mover,
            worm[n].color, worm[n].linkAddress);

    //  Recombination is executed.
    //Symbolic initialization of the recombinant
    Worm worm3;
    worm3 = wormsMutRate.new Worm(worm[1].mover,
            worm[1].color, worm[1].linkAddress);

    for (int j = 1; j <= nMovers; j++) {

        if (rec[j] == 1) {
            worm3.mover = worm1.mover;
        } else {
            worm3.mover = worm2.mover;
        }
    }

//Recombinant is memorized
    wormTwo[i] = worm1;
}
//Old population is replaced by the recombinated one
```

```
    worm = wormTwo;
}


public void exoGenetics(Graphics g) {
    // Number of generations to be run

    // This is a life
    // l is the counter of days of life.
    // m is the number of the mover;
    // W is the index of the worm.
    for (int l = 1; l <= lifeSpan; l++) {
        for (int m = 1; m <= nMovers; m++) {
            for (int W = 1; W <= nWorms; W++) {
                //Auxiliary worm: worm1
                Worm worm1;

                worm1 = worm[W];
                /* System.out.println("W = " +W +
    " m = " + m + " l= " + l);*/
                ribosome(g, m, worm1);
                worm[W] = worm1;
            }
        }
    }
}

private static void mutationRedef() {
    int fit = fitness[1];
    newFitness = fit;
    deltaFitness = Math.abs(newFitness - oldFitness);
    oldFitness = newFitness;
    if (deltaFitness < 1) {
        numbPointMutations = numbPointMutations + 1;
        if (numbPointMutations > nLinks) {
            numbPointMutations = nLinks;
        }
    } else;
}
```

```
// Evolution is called to function
public void dynamics(Graphics g) {
    System.out.println("Warmed mutation");
    System.out.println("Initial length = " + nLinks);
    System.out.println("Link number + waiting time");
    int time1 = 0;
    int n1 = nLinks;
    int neededTime;
    for (int gen = 1; gen <= nGen; gen++) {
        if (nLinks > n1) {
            n1 = nLinks;
            neededTime = gen - time1;
            time1 = gen;
            System.out.println(nLinks + "\t "
                    + neededTime);
        }
        g.drawString("Gen = " + gen, 20, 20);
        // System.out.println("Gen = " + gen);
        exoGenetics(g);
        assessFitness();
        //Gene amplification is included.
        //Body enlargement is allowed.
        reproduction(g);
        recombination();
        //every 100 generations, stagnation is revised
        if (gen % 100 == 0) {
            mutationRedef();
        }
        reInitialization(g, worm);
    } //end of gen loop
}

// This is the main method.
@Override
public void paint(Graphics g) {
    //System.out.println("Applet");
    constructors();
    occupationMatrix();
    moveDef();
    setBeginning(g);
```

```
        dynamics(g);
    }
}//End of Program WormsVaryingMutRate, B83
```

84 ***Exercise*** *Run the code and test whether or not warming site mutagenesis accelerates the evolution rate.* *Answer*

85 ***Exercise*** *In the considered program, posit geneAmplification in the first place and then pointMutation. Reverse the order of execution to see whether or not the evolutionary rate is increased.*

86 ***Challenge*** *We have every reason to believe that the evolutionary rate is not a constant function of the mutation rate. So, some values of that rate should do better than others. Would you do the corresponding study?*

87 ***Challenge*** *A necessary condition for a worm to raise its body is to raise the tail. Moreover, the information that raises the tail is also good to raise any link. Hence, the evolutionary rate could depend on the magnitude of the gene amplification process. Prove else refute that.*

88 ***Challenge*** *What might happen if one ties the amplitude of the gene amplification process to the fitness, which is measured by the success in raising the tail?*

6.4 Conclusion

To accelerate evolution remains as a challenge.

Chapter 7

Where is arithmetic?

`Evolution develops the code for a binary adder.`

89 ***Motivation.***

The genome contains information to continue life and evolution given that life and evolution already exist. At present, we have no motivation to search for a piece of DNA that encodes for mathematical operations because these are not the sort of things that one expects to find in the genome. But given that the genome contains information to solve every kind of adaptive problems that amount to solving tremendous mathematical tasks, we must expect from evolution to be able to make mathematical calculations if only we demand it.

90 ***Purpose.***

Computers are associated in our culture primarily with text processing and mathematical operations. We already saw that evolution deals directly with text processing and with nothing else. Henceforth, we must prove that with text processing one can make arithmetical operations. More to the point, we must prove that text processing in the form of evolution is capable of developing the code for an adder. We will worry just about addition because the rest could be considered, at present, as more of the same.

7.1 Where is addition?

The native operations of evolution are those of text processing, which include copy-Paste and the possibility to compare strings to decide whether or not they are the same. We must show that addition of numbers arises as an emergent property of

copyPaste operations. To that aim, let us begin with a review of the addition of binary numbers.

91 ***Binary addition***

Numbers are used to count. The fundamental instruments for counting are the fingers of the hands. That is why we count in base 10. But we may count in any base. Say, numbers in base one are a direct representation of the elements that we count. The next list contains the equivalence of numbers in base ten translated to base one:

$1 \to |$
$2 \to ||$
$3 \to |||$
$4 \to ||||$
$5 \to |||||$
$6 \to ||||||$
$7 \to |||||||$
and so on.

In the case of base two, we have two symbols 0 and 1. The next list contains the equivalence of numbers in base ten translated to base two:

$0 \to 0$
$1 \to 1$
$2 \to 10$
$3 \to 11$
$4 \to 100$
$5 \to 101$
$6 \to 110$
$7 \to 111$
$8 \to 1000$
$9 \to 1001$
$10 \to 1010$
and so on.

The addition of numbers expresses the fact that the successor of a number j is the following in the list and results as the addition of one to the given number. The addition in base one reads as follows:

$$|||||| + |||||||||||| = ||||||||||||||||$$

The addition in base two is derived from the following table:

Addition		
+	0	1
0	0	1
1	1	10

Let us keep in mind that evolution deals with strings not with numbers, so evolution looks at each entry of this table as a string that is completely devoid of meaning, i.e. as a chain of chars taken from an alphabet. To be more explicit, let us rewrite that table as a list, which consists of four strings:

$0+0=0$
$1+0=1$
$0+1=1$
$1+1=10$

We can write this list just as a single string:

$0+0=0;\ 1+0=1;\ 0+1=1;\ 1+1=10$

This is a string with no meaning, i.e., any meaning is in your mind not in the table or in the list or in the string that encodes it.

Thus, the formulation of addition rules is a variation of the Weasel problem, which consists in guessing the string

$0+0=0;\ 1+0=1;\ 0+1=1;\ 1+1=10$

If one is given this string, one can synthesize it by whatever mean including evolution. So, this problem, one of retro-engineering in which the answer is known, is considered by us as solved. Our purpose in this and next chapter is more difficult: we assume that we do not know the answer and therefore we must find it. How?

Let us devise now an algorithm based on evolution to add binary numbers. Our algorithm must function as a calculator: it receives two binary numbers as input and must produce its binary sum as output. And this must be done with copyPaste (or derived procedures) and comparisons of strings. Nothing else is allowed.

If we succeed, we will have an example that illustrates the general premise that evolution is a software synthesizer to the full, including arithmetic facilities and all that.

92 ***Our codons***

The algorithm that encodes for a binary adder will result as the output of an evolutionary process among algorithms. Our algorithms consist in a list of copyPaste instructions. This innocent declaration is very important: we depart from the belief that copyPaste is a family of operations over strings that is sufficient to design algorithms for whatever function, if only that is possible. Equivalently: copyPaste is natural to word processing and if an algorithm exists, then it can also be implemented as a copyPaste procedure, i.e., as evolution. Additionally, we suppose that the output must be an elaboration of the inputs alone.

We shall encode an algorithm in a string, which is called **chromosome**. All chromosomes are evaluated for performance and we measure the degree up to which they correctly execute binary addition. A group that includes the best chromosomes is taken, mutated and recombined for the generation of a new population of chromosomes, which is then subjected to mutation. And so on until we reach a chromosome that encodes for a binary adder that perfectly adds binary numbers.

Now, the chromosomes are strings that are naturally divided into codons. A **codon** is a substring that contains a basic instruction.

What information is to be contained in a codon?

A codon will have information to copy a given substring to memory plus information to paste the content of memory into the output string.

We need three symbols to specify a copy operation. The first indicates the source string, the second the place in the source string from which we begin to copy while the third symbol will say how many chars we will copy.

'Paste from memory into output' is an operation that can be specified with just one symbol: the place where the insertion is to be done. The problem is that the output varies with time because it is in process of being formed. The remedy is to fix from the outset the length of all our strings. This causes no problem because the meaning of a number does not depend on the zeros at its left side.

Thus, the basic structure of a copyPaste codon is as follows:

```
class Codon
{
 int fromWhat; //from which string
 int fromWhere; //from which place  on
 int howManyChars;
 int toPlace; //in which place of the output
}
```

A program is just a concatenation of codons, of copyPaste instructions. Programs are listed in the chromosomes. A chromosome is an individual: we ignore every type of biochemistry.

We see in the next code how evolution looks for the code of a binary adder. It is intended to begin with addition of inputs one bit long and then it passes to the addition of numbers two bits long and so on. One can see in the output of the console, the number of generations needed to accomplish the task.

93 ***Our program*** *The code of a genetic algorithm that presumably develops the code for a binary adder. Only mutation is considered.*

```
package ejvol2v2p;

//Program BinaryAdder, B93
//The task of devising a binary adder
//is given to evolution but only mutation is considered.
//Mutation is implemented with copyPaste.
import java.util.Random;

public class BinaryAdder {

    public BinaryAdder() {

    }

    //BinaryAdder is a class, a prototype.
    //x is an instance of BinaryAdder
    //that will be used in the sequel.
    static BinaryAdder x = new BinaryAdder();

    //****Declaration of new types as inner classes**
    //Any class in Java may become a new type,
    //much as integer and string.
    //We need to define codons and chromosomes.
    //A codon is an instruction to
    //a) choose one  string among some options,
    //b) choose a place in that string,
    //c) copy a number of chars since that place on,
    //d) insert the copied substring into the output.
```

```
class Codon {

    int fromWhat; //from which string
    int fromWhere; //from which place  on
    int howManyChars;
    int toPlace; //in which place of the output

    //A codon can be defined given some integers
    Codon(int j, int k, int l, int m) {
        fromWhat = j;
        fromWhere = k;
        howManyChars = l;
        toPlace = m;
    }

    //A codon can be copied from another codon g
    Codon(Codon g) {
        fromWhat = g.fromWhat;
        fromWhere = g.fromWhere;
        howManyChars = g.howManyChars;
        toPlace = g.toPlace;
    }

}

//A chromosome is an array of codons.
//We deparrt from scholastics,
//in which chromosomes are strings.
//Chromosome is synonymous of individual
class Chromosome {

    int chromosomeLength;
    Codon c[];

    //Everything in Java must be initialized
    //otherwise, an nillpointer error will be generated.
    //Default initialization to zero
    public Chromosome(int n) {
        Codon cd = new Codon(0, 0, 0, 0);
        c = new Codon[n];
```

```
            for (int i = 0; i < n; i++) {
                c[i] = cd;
            }
        }

        //A chromosome can be
        //copied as a clone from another one.
        public Chromosome(Chromosome g) {
            chromosomeLength = g.chromosomeLength;

            for (int i = 0; i < chromosomeLength; i++) {
                c[i] = g.c[i];
            }
        }
    }


    //******Declaration of variables*********
    //  Strings that enconde numbers to be added
    private static  String input1 = "1111";
    private static  String input2 = "1111";
    private static  String output = "0";
    //  strings are converted to numbers
    private static  int nInput1, nInput2, nOutput;
    //  Error = difference among expected
    //  addition and output
    private static  int error = 0;
    //  Number of ciphers or bits in the inputs
    private static  int lengthInput;
    //  Number of ciphers in all of our strings
    private static  int stringLength = 10;

    //  We will have a population with
    //  a number of POPSIZE individuals
    private static  final int POPSIZE = 1000;
    private static  Chromosome Pop1[];
    private static  Chromosome Pop2[];

    // Individuals are sorted by fitness in Order[].
    private static  int Order[];
```

```
    private static  int gen;
    private static  boolean problem = true;

    //  Data contains the two input strings
    //  plus the output.
    private static final int LISTLENGTH = 3;
    private static String Data[] = new String[LISTLENGTH];

    // Turn on of the random generator
    private static  Random r = new Random();



    //********Declaration of methods*****************
    //  A string of stringLength characters is generated
    //  The alphabet is {0,1}.
    private static String generateStringNumber(int stringLength) {
        String individual = "";
        //An individual is assembled char by char
        for (int j = 0; j < stringLength; j++) {
            // We generate a random binary number as Character c
            int binary = r.nextInt(2);
            //System.out.println( "binary number = "+ binary );
            char c = (char) (binary + 48);
            //Character c is converted into  string s
            String s = Character.toString(c);
            //System.out.println( "binary string = "+ s );
            // s is concatenated to Pop1[i]
            individual = individual + s;
        }
        //System.out.println( individual);
        return individual;
    }

    private static void printCodon(Codon c) {
        System.out.println("c.fromWhat = " + c.fromWhat);
        System.out.println("c.fromWhere = " + c.fromWhere);
        System.out.println("c.howManyChars = " + c.howManyChars);
        System.out.println("c.toPlace = " + c.toPlace);
        System.out.println();
```

```
    }


    private static void printChromosome(Chromosome ch) {
        int n = ch.chromosomeLength;
        System.out.println();
        for(int i = 0; i < n; i++)
            printCodon(ch.c[i]);
    }

    private static Codon generateCodon() {
        //Default initialization
        Codon c = x.new Codon(0, 0, 0, 0);
        //True initialization
        //Warning: the output (index 2) also can be used
        //as source of information
        c.fromWhat = r.nextInt(LISTLENGTH);
        int n = stringLength;
        c.fromWhere = r.nextInt(n);
        int m = c.fromWhere;
        c.howManyChars = r.nextInt(n - m);
        c.toPlace = r.nextInt(n);
        //printCodon(c);
        return c;
    }

    //  An initial population   is set up.
    private static void initialization() {
        //Formal declaration of our arrays.
        Pop1 = new Chromosome[POPSIZE];
        Pop2 = new Chromosome[POPSIZE];

        for (int i = 0; i < POPSIZE; i++) {
            //n= Number of codons in   chromosome i
            int n = r.nextInt(9) +1;
            Chromosome chr = x.new Chromosome(n);
            Pop1[i] = chr;
            Pop1[i].chromosomeLength = n;
            Pop2[i] = chr;
            Pop2[i].chromosomeLength = n;
```

```
            //System.out.println("*");
            for (int j = 0; j < n; j++) {
                Pop1[i].c[j] = generateCodon();
                Pop2[i].c[j] = generateCodon();
            }
        }

        // Declarations and default initialization
        Order = new int[POPSIZE];
        for (int i = 0; i < POPSIZE; i++) {
            Order[i] = 0;
        }
    }

    // This method takes substring a
    // and posits it at place Start inside c
    private static String Insert(String a, int Start, String c) {
        //System.out.println("original c " + c);
        //System.out.println("original a " + a);
        //System.out.println("Start " + Start);
        String s1, s2;
        // We copy  the left part of c
        s1 = c.substring(0, Start);
        // We copy  the right part of c
        s2 = c.substring(Start);
        // We concatenate the left part of c  to
        // the right part of c
        String z = s1 + a + s2;
        // System.out.println("Inserted c " + z);
        return z;
    }

    // Reads the decimal number in  string a
    private static int decoder(String a) {
        int i = 0;
        int k;
        int n = a.length();
        for (int j = 0; j < n; j++) {
            char c = a.charAt(j);
            if (c == '1') {
```

```
                k = 1;
            } else {
                k = 0;
            }
            i = k + i * 2;
        }
        return i;
    }

    //  Posits zeroes at the left of the number
    //  until all sites are occupied
    private static String complete(String s) {
        int l = stringLength - s.length();
        for (int i = 1; i < l; i++) {
            s = '0' + s;
        }
        return s;
    }

    //  The problem is defined
    private static void defineProblem(int k) {
        input1 = generateStringNumber(k);
        input1 = complete(input1);
        input2 = generateStringNumber(k);
        input2 = complete(input2);
        output = "0";
        output = complete(output);
        Data[0] = input1;
        Data[1] = input2;
        nInput1 = decoder(input1);
        nInput2 = decoder(input2);
        /*System.out.println("input1= " + input1  );
  System.out.println("input2= " + input2 );
  System.out.println("output= " + output );*/
    }

    //  Each individual is tested for how well it adds
    private static int fitness(Chromosome chr) {
        //Checking out of the algorithm with
        //some random inputs.
```

```
        error = 0;
        for (int i = 0; i < 20 * lengthInput; i++) {
            defineProblem(lengthInput);
            for (int h = 0; h < chr.chromosomeLength; h++) {
                int j = chr.c[h].fromWhat;
                int k = chr.c[h].fromWhere;
                int l = chr.c[h].howManyChars;
                int m = chr.c[h].toPlace;
                String fromWhat = Data[j];
                //printCodon(chr.c[h]);
                String a = fromWhat.substring(k, k + l);
                output = Insert(a, m, output);
                int v = output.length();
                //System.out.println("output.length()= "+v);
                //The last lengthString chars of output are copied
                output = output.substring(v + 1 - stringLength);
                //System.out.println("Output = " + output);
                //The output is a source of modifications:
                Data[2] = output;
            }
            //System.out.println();
            nOutput = decoder(output);
            int myError = Math.abs(nInput1 + nInput2 - nOutput);
            System.out.println("Input1 = " + nInput1
                    + " Input2 = " + nInput2
            + " Output = " + nOutput
             + " Error =  " + myError);
            //The error is the difference between the correct
            //addition and that proposed by the individual.
            error = error + myError;

        }
        System.out.println("Gen = " + gen + "  TOTAL Error = " + error
        return error;
    }

    // Individuals are sorted by fitness
    //  The fittest is the first, with minimum error
    private static void Sorting() {
        int Champ;
```

```
int Fitness[];
//Fitnessc[] is a copy of Fitness[]
//used as workbench
int Fitnessc[];
Fitness = new int[POPSIZE];
Fitnessc = new int[POPSIZE];

for (int i = 0; i < POPSIZE; i++) {
    Fitness[i] = fitness(Pop1[i]);
    Fitnessc[i] = Fitness[i];
}

//We sort individuals by fitness
//Fitness 0 means no errors
for (int i = 0; i < POPSIZE; i++) {
    Champ = 0;
    for (int j = 0; j < POPSIZE; j++) {
        if (Fitnessc[j] < Fitnessc[Champ]) {
            Champ = j;
        }
    }

    //The array Order classifies individuals by  fitness,
    //the fittest is number zero
    Order[i] = Champ;
    Fitnessc[Champ] = 1000000;
}
//Print job to test sorting procedure
/*
for (int i = 0; i < POPSIZE; i++) {
    System.out.println("Fitness  = " + Fitness[Order[i]]);
}

System.out.println();
*/
if (Fitness[Order[0]] == 0) //if error = 0
{
    problem = false;
    System.out.println("Task ended by generation = " + gen);
    System.out.println("Number of trials = " + gen * POPSIZE);
```

```
            System.out.println("Winner Chromosome :");
            printChromosome(Pop1[Order[0]]);
            fitness(Pop1[Order[0]]);
        }

    }

    /*

//The new generation is made
private  static void newGeneration()
{
    int   i=0;
    //The fittest individuals are reproduced
    // numbCopies times
    int numbCopies = 10;
    int   k=0;  //rank, 0 is the highest

    while (i < popSize)
    {
        int counter = numbCopies;
        //Rank k is reproduced numbCopies times
        while ((counter > 0) & (i < popSize))
        {
            //System.out.println(i + "" + l);
            Individual2[i] = Individual[Order[k]];
            //System.out.println(i + " " + counter + " " + k);
            i++;
            counter= counter-1;
        }
        k++;
    }

    for(int j = 0; j < popSize; j++)
    {
        Individual[j] = Individual2[j];
        //System.out.println(j + " " + Individual[j]);
    }

}
```

```
*/
// The new generation is made
private static void newGeneration() {
    int i = 0;
    //The fittest individuals are reproduced
    // numbCopies times
    int numbCopies = 10;
    int k = 0;  //rank, 0 is the highest

    while (i < POPSIZE) {
        int counter = numbCopies;
        while ((counter > 0) & (i < POPSIZE)) {
            //System.out.println(i + "" + l);
            Pop2[i] = Pop1[Order[k]];
            //System.out.println(i + " " + counter + " " + k);
            i++;
            counter = counter - 1;
        }
        k++;
    }

    for (int j = 0; j < POPSIZE; j++) {
        Pop1[j] = Pop2[j];
        /*System.out.println( j+ " length at newGen " +
                                Pop1[j].length);
         */
    }
}


// Chromosome g is mutated at a random codon.
private static Chromosome change(Chromosome g) {
    int l = r.nextInt(g.chromosomeLength);
    g.c[l] = generateCodon();
    return g;
}

//  All individuals are mutated
private static void Mutation() {
    for (int i = 0; i < POPSIZE; i++) {
```

```
            Pop1[i] = change(Pop1[i]);
        }
    }

    // Given a goal, it is guessed by evolution
    private static void evolSolution(int n) {
        //Individuals are sorted by fitness
        Sorting();
        //The new generation is produced
        newGeneration();
        //The new population is subjected to mutation
        Mutation();
        gen++;
    }

    public static void main(String[] args) {
        /*int l = decoder("100111");
System.out.println(l);*/

        //r.setSeed(Int) causes the same sequence of
        //random numbers to be  always generated.
        //r.setSeed(1);
        int seed = 12345;
        r.setSeed(seed);
        System.out.println("Seed = " + seed);

        for (int k = 1; k <= 1; k++) {
            // k is the length of the input, the number of bits
            lengthInput = k;
            System.out.println("Length of inputs = " + k);
            gen = 1;
            initialization();
            while (problem) {
                evolSolution(lengthInput);
            }
            problem = true;

            System.out.println();
        }
    }
```

```
}//End of program BinaryAdder, B93
```

94 ***Exercise*** *Run the program and play with the code.*

95 ***Recombination***

Our previous simulation shows that when variability is granted by mutation, reproduction of the fittest as a pure strategy has nothing to do with reliable solutions to the problem of software design. If a pure strategy does not do it, maybe a mixed one solves the problem. So, let us add another item. But, which?

In our case, the answer can be solved at once with the help of genetics: we are ignoring **recombination**, which is an operation in which new chromosomes are formed by concatenating or joining together various strokes from diverse chromosomes.

Why is it expected that recombination will solve our troubles?

In short, it is because ordinary individuals that have passed a process of selection may contain pieces of information that may solve some aspects of the problem, and we may get a full fledge solution if we join those diverse pieces.

A geometric version of this insight is as follows:

An evolutionary process is similar to getting the highest point in a landscape. The problem is that nobody knows in which mountain one will find that place. The solution is to try here and there to see what one can achieve. Easy problems are similar to climbing a landscape with just one pick. A difficult problem is similar to exploring a landscape with too many mounts most of which are covered by clouds. So, one may learn to climb over a given mount but most mounts are too low, so one must escape low mounts to access the possibility to climb very high mounts. Recombination makes that for us: it produces abrupt changes as the result of the concatenation of diverse portions of information.

96 ***Exercise*** *Modify previous program B93 to include recombination. Does recombination converts a useless program based on mutation into a useful one?* *Answer*

97 ***Probabilistic algorithms***

We used mutation and recombination to create variability that was next assessed for fitting a certain function, which in present case was to simulate a binary adder. Our method of assessing the performance of our algorithms as binary adders was to generate some binary inputs at random and to report the error between expected and observed. We cannot guarantee that we generated all possible inputs,

thus if our program qualifies a given algorithm as a binary adder, we cannot guarantee that this is indeed so. But we have a good confidence that our binary adders are worth trust. One very reason is that a solution is not immediately found.

A program as ours is called a **probabilistic algorithm**. By contrast, those programs that perform a test over all possible inputs fulfill the normal desire and have no specific name. When the verifying procedure is probabilistic, not all options are checked out but only a random sample.

98 ***Perfect vs. frustrated computation***

We say that a procedure or method implements an **approximating scheme** when the recurrent application of the procedure produces a sequence of outputs that converges to the correct solution. Eventually, the recurrence can be truncated up to certain accuracy.

We all are used to demand perfect solutions to problems, but that directive is not applicable to the study of evolution, be it for industrial or academic purposes. In our study, it is more suitable to demand a certain degree of approximation without requiring the convergence of algorithms to the perfect solution. Thus, while an **approximating scheme** guarantees that the correct solution is arrived at if sufficient computing time is granted, in a **frustrating scheme** one comes near to the answer but not necessarily arrives at it.

99 ***Exercise*** *Assess the possibilities of our programs as frustrated schemes to the problem of devising a binary adder. To that aim, instead of demanded total error equal to zero, equate it to say, 2 or 5 or a function of the number of testing trials.* *Answer*

100 ***Research*** *Must we require from natural evolution as a software developer to design algorithms that function perfectly over every input? Actually, how is life? Does nature use an approximating scheme or a frustrated one?*

7.2 Beyond copyPaste

We have argued that evolution and word processing is the very same thing. Moreover, we have reduced word processing to copyPaste. So, we concluded that copyPaste shall be enough to solve every kind of optimization problems. Nevertheless, our previous program shows us that we are completely wrong: copyPaste (in the form of mutation + recombination) is unable to add two bits (two binary numbers). Therefore, evolution cannot be reduced to copyPaste. What do we lack?

We lack a correct abstraction of the definition of the problem. In fact, the problem begins with the following table:

$0+0=0$;
$1+0=1$;
$0+1=1$;
$1+1=10$,

Let us rewrite this table in an algorithmic form:

```
If␣input1␣=␣0␣and␣input2␣=␣0␣then␣output␣=␣0;
If␣input1␣=␣0␣and␣input2␣=␣1␣then␣output␣=␣1;
If␣input1␣=␣1␣and␣input2␣=␣1␣then␣output␣=␣1;
If␣input1␣=␣1␣and␣input2␣=␣1␣then␣output␣=␣10;
```

We see that this algorithm is filled in conditional clauses. Are these conditional reduced to copyPaste?

101 ***Exercise*** *Agree or disagree with the following opinions of the Author that will orient his further work:*

1. *Conditionals cannot be reduced to copyPaste.*
2. *CopyPaste + conditionals are a complete abstraction of evolution.*
3. *Conditionals are part of nature through molecular pattern recognition: this happens each time a DNA strand sticks to another because of complementary pairing or when a DNA-protein complex recognizes a pattern in a DNA strand.*

7.3 Binary adder by human design

To learn how the new reduction of evolution operates, let us develop the code for a binary adder on our own. We present two versions:

The first implements the method of addition that we all know and which is characterized by its serial performance, i.e., one instruction after the other.

The second is an implementation of the directive that evolution can proceed in parallel at many places at the same time.

102 ***Serial solution***

Our human way of making a binary addition is based on the table of the addition given by

$0+0=0$;
$1+0=1$;
$0+1=1$;
$1+1=10$, this causes a carrier of one.
$1+1+1=11$, this also causes a carrier of one.
Next, we apply this table as follows:

```
Carrier    1011101
number 1   10101011
number 2       1101
           ---------
Addition   10111000
```

Let us verify our answer. The first number is in base 10:
$10101011 \to ((((((1\times 2+0)\times 2+1)\times 2+0)\times 2+1)\times 2+0)\times 2+1)\times 2+1=171$
The second number in base ten is: $1101 \to 13$
This two numbers add up to 184, which must be equal to the output. Let us verify it:
$10111000 \to 184$. OK.

103 ***Exercise*** *Design a program that executes binary addition over two inputs following the algorithm outlined by the previous example. Because we are thinking of evolution, the program must encode numbers as strings. Dear Reader, this is the first program that you are asked to design ab initio, so let me to express you that my heart warms at your effort and triumph, which indeed is a great one.* Answer

104 ***Parallel solution***

While a serial computation consists in the execution of instructions one after another, as in the previous implementation of the binary adder, evolution also can execute operations at the same time in diverse places. Say, duplication of a DNA strand can be originated at diverse sites along the strand and the possibility of including mutations appears also distributed. Since mutations correspond to computations, we see that natural evolution uses parallel computing. By the same token, parallel computing runs in diverse chromosomes, diverse organelles, diverse individuals, and diverse populations.

Let us learn to add in parallel. The next example, in which we repeat a binary addition made above, illustrates the technique which is a vertical application of the rules for binary sum:

$0+0=00$;
$1+0=01$;
$0+1=01$;
$1+1=10$.

In the first output of the next calculation, vertical sum, we see the application in vertical form of these rules. In the contractions, we apply the same rules but take one bit from the left side and the other from the right side of the superior line. The final contraction just changes 01 by 1 and 00 by 0. One discovers that the final contraction must be applied when there is no result of the form 10, equivalently, when the algorithm produces no change over the computation.

```
number 1:              1   0   1   0   1   0   1   1

number 2:                          1   1   0   1
                    ------------------------------------
Vertical sum          01  00  01  00  10  01  01  10
Contraction 1       0   01  00  01  01  00  01  10  00
Contraction 2         00  01  00  01  01  00  10  00  00
Contraction 3       0   00  01  00  01  01  01  00  00  00
Contraction 4         0   00  01  00  01  01  01  00  00  00
                    ------------------------------------
Final contraction     0   0   1   0   1   1   1   0   0   0
```

To implement this procedure, we will use a non standard form of encoding the inputs: we intercalate them using a separator. In that way, we can use replace functions to execute the vertical sum. Our encoding takes one char from the first input, another from the second and intercalates them positing a separator at each step. In that way, the two inputs of the shown addition looks like this:

Step one: refilling in zeros at the left of the shorter input:

```
number 1:          1   0   1   0   1   0   1   1

number 2:          0   0   0   0   1   1   0   1
```

Step two: intercalation:

```
Input              10Y00Y10Y00Y11Y01Y10Y11
```

Let us notice that we are introducing a separator, a 'Y', a symbol that is not a binary one. In certain sense, we are violating a previous accepted by us rule that says that the output is an elaboration of the inputs alone. This fact prompts the idea that a problem can be encoded in too many diverse ways and that the form of encoding could be crucial for finding the solution.

We will implement this algorithm in various steps and will show that our parallel binary adder generates an **approximating scheme**, i.e., it finds the correct solution for every input if enough time and memory are granted. The job consists in a program and an exercise.

105 *Parallel binary adders with one and two contractions*

We compare in the next program the role of one and three contractions in the performance of an algorithm that simulates a parallel binary adder. The output of the algorithm produces four columns. The first columns is the number of bits or ciphers k of the binary numbers that are tried out as inputs to our algorithm. The program takes two inputs of length k, which are passed under the algorithm with one contraction and the output is compared with the correct result given by the binary addition. This is repeated a certain number of times and the total error, $error1$ is reported in column 2. Column 3 does the same but the error, $error3$ is computed when three contractions have been included. Column 4 offers the relation $error3/error1$ that measures the relative improvement due to the inclusion of three contractions instead of one. The program follows:

```
//Program parallelBinaryAdder, B105
//The effect of one and three contractions
//is compared.
package ejvol2v2p;
```

```
import java.util.Random;

public class ParallelBinaryAdder {
    //  Strings that encode numbers to be added

    static String input1, input2;
    static String output1, output2;
    //  strings are converted to numbers
    static int nInput1, nInput2;
    static int nOutput1, nOutput2;
    //  Error = difference among expected
    //  addition and output
    static double error1, error3;
    //  Number of ciphers or bits in the inputs

    static int stringLength;
    static int lengthInput;

    // Turn on of the random generator
    static Random r = new Random();

    private static final long serialVersionUID = 1L;

    //  A string of stringLength characters is generated
    //  The alphabet is {0,1}.
    private static String generateStringNumber(int stringLength) {
        String individual = "";
        //An individual is assembled char by char
        for (int j = 1; j <= stringLength; j++) {
            // We generate a random binary number as Character c
            int binary = r.nextInt(2);
            //System.out.println( "binary number = "+ binary );
            char c = (char) (binary + 48);
            //Character c is converted into  string s
            String s = Character.toString(c);
            //System.out.println( "binary string = "+ s );
            // s is concatenated to Pop1[i]
            individual = individual + s;
        }
        //System.out.println( individual);
```

```
        return individual;
    }

    // Reads the decimal number in  string a
    private static int decoder(String a) {
        int i = 0;
        int k = 0;
        int n = a.length();
        for (int j = 0; j < n; j++) {
            char c = a.charAt(j);
            if (c == '1') {
                k = 1;
            } else {
                k = 0;
            }
            i = k + i * 2;
        }
        return i;
    }

    //  Posits zeros at the left of the number
    //  until all sites are occupied
    private static String complete(String s) {
        int j = stringLength - s.length();
        for (int i = 1; i <= j; i++) {
            s = '0' + s;
        }
        return s;
    }

    //  The problem is defined
    private static void defineProblem(int k) {
        input1 = generateStringNumber(k);
        input1 = complete(input1);
        input2 = generateStringNumber(k);
        input2 = complete(input2);
        nInput1 = decoder(input1);
        nInput2 = decoder(input2);
        /*System.out.println("input1= " + input1  );
  System.out.println("input2= " + input2 );
```

```
     */
}

//Intercalates input 1 with input2
//char by char
private static String intercalation() {
    int n = input1.length();
    int m = input2.length();
    stringLength = Math.max(n, m);
    input1 = complete(input1);
    input2 = complete(input2);
    String output = "";
    for (int i = 0; i < stringLength; i++) {
        output = output + input1.charAt(i) + input2.charAt(i) + "Y";
    }
    return output;
}

//  Only one contraction is implemented
private static String solution1() {
    String output1 = intercalation();

    //Vertical sum
    String output2 = output1.replaceAll("10", "01");

    String output3 = output2.replaceAll("11", "10");

    //Contraction1
    String output4 = output3.replaceAll("1Y0", "0Y1");

    String output5 = output4.replaceAll("1Y1", "1Y0");

    //A zero is inserted
    int n = output5.length();
    String output6 = output5.substring(0, n - 2)
            + "0" + output5.substring(n - 2);

    //Final contraction
    String output7 = output6.replaceAll("01", "1");
```

```
        String output8 = output7.replaceAll("00", "0");

        String output9 = output8.replaceAll("10", "1");

        String output10 = output9.replaceAll("Y", "");

        /* System.out.println("output1= " + output1 );
System.out.println("output2= " + output2 );
System.out.println("output3= " + output3 );
System.out.println("output4= " + output4 );
System.out.println("output5= " + output5 );
System.out.println("output6= " + output6 );
System.out.println("output7= " + output7 );
System.out.println("output8= " + output8 );
System.out.println("output9= " + output9 );
System.out.println("output10= " + output10 );
         */
        return output10;
    }

    //  Three contractions are implemented
    private static String solution2() {
        String output = intercalation();
        //Vertical sum
        output = output.replaceAll("10", "01");
        output = output.replaceAll("11", "10");

        int n = output.length();
        //Contraction1
        output = output.replaceAll("0Y0", "M00");
        output = output.replaceAll("1Y0", "M01");
        output = output.replaceAll("0Y1", "M01");
        output = output.replaceAll("1Y1", "M10");

        //A "M0" is inserted
        output = output.substring(0, n - 2)
                + "M0" + output.substring(n - 2);
        output = output.replaceAll("Y", "");
        //System.out.println("output1= " + output);
        //Contraction2
```

```
        output = output.replaceAll("0M0", "Y00");
        output = output.replaceAll("1M0", "Y01");
        output = output.replaceAll("0M1", "Y01");
        output = output.replaceAll("1M1", "Y10");
        //A "Y0" is inserted
        output = output.substring(0, n)
                + "Y0" + output.substring(n);

        // System.out.println("output2= " + output);
        //Contraction3
        output = output.replaceAll("0Y0", "M00");
        output = output.replaceAll("1Y0", "M01");
        output = output.replaceAll("0Y1", "M01");
        output = output.replaceAll("1Y1", "M10");
        output = output.replaceAll("Y", "");
        n = output.length();
        //A "M0" is inserted
        output = output.substring(0, n - 1)
                + "M0" + output.substring(n - 1);

        //System.out.println("output3= " + output);
        //Final contraction and housekeeping
        output = output.replaceAll("01", "1");
        output = output.replaceAll("00", "0");
        output = output.replaceAll("10", "1");
        output = output.replaceAll("Y", "");
        output = output.replaceAll("M", "");

        return output;
    }

    // Two algorithms are tested for how well they add
    static public void workSolutions(int lengthInput) {
        //Checking out of  algorithms with
        //some random inputs.
        error1 = 0;
        error3 = 0;

        for (int i = 0; i < 10 * lengthInput; i++) {
            defineProblem(lengthInput);
```

```
            output1 = solution1();
            nOutput1 = decoder(output1);
            //The error is the difference between the correct
            //addition and that proposed by the individual.
            error1 = error1 + Math.abs(nInput1 + nInput2 - nOutput1);
            output2 = solution2();
            nOutput2 = decoder(output2);
            error3 = error3 + Math.abs(nInput1 + nInput2 - nOutput2);
            /*
            if (i == 1) {
                System.out.println("input1= " + input1);
                System.out.println("input2= " + input2);
                System.out.println("output1= " + output1);
                System.out.println("output2= " + output2);
                System.out.println("error1 = " + error1);
                System.out.println("error3 = " + error3);
            }
            */
        }
    }


    public static void test() {
        System.out.println("Beginning of test");
        input1 = "1111111111";
        input2 = "1010101010";
        //input1 = "1001111111";
        //input2 = "1010101010";

        System.out.println("input1= " + input1);
        System.out.println("input2= " + input2);
        output2 = solution2();
        System.out.println("output= " + output2);
        //Strings are transformed to int
        nInput1 = decoder(input1);
        nInput2 = decoder(input2);
        nOutput2 = decoder(output2);
        //The error is the difference between the correct
        //addition and that proposed by the algorithm.
        int error = Math.abs(nInput1 + nInput2 - nOutput2);
```

```
        System.out.println("Error= " + error);
        System.out.println("End of test");
        System.out.println();
    }


    public static void main(String[] args) {
        /*int l = decoder("100111");
 System.out.println(l);*/
        test();
        //r.setSeed(Int) causes the same sequence of
        //random numbers to be  always be generated.
        //r.setSeed(1);
        int seed = 54321;
        r.setSeed(seed);
        System.out.println("Seed = " + seed);
        System.out.print("inputLength \t  error1 \t error3 \t ");
        System.out.println("relation");
        for (int k = 1; k <= 100; k++) {
            // k is the number of bits of inputs
            lengthInput = k;
            workSolutions(lengthInput);
            double errorRelation = error3 / error1;
            System.out.print(k + "\t");
            System.out.print(error1 + "\t");
            System.out.print(error3 + "\t");
            System.out.println(errorRelation);
        }
    }
}//Program parallelBinaryAdder, B105
```

106 ***Exercise*** *Run the code and analyze the behavior of the relation error*3/*error*1.
Answer

107 ***Refined code***

Let us modify the code to study the relation *error*4/*error*1 where *error*4 is the error made when 4 pairs contractions are included. The algorithm follows:

```
//Program FullParallelBinaryAdder, B107
```

```
//The effect of k contractions
//is studied and compared to that of one
//single contraction.


package ejvol2v2p;


import java.util.Random;


public class FullParallelBinaryAdder {
    //The algorithm takes pairs of numbers
    //with numberOfBits ciphers.

    static int numberOfPairsOfContractions = 15;
    //Strings that encode numbers to be added
    static String input1, input2;
    static String output1, output2;
    //strings are converted to numbers
    static int nInput1, nInput2;
    static double nOutput1, nOutput2;
    //Error = difference among expected
    //addition and output
    static double error1, error4;
    //Number of ciphers or bits in the inputs

    static int stringLength;
    static int lengthInput;
    static int n;
    static boolean print = true;

    //Turn on of the random generator
    static Random r = new Random();

    private static final long serialVersionUID = 1L;

    //A string of stringLength characters is generated
    //The alphabet is {0,1}.
    private static String generateStringNumber(int stringLength) {
```

```
        String individual = "";
        //An individual is assembled char by char
        for (int j = 1; j <= stringLength; j++) {
            // We generate a random binary number as Character c
            int binary = r.nextInt(2);
            //System.out.println( "binary number = "+ binary );
            char c = (char) (binary + 48);
            //Character c is converted into  string s
            String s = Character.toString(c);
            //System.out.println( "binary string = "+ s );
            // s is concatenated to Pop1[i]
            individual = individual + s;
        }
        //System.out.println( individual);
        return individual;
    }

    //Reads the decimal number in  string a
    private static int decoder(String a) {
        int i = 0;
        int k ;
        int nn = a.length();
        for (int j = 0; j < nn; j++) {
            char c = a.charAt(j);
            if (c == '1') {
                k = 1;
            } else {
                k = 0;
            }
            i = k + i * 2;
        }
        return i;
    }

    //Posits zeros at the left of the number
    //until all sites are occupied
    private static String complete(String s) {
        int j = stringLength - s.length();
        for (int i = 1; i <= j; i++) {
            s = '0' + s;
```

```
        }
        return s;
    }

    //The problem is defined
    private static void defineProblem(int k) {
        input1 = generateStringNumber(k);
        input1 = complete(input1);
        input2 = generateStringNumber(k);
        input2 = complete(input2);
        nInput1 = decoder(input1);
        nInput2 = decoder(input2);
        if (print)
        {
            System.out.println("\ninput1= " + input1 + " -> " + nInput1
            System.out.println("input2= " + input2 + " -> " + nInput2)
        }


    }

    //Intercalates input 1 with input2
    //char by char and posits a separetor "Y"
    private static String intercalation() {
        int nn = input1.length();
        int m = input2.length();
        stringLength = Math.max(nn, m);
        input1 = complete(input1);
        input2 = complete(input2);
        String output = "";
        for (int i = 0; i < stringLength; i++) {
            output = output + input1.charAt(i) + input2.charAt(i) + "Y"
        }
        return output;
    }

    private static String verticalSum() {
        String outputI = intercalation();
        String outputR = outputI.replaceAll("10", "01");//0
        String output3 = outputR.replaceAll("11", "10");
```

```
        return output3;
    }

    private static String contraction1(String output) {
        output = output.replaceAll("0Y0", "M00");//2
        output = output.replaceAll("1Y0", "M01");
        output = output.replaceAll("0Y1", "M01");
        output = output.replaceAll("1Y1", "M10");
        n = output.length();
        //A "M0" is inserted
        output = output.substring(0, n - 2)
                + "M0" + output.substring(n - 2);//6
        output = output.replaceAll("Y", ""); //7
        return output;
    }

    private static String contraction2(String output) {
        output = output.replaceAll("0M0", "Y00");//8
        output = output.replaceAll("1M0", "Y01");
        output = output.replaceAll("0M1", "Y01");
        output = output.replaceAll("1M1", "Y10");
        n = output.length();
        //A "Y0" is inserted
        output = output.substring(0, n-1)
                + "Y0" + output.substring(n-1);//12
        return output;
    }

    private static String contraction(String output) {
        output = output.replaceAll("0Y0", "M00");//13
        output = output.replaceAll("1Y0", "M01");
        output = output.replaceAll("0Y1", "M01");
        output = output.replaceAll("1Y1", "M10");//16

        n = output.length();
        //A "M0" is inserted
        output = output.substring(0, n - 1)
                + "M0" + output.substring(n - 1);
        output = output.replaceAll("Y", ""); //18
        output = contraction2(output);
```

```
        return output;
    }

    private static String finalContraction(String output) {
        output = output.replaceAll("01", "1");
        output = output.replaceAll("00", "0");
        //In case of uncooked solutions
        output = output.replaceAll("10", "1");
        output = output.replaceAll("Y", "");
        output = output.replaceAll("M", "");
        return output;
    }

    //Only one contraction is implemented
    private static String solution1() {
        String output = verticalSum();
        output = contraction1(output);
        output = finalContraction(output);
        return output;
    }

    //Contractions to the full
    private static String solution2() {
        String output = verticalSum();
        System.out.println("verticalSum = " + output);
        output = contraction1(output);
        System.out.println("contraction1 = " + output);
        output = contraction2(output);
        System.out.println("contraction2 = " + output);

        for (int l = 1; l < numberOfPairsOfContractions; l++) {
            output = contraction(output);
            System.out.println("contraction = " + output);
        }

        output = finalContraction(output);
            System.out.println("Final cntrc = " + output);
        return output;
    }
```

```
    //Two algorithms are tested for how well they add
    static public void workSolutions(int lengthInput) {
        //Checking out of  algorithms with
        //some random inputs.
        error1 = 0;
        error4 = 0;

        for (int i = 0; i < 10 * lengthInput; i++) {
            defineProblem(lengthInput);
            output1 = solution1(); //one contraction
            nOutput1 = decoder(output1);
            //The error is the difference between the correct
            //addition and that proposed by the individual.
            error1 = error1 + Math.abs(nInput1 + nInput2 - nOutput1);
            output2 = solution2();//as many contractions as desired
            nOutput2 = decoder(output2);

            error4 = error4 + Math.abs(nInput1 + nInput2 - nOutput2);
            if (print)
            {
                System.out.println("output2 = " + output2
                        + " -> " + nOutput2 );

                System.out.println("error4 = " + error4);
            }
        }
    }

    public static void main(String[] args) {
        /*int l = decoder("100010");
            System.out.println(l); */

        //r.setSeed(Int) causes the same sequence of
        //random numbers to be  always be generated.
        //r.setSeed(1);
        int seed = 54321;
        r.setSeed(seed);
        System.out.println("Seed = " + seed);
        System.out.println("inputLength \t  "
                + "error1 \t " + " errork \t relation");
```

```
        for (int k = 1; k <= 10; k++) {
            // k is the number of places or bits of inputs
            lengthInput = k;
            workSolutions(lengthInput);
            double errorRelation = error4 / error1;
            System.out.print(k + "\t");
            System.out.print(error1 + "\t");
            System.out.print(error4 + "\t");
            System.out.println(errorRelation);
        }
    }
}//End of Program FullParallelBinaryAdder, B107
```

108 ***Exercise*** *Run the code. Decide whether or not we have an approximating scheme, i.e., if we can make certain that the more contractions are included the more exact gets the algorithm.* *Answer*

109 *Can logic be dispensed for?*

It is very interesting that our serial adder is based in a method that crucially depends on a set of nested ifs that form a ladder, which are at the heart of logical analysis. By contrast, the parallel version lacks such logical tester. This prompts a question: Is logic dispensable?

The answer is a rotund not. Indeed, the parallel version has many hidden ifs: each time a recognizing operation is done, one is executing and if operation. This happens in instructions of the form 'replace a by b inside c', which includes a comparison of a with all substrings of c.

Let us underscore again that pattern recognition is endemic to evolution through DNA matching of complementary strands that is extended to DNA - protein pairing. Thus, logic is also endemic to evolution and we can employ it safely in our simulations.

110 *Time is scare*

Modern computing technology is so fast that it marvels everyone. But our recurrent failure in trying to accelerate evolution has proved that time continues to be a very scarce resource.

Let us point out that the wasted time is proportional to the number of iterations that must be executed serially in order to accomplish a given task. To fix ideas, let us think of our two binary adders, one serial the other parallel. In the serial version, we must execute in tandem as many sums as there are bits in the larger input. But in the parallel version, the number of operations that must be executed in tandem is directly related to the number of contractions that must be done to terminate the program.

Our program FullParallelBinaryAdder, B107, shows that with some 15 pairs of contractions, one can add numbers with a length larger than 200.

This results allows us to suspect that the wasted time to solve a problem may depend on the form of attack, serial or parallel, and that a parallel attack might substantially diminish the invested time to solve a problem.

The hope that parallelism is a very effective speeding strategy has promoted since ancient times the idea that computing is best done in a parallel than in serial approach. For that reason, parallel computers with a huge number of processors have been built and commercialized at a high cost.

Our evolJava Community has the option offered by Java of uniting million of desktop computers in a single structure called a **grid** to exploit the potentialities of parallel computation. As we can imagine, this demands daunting efforts from the part of developers. Nevertheless, the grid is not as important as the community itself and we will not imagine a grid of computers but a grid of persons committed to the study of evolution. This is important and must be clearly remembered.

7.4 A nice hint for evolution

Our previous experience in which we passed from a failure to a success in the design of a binary adder could be framed as follows: a very general programming scheme converges to the solution of a specific problem very slowly but can be adapted to solve other problems rather easily. By contrast, a very specialized programming scheme, as our parallel binary adders and that are full in hints, can swiftly solve specific problems very fast but are useless to deal with other problems. So, hints push us into the realm of specialized programming. Now, it seems that we cannot escape specialized programming because very general unspecific programming may find the solution only after an infinite time of iterations or with an infinite cost in memory.

Specialized programming means specific hints. Therefore, it is a good moment to discuss a proposal to automatically generate hints that was formulated thanks to the study of the genome.

111 ***The importance of duplications***

According to evolutionists, a good portion of the evolution of the genome can be explained by recurring to just one single type of mutation: duplication of some part of the genome (Wen-Hsiung, [54] 1997, chapter 10).

From the computing science stand view, the thesis of duplications reads like this: usual tasks in programming are solved by recursion to some few units. Thus, programs that solve usual tasks must be full in for and while loops, and in slight variations to some procedures (given in Java by parameterized methods). This is certainly true and it is the very reason of the existence of computing science and all derived technology. In other words, the discovery of molecular biologists about the relevance of duplications is included in Java as part of its very essence. It also has been included in our simulations because there is no simulation without a for-loop.

Duplications are included in our simulations of evolution in the form of copy-Paste operations. Nevertheless, there is a remarkable difference within executing copyPaste operations at random and the thesis of duplications, which is at the heart of computing science: the allowed mutations over our chromosomes shall give preference to duplications rather than to use copyPaste operations at random which include frequent deletions (a deletion is also a copyPaste operation: it amounts to copy the non deleted regions), recombination, replacements and all what one can imagine. The form to apply this is to favor duplications in our programs.

In hindsight, the binary adders that we have devised comply with the specifications of the thesis of duplications because they are dominated by for loops and contain repetitive use of certain instructions, say, replacement of one substring by another.

In spite of the fact that the thesis of duplications is automatically included in all what we done, it is nevertheless interesting to keep an eye on the treatment we give to duplications.

112 *Translation to evolutionary language and grammars*

To arm evolution with appropriate hints to solve the problem of designing a binary adder we have solved that same task by ourselves. Now, we are in position of formulating the most powerful hint that can be invented to solve the problem of designing a parallel binary adder: each method in the program FullParallelBinaryAdder, B107, will be taken as an elementary building symbol for our genetic code. So, a chromosome will be a string with symbols in which it is written the order as the diverse methods will be called. At this step, we will have only one chromosome with exactly the same information given by our FullParallelBinaryAdder. In the next step, chromosomes will be assembled at random and tested for efficacy to add binary numbers.

Our methods consists of various instructions of the form replace a by b inside c. These sets of rules are called **grammars**.

Erratum: in the first version of this volume, a grammar was defined as a set of words or strings of chars from an alphabet.

Additionally, evolution is imagined in biology as a mechanism to cause an automatic escalade toward complexity. Thus, we must include in our simulations the possibility of learning. In relation with a binary adder this is readily implemented: once we have learned to add short inputs, we try to update our know-how to add larger ones.

The program that contains the translation follows:

```
//Program BinaryAdderTrans, B112
//The FullParallelBinaryAdder, B107,
//is translated into evolutionary language.
//The style is   cleaned.
package ejvol2v2p;

import java.util.Random;

public class BinaryAdderTrans {
    //The algorithm takes pairs of numbers
    //with numberOfBits ciphers.

    static int numberOfPairsOfContractions = 15;
    //Strings that encode numbers to be added
    static String input1, input2;
    static String output1, output2;
    //strings are converted to numbers
    static int nInput1, nInput2;
    static double nOutput1, nOutput2;
    //Error = difference among expected
    //addition and output
    static double error1, errork;
    //Number of ciphers or bits in the inputs

    static int stringLength;
    static int lengthInput;
    static int n;
```

```
static String chromosome1 = "";
static String chromosome2 = "";

//Turn on of the random generator
static Random r = new Random();



// A string of stringLength characters is generated
// The alphabet is {0,1}.
private static String generateStringNumber(int stringLength) {
    String individual = "";
    //An individual is assembled char by char
    for (int j = 1; j <= stringLength; j++) {
        // We generate a random binary number as Character c
        int binary = r.nextInt(2);
        //System.out.println( "binary number = "+ binary );
        char c = (char) (binary + 48);
        //Character c is converted into  string s
        String s = Character.toString(c);
        //System.out.println( "binary string = "+ s );
        // s is concatenated to Pop1[i]
        individual = individual + s;
    }
    //System.out.println( individual);
    return individual;
}

// Reads the decimal number in  string a
private static int decoder(String a) {
    int i = 0;
    int k;
    int length = a.length();
    for (int j = 0; j < length; j++) {
        char c = a.charAt(j);
        if (c == '1') {
            k = 1;
        } else {
            k = 0;
        }
```

```
            i = k + i * 2;
        }
        return i;
    }

    // Posits zeros at the left of the number
    // until all sites are occupied
    private static String complete(String s) {
        int j = stringLength - s.length();
        for (int i = 1; i <= j; i++) {
            s = '0' + s;
        }
        return s;
    }

    // The problem is defined
    private static void defineProblem(int k) {
        input1 = generateStringNumber(k);
        input1 = complete(input1);
        input2 = generateStringNumber(k);
        input2 = complete(input2);
        nInput1 = decoder(input1);
        nInput2 = decoder(input2);
        /*System.out.println("input1= " + input1  );
  System.out.println("input2= " + input2 );
         */
    }

    // Intercalates input 1 with input2
    // char by char and posits a separator "Y"
    private static String intercalation() {
        int length1 = input1.length();
        int length2 = input2.length();
        stringLength = Math.max(length1, length2);
        input1 = complete(input1);
        input2 = complete(input2);
        String output = "";
        for (int i = 0; i < stringLength - 1; i++) {
            output = output + input1.charAt(i) + input2.charAt(i) + "Y";
        }
```

```
        output = output + input1.charAt(stringLength - 1)
                + input2.charAt(stringLength - 1);
        return output;
    }

    private static String verticalSum(String output) {
        output = output.replaceAll("10", "01");
        output = output.replaceAll("11", "10");
        return output;
    }

    private static String contraction1(String output) {
        output = output.replaceAll("0Y0", "M00");
        output = output.replaceAll("1Y0", "M01");
        output = output.replaceAll("0Y1", "M01");
        output = output.replaceAll("1Y1", "M10");
        n = output.length();
        //A "M0" is inserted
        output = output.substring(0, n - 1)
                + "M0" + output.substring(n - 1);

        return output;
    }

    private static String contraction2(String output) {
        output = output.replaceAll("0M0", "Y00");
        output = output.replaceAll("1M0", "Y01");
        output = output.replaceAll("0M1", "Y01");
        output = output.replaceAll("1M1", "Y10");
        //A "Y0" is inserted
        n = output.length();
        output = output.substring(0, n - 1)
                + "Y0" + output.substring(n - 1);
        return output;
    }

    private static String finalContraction(String output) {
        output = output.replaceAll("01", "1");
        output = output.replaceAll("00", "0");
        //In case of uncooked solutions
```

```
        output = output.replaceAll("10", "1");
        output = output.replaceAll("Y", "");
        output = output.replaceAll("M", "");
        return output;
    }

    // This is  our genetic code,
    // we use five codons
    private static String geneticCode(char c, String output) {
        switch (c) {
            case 'A':
                output = verticalSum(output);
                break;
            case 'B':
                output = contraction1(output);
                break;
            case 'C':
                output = contraction2(output);
                break;
            case 'D':
                output = finalContraction(output);
                break;
        }
        return output;
    }

    //  This interpreter plays the role of the ribosome
    private static String interpreter(String chro, String output) {
        int n = chro.length();
        for (int i = 0; i < n; i++) {
            char c = chro.charAt(i);
            output = geneticCode(c, output);
        }
        return output;
    }

    // This is the old solution1
    private static String solution1() {
        String output = intercalation();
        output = verticalSum(output);
```

```
        output = contraction1(output);
        output = finalContraction(output);
        return output;
    }
    // This is solution1 in evolutionary language

    private static String genSolution1() {
        String output = intercalation();
        chromosome1 = "abe";
        output = interpreter(chromosome1, output);
        return output;
    }

    // This is the old solution2;
    private static String solution2() {
        String output = intercalation();
        output = verticalSum(output);
        for (int l = 0; l < numberOfPairsOfContractions; l++) {
            output = contraction1(output);
            output = contraction2(output);
        }
        output = finalContraction(output);
        return output;
    }

    //This is solution2 in evolutionary language
    private static String genSolution2() {
        String output = intercalation();
        chromosome1 = "ABCBCBCBCBCBCBCBCBCBCBCBCBCBCBCD";
        output = interpreter(chromosome1, output);
        return output;
    }

    // Two algorithms are tested for how well they add
    static public void workSolutions(int lengthInput) {
        //Checking out of  algorithms with
        //some random inputs.
        error1 = 0;
        errork = 0;
```

```
        for (int i = 0; i < 10 * lengthInput; i++) {
            defineProblem(lengthInput);
            output1 = genSolution1(); //one contraction
            nOutput1 = decoder(output1);
            //The error is the difference between the correct
            //addition and that proposed by the individual.
            error1 = error1 + Math.abs(nInput1 + nInput2 - nOutput1);
            output2 = genSolution2();//as many contractions as desired
            nOutput2 = decoder(output2);
            errork = errork + Math.abs(nInput1 + nInput2 - nOutput2);
        }
    }

    public static void main(String[] args) {
        /*int l = decoder("100010");
  System.out.println(l); */

        //r.setSeed(Int) causes the same sequence of
        //random numbers to be  always be generated.
        //r.setSeed(1);
        int seed = 54321;
        r.setSeed(seed);
        System.out.println("Seed = " + seed);
        System.out.println("inputLength \t  error1 \t "
                + "errork \t relation");
        for (int k = 1; k <= 200; k++) {
            // k is the number of places or bits of inputs
            lengthInput = k;
            workSolutions(lengthInput);
            double errorRelation = errork / error1;
            System.out.print(k + "\t ");
            System.out.print(error1 + "\t ");
            System.out.print(errork + "\t ");
            System.out.println(errorRelation);
        }
    }
}//End of Program BinaryAdderTrans, B112
```

113 ***Exercise*** *Run the code. Verbalize its design and functioning. Change the number of implemented contractions and study the ensuing effect.*

114 ***More hints***

We solved the problem of designing a binary adder and used the methods found in the solution program as the letters of an alphabet. Next, we used that alphabet to translate the know-how of the solution program into evolutionary language. As we will see, our alphabet allows evolution to design a binary adder in a very expedite way. So, our alphabet functions like a hint given to evolution that functions like an entity with artificial intelligence.

Our previous hint is not the only possible one. There are infinitely many possible hints. The next set of instructions also can be used as a hint: it is a bit more elementary and so the corresponding program shall converge slower than our former election.

```
//a: Table1 of the addition for the vertical sum
String  output2 = output1.replaceAll("10", "01");
String  output3 = output2.replaceAll("11", "10");

//b: Table2   of addition for contraction1
output = output.replaceAll("0Y0", "M00");
output = output.replaceAll("1Y0", "M01");
output = output.replaceAll("0Y1", "M01");
output = output.replaceAll("1Y1", "M10");

//c: Table3 of addition for contraction 2
output = output.replaceAll("0M0", "Y00");
output = output.replaceAll("1M0", "Y01");
output = output.replaceAll("0M1", "Y01");
output = output.replaceAll("1M1", "Y10");

//d: Measurement
n = output.length();

//e: Insertion1
output = output.substring(0,n-1)
                +"M0"+output.substring(n-1);

//f: Insertion2
output = output.substring(0,n-1)
```

```
                +"Y0"+output.substring(n-1);


 //h: Table4 of decoding
 output = output.replaceAll("01", "1");
 output = output.replaceAll("00", "0");
 output = output.replaceAll("10", "1");
 output = output.replaceAll("Y", "");
 output = output.replaceAll("M", "");

 //i:deletion
 output = output.replaceAll("Y", "");
```

We do not include in this list the intercalation method because it is part of the encoding of our problem.

One might think that we are engaged in a futile exercise: What is the aim of solving with evolution a problem that already has been solved by other methods? Does this means that evolution is useless if one does not know from the beginning how to solve the problem?

These questions are very important and must not be forgotten even if our next answer looks satisfactory:

- Programming evolution to solve a problem without giving appropriate hints is to get embarked in a nightmare of computations that may lead to nothing - in the scale of time of an impatient human being.

- All applications of evolution to industry and science are armed with very effective hints, whose existence is not always declared, and may be hidden just in the way as the problem is encoded: there are no limit to the diversity as a problem can be encoded and some forms could induce faster solution than others.

- A very usual form as a hint is embedded into a simulation of evolution is by providing an alphabet, a combinatorial basis from which the problem can be 'readily' solved. The developer must make sure that the solution is in the span of the combinatorial power of the chosen basis. This must be done from the very beginning although the actual form of the solution may be unknown. The natural example of a hint is the immune system: it contains a **combinatorial basis** to build antibodies against antigens. This

hint is effective at the population level but not at the individual one. The reason is that bases are different from individual to individual and for a given antigen some bases may do it better than others.

- To give effective hints is a task that is outside the scope of actual science. Effective hints are copied from nature else invented by a human being with a lot of expertise.

- In the long range, the study of evolution could be converted into the study of hints to have evolution functioning.

115 *Combinatorial bases as hints*

We have introduced the concept of combinatorial basis as fundamental to our theorization of evolution, so let us explain it a bit. This definition generalizes the concept of genetic code: the information of all genomes consists of strings of codons taken from a list with 64 possibilities. These 64 codons form a combinatorial basis for the genomes of nature. Most genomes use the same basis, the same genetic code. But there are exceptions, say, animal mitochondria use AUA for methionine instead of isoleucine as in the nuclear code.

In general, a **combinatorial basis** is a set of modules composed of elementary instructions. Modules are named with a letter and the resultant alphabet is used to build chromosomes. A combinatorial basis may enjoy three properties: sufficiency to solve certain family of problems, efficiency in the velocity of convergence to the solution (full or frustrated), and semantic independence, i.e., consisting in members that have not equivalent functions.

The semantics of an instruction is defined by a genetic code which establishes a correspondence between a codon or elementary instruction and the execution of the action prescribed by the genetic code. In nature, to each codon corresponds an amino acid or a grammatical signal like begin or end. But in applications, the genetic code can contain any type of correspondence. In our present example, the genetic code associates to each codon a method or procedure in Java. So, our chromosomes are in fact Java programs.

The genetic code is a very powerful hint. Let us see why: 61 out of 64 codons encode information to include an amino acid in a protein string. An amino acid is a molecule of the form $NH_3^+ - CR - COOH^-$, where the R is a side chain. For alanine, R is simply an hydrogen atom. Humans contain 25 amino acids, but the genetic code contains only 20. So, there are five amino acids that are synthesized by the cell from the others 20: a primordial protein is assembled and then a given amino acid is enzymatically modified. One example: 4-hydroxyproline is a modification of proline (Schmid, [46] 1982) that is found in collagen, whose synthesis

demands the collaboration of vitamin C. That is why vitamin C is good against infections: it helps in the repairing process of tissues.

Plants produce over 150 amino acids. But, theoretically, the number of possible amino acids could be infinite. So, a question springs at once: Is any election of 20 amino acids among that infinitude suitable for a genetic code?

Most probably not. So, how and why the present genetic code become established? This theme is currently investigated (see http://www.evolvingcode.net/) but our evolJava community could be delighted with its own study of the problem.

By the same token, any combinatorial basis in applied evolution presupposes one election among an infinitude, the majority of which could be useless because of inefficacy or by the ensuing exaggeratedly big combinatorial explosions (too many computations before arriving to a solution). This is why the design of software by evolutionary means is so difficult and mostly ignored or unknown.

116 *Evolution uses hints to make a new try*

We can now use evolution to design a binary adder using an appropriate combinatorial basis as hint. The inclusion of learning is considered as a fundamental piece of evolution and it is also included. To be fair, evolution runs many strategies in parallel, some of which enable learning but others not. Thus, learning is also a hint. Our new version 'recombines' programs AdditionRec, B96, in the exercises, and BinaryAdderTranslation, B112. Our recombination is far from a genetic one: it is an intelligent accommodation of necessary portions and appropriate changes into a functional unit. Various very lengthy trials were necessary for the Author before a good hint was found, i.e. one that causes the program to converge to a solution in a scale of time that is satisfactory for an impatient human being.

In the last program, BinaryAdderTranslation, B112, we found a solution chromosome to our task of designing a binary adder. So, we can think that we are again dealing with a **Weasel problem**, i.e., making a job to guess a predefined string. In the present case that could be right because we have chosen a combinatorial basis that puts the problem at one step from being solved. But in general that is not the case: a problem with a slight complexity could be solved by many different algorithms even if one solution is already known.

Since we are matching a predefined function, binary addition, we can say that we are using evolution in a design process. But the design is our responsibility not that of evolution.

The code follows:

```
//Program BinAdderHintEvol, B116
```

```
//Evolution is armed with powerful hints
//to synthesize a binary adder.
//In first instance, learning is allowed:
//from knowing how to add a short input to learning
//how to add a larger one.
//In second place, a tested combinatorial basis
//is directly included in our genetic code.
//In third place, we use an encoding that
//facilitates a parallel attack.

package ejvol2v2p;


import java.util.Random;

public class BinAdderHintEvol {
    //Strings that encode numbers to be added

    static String input1 = "1111";
    static String input2 = "1111";
    static String output = "0011";
    //Strings are converted to numbers
    static int nInput1, nInput2, nOutput;
    //Error = difference among expected addition
    //and output
    static int error = 0;
    //Number of ciphers or bits in the inputs
    static int lengthInput;
    //Max lengthInput
    static int stringLength = 30;
    //Chromosomes have a varying number of codons
    static int numbCodons = 8;

    static int numberOfPairsOfContractions = 15;
    static int n;

    //Chro1 and Chro2 are populations of individuals
    static String Chro1[];
    static String Chro2[];
    //limit is the number of individuals
```

```
static int limit = 30;

//Individuals are sorted by fitness in Order[].
static int Order[];
static int gen;
static boolean problem = true;

//VARIABLE PARAMETERS
static double pointMutationRate = 0.9;//per individual
static double dupliRate = 0.3;//per individual
static double deleRate = 0.3;//per individual

//Turn on of the random generator
static Random r = new Random();

private static final long serialVersionUID = 1L;

//********Declaration of methods*****************
//A string of stringLength characters is generated
//The alphabet is {A,B,C}. Final D is given for granted.
private static String generateString(int n) {
    String individual = "";
    //An individual is assembled char by char
    for (int j = 1; j <= n; j++) {
        // We generate a random  number less than 3
        int numb = r.nextInt(4);
        //System.out.println( "Number = "+ numb);
        //numb defines a letter
        char c = (char) (numb + 65);
        //Character c is converted into  string s
        String s = "" + c;
        // s is concatenated to Chro1[i]
        individual = individual + s;
    }
    //System.out.println( individual);
    return individual;
}

//An initial population   is set up.
private static void initialization() {
```

```
        //Formal declaration of our arrays.
        Chro1 = new String[limit];
        Chro2 = new String[limit];

        for (int i = 0; i < limit; i++) {
            //n= Number of codons in   chromosome i
            int m = r.nextInt(numbCodons) + 1;
            for (int j = 0; j < m; j++) {
                Chro1[i] = generateString(m);
                Chro2[i] = generateString(m);
                //Test: this chro must solve all problems
                /*Chro1[i]   = "ABCBCBCBCBCBCBCBCBCBCBCBCBCBCBCD";
   Chro2[i]  = "ABCBCBCBCBCBCBCBCBCBCBCBCBCBCBCD";*/

            }
        }

        // Declarations and default initialization
        Order = new int[limit];
        for (int i = 0; i < limit; i++) {
            Order[i] = 0;
        }
    }

    // A string of stringLength characters is generated
    // The alphabet is {0,1}.
    private static String generateStringNumber(int lengthInput) {
        String individual = "";
        //An individual is assembled char by char
        for (int j = 1; j <= lengthInput; j++) {
            // We generate a random binary number as Character c
            int binary = r.nextInt(2);
            //System.out.println( "binary number = "+ binary );
            char c = (char) (binary + 48);
            //Character c is converted into  string s
            String s = "" + c;
            //System.out.println( "binary string = "+ s );
            // s is concatenated to individual
            individual = individual + s;
        }
```

```
        //System.out.println( individual);
        return individual;
    }

    //Reads the decimal number in  string a
    private static int decoder(String a) {
        int i = 0;
        int k;
        int l = a.length();
        for (int j = 0; j < l; j++) {
            char c = a.charAt(j);
            if (c == '1') {
                k = 1;
            } else {
                k = 0;
            }
            i = k + i * 2;
        }
        return i;
    }

    //Posits zeros at the left of the number
    //until all sites are occupied
    private static String complete(String s) {
        int j = stringLength - s.length();
        for (int i = 1; i <= j; i++) {
            s = '0' + s;
        }
        return s;
    }

    //The problem is defined
    private static void defineProblem(int lengthInput) {
        input1 = generateStringNumber(lengthInput);
        input1 = complete(input1);
        input2 = generateStringNumber(lengthInput);
        input2 = complete(input2);
        nInput1 = decoder(input1);
        nInput2 = decoder(input2);
        // System.out.println("input1= " + input1  );
```

```
        //System.out.println("input2= " + input2 );
    }

    //Intercalates input 1 with input2
    //char by char and posits a separator "Y"
    private static String intercalation() {
        int l1 = input1.length();
        int l2 = input2.length();
        stringLength = Math.max(l1, l2);
        input1 = complete(input1);
        input2 = complete(input2);
        String result = "";
        for (int i = 0; i < stringLength - 1; i++) {
            result = result + input1.charAt(i) + input2.charAt(i) + "Y
        }
        result = result + input1.charAt(stringLength - 1)
                + input2.charAt(stringLength - 1);
        return result;
    }

    private static String verticalSum(String output) {
        output = output.replaceAll("10", "01");
        output = output.replaceAll("11", "10");
        return output;
    }

    private static String contraction1(String output) {
        output = output.replaceAll("0Y0", "M00");
        output = output.replaceAll("1Y0", "M01");
        output = output.replaceAll("0Y1", "M01");
        output = output.replaceAll("1Y1", "M10");
        n = output.length();
        //A "M0" is inserted
        output = output.substring(0, n - 1)
                + "M0" + output.substring(n - 1);

        return output;
    }

    private static String contraction2(String output) {
```

```
        output = output.replaceAll("0M0", "Y00");
        output = output.replaceAll("1M0", "Y01");
        output = output.replaceAll("0M1", "Y01");
        output = output.replaceAll("1M1", "Y10");
        //A "Y0" is inserted
        n = output.length();
        output = output.substring(0, n - 1)
                + "Y0" + output.substring(n - 1);
        return output;
    }

    private static String finalContraction(String output) {
        output = output.replaceAll("01", "1");
        output = output.replaceAll("00", "0");
        //In case of uncooked solutions
        output = output.replaceAll("10", "1");
        output = output.replaceAll("Y", "");
        output = output.replaceAll("M", "");
        return output;
    }

    //This is our genetic code.
    //We use four codons
    private static String geneticCode(char c, String output) {
        switch (c) {
            case 'A':
                output = verticalSum(output);
                break;
            case 'B':
                output = contraction1(output);
                break;
            case 'C':
                output = contraction2(output);
                break;
        }
        return output;
    }

    //This is our ribosome
    private static String interpreter(String chro, String output) {
```

```
        int l = chro.length();
        for (int i = 0; i < l; i++) {
            char c = chro.charAt(i);
            output = geneticCode(c, output);
        }
        return output;
    }

    //Just for reference:
    //This is the old solution2;
    private static String solution2() {
        String result = intercalation();
        result = verticalSum(result);
        for (int l = 0; l < numberOfPairsOfContractions; l++) {
            result = contraction1(result);
            result = contraction2(result);
        }
        result = finalContraction(result);
        return result;
    }

    //Just for reference:
    //This is solution2 in evolutionary language
    private static String genSolution2() {
        String result = intercalation();
        String chromosome1 = "ABCBCBCBCBCBCBCBCBCBCBCBCBCBCBCD";
        result = interpreter(chromosome1, result);
        return result;
    }

    //Each individual is tested for how well it adds
    static public int fitness(String chr) {
        //Checking out of the algorithm with
        //some random inputs.
        error = 0;
        for (int i = 0; i < 20 * lengthInput; i++) {
            defineProblem(lengthInput);
            String result = intercalation();
            result = interpreter(chr, result);
            result = finalContraction(result);
```

```
            //System.out.println();
            nOutput = decoder(result);
            //The error is the difference between the correct
            //addition and the result proposed by chr.
            error = error + Math.abs(nInput1 + nInput2 - nOutput);
        }
        //System.out.println("chr = " + chr);
        //System.out.println("Gen = " + gen + "Error = " + error);
        return error;
    }

    //Individuals are sorted by fitness.
    //the fittest is the first, with minimum error
    private static boolean sorting() {
        int Champ;
        int Fitness[];
        //Fitnessc[] is a copy of Fitness[]
        //used as workbench
        int Fitnessc[];
        Fitness = new int[limit];
        Fitnessc = new int[limit];
        for (int i = 0; i < limit; i++) {
            Fitness[i] = 1000000;
            Fitnessc[i] = 1000000;
        }
        for (int i = 0; i < limit; i++) {
            Fitness[i] = fitness(Chro1[i]);
            Fitnessc[i] = Fitness[i];
        }
        //We sort individuals by fitness
        // Fitness 0 means no errors
        for (int i = 0; i < limit; i++) {
            Champ = 0;
            for (int j = 0; j < limit; j++) {
                if (Fitnessc[j] < Fitnessc[Champ]) {
                    Champ = j;
                }
            }
            //The array Order classifies individuals by  fitness.
            //The fittest is number zero
```

```
            Order[i] = Champ;
            Fitnessc[Champ] = 1000000;
        }
        /*System.out.print("Champ= " + Chro1[Order[0]]);
            System.out.print("   Min Error = " + Fitness[Order[0]]);
            System.out.println();  */

                   //Printing job to test sorting procedure, limit = 5
                   /*for(int i = 0; i< limit;i++)
            System.out.println("Fitness  = " + Fitness[ i] );
            for(int i = 0; i< limit;i++)
            System.out.println(" Rank   " + i +" = "+ Order[i]);

            for(int i = 0; i< limit;i++)
            System.out.println("Fitness  = " + Fitness[Order[i]]);

            System.out.println();
         */
        if (Fitness[Order[0]] == 0) //if error = 0
        {
            problem = false;
            System.out.println("Task ended by generation =" + gen);
            System.out.println("Number of trials = " + gen * limit);
            System.out.println("Solution= " + Chro1[Order[0]]);
            System.out.println();
        }
        return problem;
    }

    //The new generation is made
    private static void newGeneration() {
        int i = 0;
        //The fittest individuals are reproduced
        // numbCopies times
        int numbCopies = 10;
        int k = 0;  //rank, 0 is the highest

        while (i < limit) {
            int counter = numbCopies;
            while ((counter > 0) & (i < limit)) {
```

```
                //System.out.println(i + "" + l);
                Chro2[i] = Chro1[Order[k]];
                //System.out.println(i + " " + counter + " " + k);
                i++;
                counter = counter - 1;
            }
            k++;
        }

        for (int j = 0; j < limit; j++) {
            Chro1[j] = Chro2[j];
            //System.out.println( j+ " length at newGen " + Pop1[j].length);
        }
    }

    //Recombination is enabled
    private static String rec(int l) {
        String s = Chro2[l];
        String s1 = "";
        int m = s.length();
        if (m > 0) {
            int m1 = r.nextInt(m);
            s1 = s.substring(0, m1);
        }

        int k = r.nextInt(limit);
        String t = Chro2[k];
        int nn = t.length();
        String t2 = "";
        if (nn > 0) {
            int m2 = r.nextInt(nn);
            t2 = t.substring(m2);
        }

        //Recombination
        return s1 + t2;
    }

    //The definitive population  is a recombinant one.
    private static void recombination() {
```

```
    //Chro1 is a recombinant version of Chro2
    for (int l = 0; l < limit; l++) {
        Chro2[l] = rec(l);
    }
    //Chro2 is transferred to Chro1
    for (int i = 0; i < limit; i++) {
        Chro1[i] = Chro2[i];
    }
}

//This method generates random chars
private static char randomChar() {
    // We generate a random   number less than 5
    int numb = r.nextInt(3);
    //System.out.println( "Number = "+ numb );
    char c = (char) (numb + 65);
    return c;
}

//String a is mutated at a random place.
//Method taken from weasel by Max Scott
private static String pointMutation(String a) {
    //System.out.println("Old a" + a);
    int l = r.nextInt(a.length());
    //System.out.println(l);
    String z = ((l <= 0) ? "" : a.substring(0, l))
            + randomChar()
            + ((l >= (a.length() - 1)) ? ""
                    : a.substring(l + 1));
    // System.out.println("Point Mutated a" + z);
    return z;
}

// A substring sub inside a is duplicated;
//sub is chosen at random.
private static String duplication(String a) {
    //System.out.println("old a = " + a);
    if (a.length() == 1) {
        a = a + a;
```

```
        } else {
            int l = r.nextInt(a.length());
            int m = r.nextInt(a.length());
            int min = Math.min(l, m);
            int max = Math.max(l, m);
            String sub = a.substring(min, max);
            //System.out.println(l + " " + m);
            a = a.substring(0, max)
                    + sub + a.substring(max);
        }
        //System.out.println("Duplication in a = " + a);
        return a;
    }

    private static String deletion(String a) {
        //System.out.println("old a = " + a);
        int l = r.nextInt(a.length());
        int m = r.nextInt(a.length());
        int min = Math.min(l, m);
        int max = Math.max(l, m);
        //Deletes from min to max
        String z = a.substring(0, min) + a.substring(max);
        return z;
    }

    //Individuals are mutated
    private static void mutation() {
        for (int i = 0; i < limit; i++) {
            double p = r.nextDouble();
            if (Chro1[i].equals("")) {
                Chro1[i] = generateString(numbCodons);
            }
            if (p < pointMutationRate) {
                Chro1[i] = pointMutation(Chro1[i]);
            }
            double q = r.nextDouble();
            if (q < dupliRate) {
                Chro1[i] = duplication(Chro1[i]);
            }
            double s = r.nextDouble();
```

```
            if (s < deleRate) {
                Chro1[i] = deletion(Chro1[i]);
            }
        }
    }

    //Given a goal, it is guessed by evolution
    private static void evolSolution(int n) {
        problem = true;
        //Individuals are sorted by fitness
        problem = sorting();

        while (problem) {
            //The new generation is produced
            //as copies of the fittest
            newGeneration();
            recombination();
            //The new population is subjected to mutation
            mutation();
            gen++;
            problem = sorting();
        }
        /*System.out.println("Length of input = "
+ lengthInput+"GEN= " + gen);*/
    }

    public static void main(String[] args) {
        /*int l = decoder("100111");
System.out.println(l);*/

        //r.setSeed(Int) causes the same sequence of
        //random numbers to be  always be generated.
        //r.setSeed(1);
        //int seed = 54321;
        //r.setSeed(seed);
        //System.out.println("Seed = " + seed);
        initialization();
        for (int k = 1; k <= 30; k++) {
            // k is the number of places or bits of inputs
            lengthInput = k;
```

```
            System.out.println("Length of input = " + lengthInput);
            gen = 1;
            evolSolution(lengthInput);
        }
    }
}//End of BinAdderHintEvol, B116
```

117 ***Exercise*** *Run the code various times. You can try various launches in parallel. This will take some time, so, you can use that time to study the code to clearly understand the nature of the hints given in the encoding of our genetic code. Verify whether or not the program finds a professional answer given by*

"ABCBCBCBCBCBCBCBCBCBCBCBCBCBCBC"

or if it just finds an approximation that is improved with time or maybe a variant. Answer

118 ***Exercise*** *We used in the program BinAdderHintEvol, B116, a hint that consisted in a division of labor in the next categories: intercalation(encoding), calculation and finalContraction(a sort of decoding). Intercalation and the final contraction were given for free of charge and were executed by the non evolutionary part of the program. Please, include finalContraction in the evolutionary quest: add a new letter, say D, for finalContraction, in the alphabet as well as in the genetic code. Run the new program to see the effect of this change and make and estimation of the power of the hint that consists in executing the finalContraction outside the evolutionary search.* Answer

7.5 Binary adder with a weaker hint

We have used evolution to design a binary adder and we have succeeded. Nevertheless, that exercise was good for training but it conveys no glory because we used very precise hints that posed the problem at one step of being solved. Our hints were given in the form of a powerful combinatorial basis that made things easy for evolution. Nevertheless, the training was fruitful: now we know that evolution cannot be reduced to copyPaste, as we thought formerly, but that `copyPaste + conditional if clauses` were a fair reduction. Necessary `ifs` are present in all replace commands because their operation begins with comparisons.

So, it is time to ask ourselves: What will happen if we deeply weaken hints?

119 ***Weakening hints***

Our task is to devise an evolutionary environment in which evolution could find by itself very explicit hints to solve the problem at hand: to design a parallel binary adder.

Our previous hint consisted in given an alphabet whose letters pointed to Java modules that in ensemble were capable of doing the asked task. We can weaken our hint if instead of those modules we give their constituent sentences. To facilitate checking, we transcribe here all methods that were used in our genetic code. The method intercalation() is not listed because it is part of the encoding of the problem and so it cannot be dispensed for. Exerts follow:

```
private static String verticalSum(String output) {
    output = output.replaceAll("10", "01");
    output = output.replaceAll("11", "10");
    return output;
}

private static String contraction1(String output) {
    output = output.replaceAll("0Y0", "M00");
    output = output.replaceAll("1Y0", "M01");
    output = output.replaceAll("0Y1", "M01");
    output = output.replaceAll("1Y1", "M10");
    n = output.length();
    //A "M0" is inserted
    output = output.substring(0, n - 1)
            + "M0" + output.substring(n - 1);

    return output;
}

private static String contraction2(String output) {
    output = output.replaceAll("0M0", "Y00");
    output = output.replaceAll("1M0", "Y01");
    output = output.replaceAll("0M1", "Y01");
    output = output.replaceAll("1M1", "Y10");
    //A "Y0" is inserted
    n = output.length();
    output = output.substring(0, n - 1)
            + "Y0" + output.substring(n - 1);
    return output;
```

```
}

private static String finalContraction(String output) {
    output = output.replaceAll("01", "1");
    output = output.replaceAll("00", "0");
    //In case of uncooked solutions
    output = output.replaceAll("10", "1");
    output = output.replaceAll("Y", "");
    output = output.replaceAll("M", "");
    return output;
}
```

So, all elementary instructions are:

```
    output = output.replaceAll("10", "01");
    output = output.replaceAll("11", "10");

    output = output.replaceAll("0Y0", "M00");
    output = output.replaceAll("1Y0", "M01");
    output = output.replaceAll("0Y1", "M01");
    output = output.replaceAll("1Y1", "M10");
    n = output.length();
    //A "M0" is inserted
    output = output.substring(0, n - 1)
            + "M0" + output.substring(n - 1);

    output = output.replaceAll("0M0", "Y00");
    output = output.replaceAll("1M0", "Y01");
    output = output.replaceAll("0M1", "Y01");
    output = output.replaceAll("1M1", "Y10");
    //A "Y0" is inserted
    n = output.length();
    output = output.substring(0, n - 1)
            + "Y0" + output.substring(n - 1);


    output = output.replaceAll("01", "1");
```

```
        output = output.replaceAll("00", "0");
        //In case of uncooked solutions
        output = output.replaceAll("10", "1");
        output = output.replaceAll("Y", "");
        output = output.replaceAll("M", "");

    }
```

A representative subset of these instructions is the following:

```
output = output.replaceAll("10", "01");
output = output.replaceAll("0Y0", "M00");
output = output.replaceAll("0M0", "Y00");
output = output.replaceAll("01", "1");
output = output.replaceAll("Y", "");
output = output.replaceAll("M", "");

n = output.length();
output = output.substring(0,n-1)
                 +"M0"+output.substring(n-1);
output = output.substring(0,n)
             +"Y0"+output.substring(n);
```

This very specific set of instructions can be abstracted to a grammar that is completely described by a generalized replacement rule plus a measurement plus an instruction to make an insertion:

```
Let a, b, c be strings with length less than 3 (0 included)
formed from the alphabet {0,1,M,Y}.

//Replacement
output = output.replaceAll(a, b);

n = output.length(); (measurement)
Let k be 0 or 1 or 2:
//Insertion
output = output.substring(0,n-k)
                 +"c"+output.substring(n-k);
```

It is by no means obvious how one can use this new grammar to implement a binary adder. So, it does not pose the problem one step apart from being solved and so we will use it to try to save our honor.

120 ***Our new codons***

Our codons must have the following type of information:

- A zero else a one to declare if we deal with a replacement else with an insertion.
- If we have a generalized replacement, we must decide which substring a must be replaced by which other b.
- If we have an insertion, we always must make a measurement to know the length of the target string and next define a place k for insertion of a given substring c.

Thus, a codon consists of three integers. The first integer takes values on the set $\{0,1\}$. If we have a zero, a replacement, the other two integers are interpreted as two addresses in a vector of strings to decide which substring a must be replaced by which other b. If we have a one, an insertion, the second integer denotes the place of insertion while the third integer denotes the address in the vector of strings to decide which string c must be inserted.

Our codons work if we define the vector of source strings. So, we will posit in that vector all possible strings that can be formed with less than 3 letters from the alphabet $\{0,1,M,Y\}$.

121 ***Our code*** *The following program pose to evolution the problem of designing a binary adder. A low level but not trial hint is used. The code follows:*

```
//Program AdderEvol, B121.
//The design of a binary adder is given as a task
//to evolution.
//A low level hint is used.

package ejvol2v2p;

import java.util.Random;
```

```
public  class AdderEvol {
    //The next constructor is necessary when
    //we have new types or inner classes.

    public AdderEvol() //constructor
    {
    }

    private static AdderEvol x = new AdderEvol(); //x = Instance



    //****Declaration of new types as inner classes**
    //New type Codon
    static class Codon {

        int first; //0=replacement, 1=insertion
        int second; //place or address
        int third;  // address in a vector of strings

        Codon() {
            first = 0;
            second = 0;
            third = 0;
        }

        //A codon can be defined by giving some integers
        Codon(int j, int k, int l) {
            first = j;
            second = k;
            third = l;
        }

        //A codon can be copied from another codon g
        Codon(Codon g) {
            first = g.first;
            second = g.second;
            third = g.third;
        }
    }
```

```
//New type Chromosome
//A chromosome is an array of codons.
//Chromosome is synonymous of individual
static class Chromosome {

    int length;
    Codon c[];

    //Everything in Java must be initialized
    //otherwise, an nillpointer error will be generated.
    //Default initialization to zero
    public Chromosome(int n) {
        Codon cd = new Codon(0, 0, 0);
        c = new Codon[n];
        for (int i = 0; i < n; i++) {
            c[i] = cd;
        }
    }

    //A chromosome can be
    //copied as a clone from another one.
    public Chromosome(Chromosome g) {
        chromosomeLength = g.length;

        for (int i = 0; i < chromosomeLength; i++) {
            c[i] = g.c[i];
        }
    }
}


//******Declaration of variables*********

//for exact fitting, pose ToleratedError = 0;
//You might need 10 years to accomplish your work.
private static int ToleratedError = 10;
private static int numberOfTrials;

//  Strings that encode numbers to be added
```

```
//  strings are converted to numbers
private static int nInput1, nInput2, nOutput;
//  Error = difference among expected
//  addition and output
private static double error = 0;
//  Number of ciphers or bits in the inputs
private static int lengthInput;
//  Number of ciphers in all of our strings
private static int stringLength = 20;
private static int chromosomeLength;
private static int maxNumbCodons = 10;

//  We will have a population with
//  a number of POPSIZE individuals
private static int POPSIZE = 10;
private static Chromosome Pop1[];
private static Chromosome Pop2[];

// Individuals are sorted by fitness in Order[].
private static int Order[];
private static int gen;
private static boolean problem = true;

// CONTROL PARAMETERS
private static double pointMutationRate = 0.5;//per individual
private static double dupliRate = 0.01;//per individual
private static double deleRate = 0.01;//per individual

//  List contains all   words from the alphabet {0,1,M,Y}
//  with 3 or less chars
private static final int NLIST = 85;
private static String List[] = new String[NLIST];

private static boolean print = false;

// Turn on of the random generator
private static Random r = new Random();
```

```
//********Declaration of methods*****************
// All words with less than 3 chars are generated.
//Alphabet = {0,1,M,Y}
private static void generateList() {
    String alphabet[] = {"0", "1", "M", "Y"};
    List[0] = "";

    //strings with one char
    int counter = 1;
    for (int i = 1; i < 5; i++) {
        List[counter] = alphabet[i - 1];
        counter = counter + 1;
    }

    //strings with two chars
    for (int i = 1; i < 5; i++) {
        for (int j = 1; j < 5; j++) {
            List[counter] = List[i] + alphabet[j - 1];
            counter = counter + 1;
        }
    }

//    strings with three chars
    for (int i = 5; i < 21; i++) {
        for (int j = 1; j < 5; j++) {
            List[counter] = List[i] + alphabet[j - 1];
            counter = counter + 1;
        }
    }

    //Printing checkout
    System.out.println("The vector with all source strings");
    for (int i = 0; i < 85; i++) {
        System.out.println(i + " " + List[i]);
    }
    System.out.println();
}

//  A string of stringLength characters is generated
//  The alphabet is {0,1}.
```

```
private static String generateStringNumber(int stringLength) {
    String sNumber = "";
    //An individual is assembled char by char
    for (int j = 0; j < stringLength; j++) {
        // We generate a random binary number as Character c
        int binary = r.nextInt(2);
        sNumber = sNumber +(char) (binary + 48);
    }
    //System.out.println( " sNumber = " + sNumber);
    return sNumber;
}


private static void printCodon(Codon cd) {
    System.out.println("cd.first = " + cd.first
            + "  cd.second = " + cd.second
            + "  cd.third = " + cd.third );


    if (cd.first == 0)//replacement
    {
        String a = List[cd.second];
        String b = List[cd.third];
        System.out.println("Replace "  +  a  + " by " + b);


    }
    else
    {
        int k = cd.second;
        String c = List[cd.third];
        System.out.println("Insert " + c
                + " at place n - " + k );
    }

}



private static void printChromosome(Chromosome ch) {
```

```
        int n = ch.length;
        System.out.println();
        for (int i = 0; i < n; i++) {
            printCodon(ch.c[i]);
        }
    }


    private static Codon generateCodon() {
        //Default initialization
        Codon c =  new Codon(0, 0, 0);
        //True initialization
        //0(replacement) else 1(insertion)
        c.first = r.nextInt(2);
        if (c.first == 0) {//Replacement
            //Second must be replaced by third.
            //There are 85 possible strings with 3 chars
            c.second = r.nextInt(85);
            c.third  = r.nextInt(85);
        }

        if (c.first == 1) {//Insertion
            //Insert at n else n-1
            c.second = r.nextInt(2);
            //String to insert
            c.third = r.nextInt(85);
        }


        return c;
    }


    private static Chromosome generateChro(int n) {
        Chromosome chr =  new Chromosome(maxNumbCodons);
        //System.out.println("Length at gen = " + n);
        chr.length = n;
        for (int j = 0; j < n; j++) {
            chr.c[j] = generateCodon();
        }
```

```
    return chr;
}


//  An initial population   is setted up.
private static void initialization() {

    System.out.println( decoder("11111"));
    //Formal declaration of our arrays.
    Pop1 = new Chromosome[POPSIZE];
    Pop2 = new Chromosome[POPSIZE];

    Chromosome chr =  new Chromosome(maxNumbCodons);
    for (int i = 0; i < POPSIZE; i++) {
        //n= Number of codons in   chromosome i
        int n = r.nextInt(maxNumbCodons);
        n = Math.max(2, n);
        //System.out.println("NumbCodons = " + n);
        Pop1[i] = generateChro(n);
        Pop2[i] = chr;
    }

    // Declarations and default initialization
    Order = new int[POPSIZE];
    for (int i = 0; i < POPSIZE; i++) {
        Order[i] = 0;
    }
}

//Reads the  number encoded by the binary string a
// Unprocessed literals are ignored
private static int decoder(String a) {
    int number = 0;
    int n = a.length();
    for (int j = 0; j < n; j++)
    {
        char c = a.charAt(j);
        switch (c)
        {
            case '0': number = number * 2; break;
```

```
                case '1': number = 1 + number * 2;    break;
            }
            //System.out.println(c + " -> " + number);
            //Literals are not caught
        }
        return number;
    }


    //  Posits zeros at the left of the number
    //  until all sites are occupied
    private static String complete(String s) {
        int l = stringLength - s.length();
        for (int i = 1; i < l; i++) {
            s = '0' + s;
        }
        return s;
    }

    // Intercalates input1 with input2
    // char by char and posits a separator "Y"
    private static String intercalation(String input1, String input2) {
        int n = input1.length();
        int m = input2.length();
        int max = Math.max(n, m);
        String output = "";
        for (int i = 0; i < max; i++) {
            output = output + input1.charAt(i) + input2.charAt(i) + "Y";
        }
        return output;
    }




    // This is our genetic code:
    private static String geneticCode(Codon cd, String output) {

        int n = output.length();
```

```
        if (cd.first == 0)//replacement
        {
            String a = List[cd.second];
            String b = List[cd.third];
            //System.out.println("Replace "  +  a  + " by " + b);
            //if( (n < 7) && ( a.length() >= b.length()))
                output = output.replace(a, b);
        }


            if (cd.first == 1)//insertion
        {
            int k = cd.second;
            String c = List[cd.third];
            //System.out.println("Insert " + c + " at place n - " + k),
            if (n > 0) {

                if (n - k > 0) {
                    output = output.substring(0, n - k)
                            + c + output.substring(n - k);
                }
            } else {
                output = c;
            }
        }

        return output;
    }


    // This is our ribosome.
    // Chromosome chr is executed as algorithm
    //  whose instructions operate in tandem over output
    private static String interpreter(Chromosome chro,
            String output) {
        int n = chro.length;
        for (int i = 0; i < n; i++) {
            output = geneticCode(chro.c[i], output);
        }
        return output;
```

```
    }


    //  Each individual is tested for how well it adds.
    //Random inputs.
    private static double fitness(Chromosome chr) {
        error = 0;
        /*
            System.out.println("Gen = " + gen + " " +
                "chr.length = "+chr.length);
        */
        for (int i = 0; i < numberOfTrials; i++) {

            String input1 = generateStringNumber(lengthInput);
            //input1 = complete(input1);
            String input2 = generateStringNumber(lengthInput);
            //input2 = complete(input2);
            String intercalation = intercalation(input1, input2);
            //intercalation = complete(intercalation);
            String interpretation = interpreter(chr, intercalation);
            //System.out.println();
            int result = decoder(interpretation);
            nInput1 = decoder(input1);
            nInput2 = decoder(input2);

            //The error is the difference between the correct
            //addition and that proposed by  chro.
            int myError = Math.abs(nInput1 + nInput2 - result);
            if (print)
            {
                System.out.println(
                      " Input1 = " + input1
                    + " Input2 = "   + input2
                    + " Intercalation = " + intercalation
                    + " Interpretation = " +  interpretation
                    + " Result = " + result
                    + " Error = "
                    + myError);
            }
            error = error + myError;
```

```
    }
if (print) System.out.println("Total error = " + error);
    return Math.abs(error);
}


// Individuals are sorted by fitness
//  The fittest is the first, with minimum error
private static boolean sorting() {
    //System.out.println("sorting");
    problem = true;
    int Champ = 0;
    double Fitness[];

    Fitness = new double[POPSIZE];


    for (int i = 0; i < POPSIZE; i++) {
        Fitness[i] = fitness(Pop1[i]);
        System.out.println("i = " + i + " lengthInput = " +lengthI
        " Gen = " + gen +  " Total Errorrrr = " + Fitness[i]);
    }

    //We sort individuals by fitness
    // Fitness 0 means no errors
    for (int i = 0; i < POPSIZE; i++) {
        Champ = 0;
        for (int j = 0; j < POPSIZE; j++) {
            if (Fitness[j] < Fitness[Champ]) {
                Champ = j;
            }
        }

        //The array Order classifies individuals by  fitness.
        //the fittest is number zero
        Order[i] = Champ;
        Fitness[Champ] = 1000000;
    }
    System.out.println("lengthInput = " + lengthInput
```

```
                + " Gen = " + gen
                + " Champ = "  + Order[0]
                );
        /*
        //Print job to test sorting procedure, POPSIZE = 5

 for(int i = 0; i< POPSIZE;i++)
  System.out.println("Fitness  = " + Fitness[Order[i]]);

 System.out.println();
         */
        if (fitness(Pop1[Order[0]]) <= ToleratedError) //if error = 0
        {
            problem = false;
            System.out.println("\nTask ended by generation = " + gen);
            System.out.println("Number of trials = " + numberOfTrials);
            System.out.println("\nCodons of the winner chromosome");
            printChromosome(Pop1[Order[0]]);
            print = true;
            System.out.println("\nTest of the winner with a new set of random
inputs");
            fitness(Pop1[Order[0]]);
            print = false;
        }
        return problem;
    }

    // The new generation is made
    private static void newGeneration() {
        int i = 0;
        //The fittest individuals are reproduced
        // numbCopies times
        int numbCopies = 10;
        int k = 0;  //rank, 0 is the highest

        while (i < POPSIZE) {
            int counter = numbCopies;
            while ((counter > 0) & (i < POPSIZE)) {
                //System.out.println(i + "" + l);
                Pop2[i] = Pop1[Order[k]];
```

```
                //System.out.println(i + " " + counter + " " + k);
                i++;
                counter = counter - 1;
            }
            k++;
        }

        for (int j = 0; j < POPSIZE; j++) {
            Pop1[j] = Pop2[j];
            /*
   System.out.println( j+ " length at newGen " +
    Pop1[j].length);
             */
        }
    }

    // Recombination is enabled
    private static Chromosome rec(int l) {
        //the first part of "l" is copied
        int m1 = 1;
        int m = Pop1[l].length;
        if (m > 0) {
            m1 = r.nextInt(m);
            for (int i = 0; i < m1; i++) {
                Pop2[l].c[i] = Pop1[l].c[i];
            }
        }
        //The second part of "k" is copied
        int k = r.nextInt(POPSIZE);
        int n = Pop1[k].length;
        if (n > 0) {
            int n1 = r.nextInt(n);
            for (int i = m1; i < m1 + n - n1; i++) {
                if (i < maxNumbCodons) {
                    Pop2[l].c[i] = Pop1[k].c[n1 + i - m1];
                }
            }
        }
        return Pop2[l];
    }
```

```
    // The definitive population is a recombinant one.
    private static void recombination() {


        //Recombination
        for (int l = 0; l < POPSIZE; l++) {
            Pop2[l] = rec(l);
        }
        //Pop2 is transferred to Pop1
        for (int i = 0; i < POPSIZE; i++) {
            Pop1[i] = Pop2[i];
        }
    }

    // Chro g a is mutated at a random codon.
    private static Chromosome pointMutation(Chromosome g) {
        //System.out.println("PointMutation");
        int l = r.nextInt(g.length);
        g.c[l] = generateCodon();
        return g;
    }

    private static Chromosome duplication(Chromosome g) {
        //System.out.println("duplication");
        /*
  if (g.length ==0)
  System.out.println("Old length at dup = " + g.length);
         */
        int l = r.nextInt(g.length);
        int m = r.nextInt(g.length);
        int min = Math.min(l, m);
        int max = Math.max(l, m);
        //System.out.println("Old length at dup = " + g.length);
        //System.out.println("min = " + min + " Max " + max);
        Chromosome chr = g;
        int top = Math.min(2 * max - min, maxNumbCodons);
        for (int i = max; i < top; i++) {
            g.c[i] = chr.c[i - (max - min)];
        }
```

```
        int toptop = Math.min(chr.length + max - min, maxNumbCodons);
        if (top < maxNumbCodons) {
            for (int i = top; i < toptop; i++) {
                g.c[i] = chr.c[i - (max - min)];
            }
        }
        g.length = toptop;
        /*
  * System.out.println("min = " + min +
   "max = " + max +  "top = " + top + "toptop = " + toptop);
         */
        //System.out.println("New length at dup = " + g.length);

        //System.out.println();
        return g;
    }

    private static Chromosome deletion(Chromosome g) {
        //System.out.println("deletion");
        //System.out.println("Old length at del = " + g.length);
        int l = r.nextInt(g.length);
        int m = r.nextInt(g.length);
        int min = Math.min(l, m);
        int max = Math.max(l, m);
        //Deletion = identity codon
        for (int i = max; i < g.length; i++) {
            g.c[i - max + min] = new Codon();
        }

        //System.out.println("Max = " + max + "Min = " + min);
        g.length = g.length - (max - min);
        //System.out.println("New length at del = " + g.length);

        if (g.length < 2) {

            int n = r.nextInt(maxNumbCodons);
            n = Math.max(2, n);
            g = generateChro(n);
        }
        return g;
```

```
    }

    // Chromosome g is mutated at a random codon.
    private static Chromosome change(Chromosome g) {
        double p = r.nextDouble();
        if (p < pointMutationRate) {
            g = pointMutation(g);
        }
        double q = r.nextDouble();
        if (q < dupliRate) {
            g = duplication(g);
        }
        double s = r.nextDouble();
        if (s < deleRate) {
            g = deletion(g);
        }
        return g;
    }


    //  All individuals are mutated
    private static void mutation() {
        for (int i = 0; i < POPSIZE; i++) {
            //System.out.println( "gen = " + gen);
            //int l = Pop1[i].length;
            //System.out.println( i + " oldLength at mut = " + l);
            Pop1[i] = change(Pop1[i]);
            //int m = Pop1[i].length;
            //System.out.println(i + " newLength at mut = " + m);
        }
    }

    // Given a goal, it is guessed by evolution
    private static void evolSolution(int n) {
        //Individuals are sorted by fitness
        problem = sorting();
        while (problem) {
            newGeneration();
            //The new population is subjected to mutation
            //mutation();
```

```
            recombination();
            gen++;
            if (gen % 1000 == 0) {
                System.out.println("Length of input = "
                        + lengthInput + " Gen = " + gen);
            }
            problem = sorting();
        }
    }

    public static void main(String[] args) {
        /*int l = decoder("100111");
        System.out.println(l);*/

        //r.setSeed(Int) causes always the same sequence of
        //random numbers to be generated.
        //r.setSeed(1);
        generateList();
        /*
            int seed = 1000000;
            r.setSeed(seed);
            System.out.println("Seed = " + seed);
        */
        initialization();
        for (int k = 1; k <= 1; k++) {
            // k is the number of places or bits of inputs
            lengthInput = k;
            numberOfTrials = 10 * lengthInput;
            System.out.println("Length of input = " + lengthInput);
            // maxNumbCodons = 5*k;
            gen = 1;
            problem = true;
            while (problem) {
                evolSolution(lengthInput);
            }
            problem = false;
            System.out.println();
        }
    }
}//End of Program AdderEvol, B121.
```

122 ***Exercise*** *Run the program and play with the code. Play enough to accept else reject the appreciation of the Author: the algorithm can produce low quality solutions but as a rule they consist of code whose instructions are not applicable and so, they do nothing. Perfect solutions are not reached in within the patience of the Author. Why? It is because we deal with an irreducible combinatorial explosion. To see what the combinatorial explosion consist in let us make a calculation to know the number of independent trials that must be done to check all the possibilities to build a program that adds two bits. This case corresponds to the usual table of the sum, 0 + 0 = 0, and other 3 equations. The direct encoding of the adding algorithm has four replace instructions. Because each instruction has two arguments that are taken from a set with 85 elements, we need to check* $85^8 > 2.7 \times 10^{15}$ *cases. This number is outside the possibilities of desktops. Now, a question is in order: what is the role of evolution, if any? Supposing that the program is correct, we might claim that evolution is nearly useless, i.e., that the space of replace clauses of our particular language is totally discrete, disconnected one from another. Hence, there is no small change and so, evolution is nearly useless.*

123 ***No learning***

Evolution is a learning process guided by accumulation of small change. That is why evolution is beloved everywhere. But our last exercise fails to show us what we expected: the fitness function shall decrease with time but our program produces a fitness that dances around certain average value but does not find its way to success. Dancing could be a result of our probabilistic testing: we test a program not over all possible inputs but over a sample, so the dancing in fitness also could reflect the dancing of the sample. But the important thing is that we do not find the way down to success.

What do we find in here? A recurrent failure of us as developers? A misunderstanding of evolution? An intrinsic failure of evolution?

With time and effort, we will make our mind but there is a difficulty that will remain as a recurrent problem forever: software is chaotic, i.e., a minor change produces terrible separations from the goal. So, mutation does not necessarily produces raw material for improving our state but it functions as a stumbling stone that causes one to fall in the basin of a **combinatorial explosion** from where one cannot leave because of the immense numbers of alternatives to test out.

We find the chaotic nature of software at the root of all our problems. one may commit a minor bug that escapes from his or her control but that causes a total crash of the function that must be fulfilled.

124 ***Exercise*** *Before going into desperation, it would be convenient to check two major causes of discomfort. First: we have as yet not proved that a perfect*

algorithm is attainable for our program and, second: what we have called point-Mutation is a change of one codon for another. But a codon is too large an entity and so we loss the opportunity to produce tiny changes that possibly produce small but decisive tuning effects. Please, correct these two problems. Hint: to prove that the program contains the solution as a possible state, it is suffice to encode a known solution in a chromosome and to run it to verify that it solves the problem. The known solution is found in program Program FullParallelBinaryAdder, B107. On the other hand, to allow for fine mutations, it is enough to change the inner components of the codons instead of codons themselves. Answer

To facilitate the task of solving this exercise, let us reproduce here the vector with all source strings:

```
0
1 0
2 1
3 M
4 Y
5 00
6 01
7 0M
8 0Y
9 10
10 11
11 1M
12 1Y
13 M0
14 M1
15 MM
16 MY
17 Y0
18 Y1
19 YM
20 YY
21 000
22 001
23 00M
24 00Y
25 010
26 011
27 01M
```

```
28 01Y
29 0M0
30 0M1
31 0MM
32 0MY
33 0Y0
34 0Y1
35 0YM
36 0YY
37 100
38 101
39 10M
40 10Y
41 110
42 111
43 11M
44 11Y
45 1M0
46 1M1
47 1MM
48 1MY
49 1Y0
50 1Y1
51 1YM
52 1YY
53 M00
54 M01
55 M0M
56 M0Y
57 M10
58 M11
59 M1M
60 M1Y
61 MM0
62 MM1
63 MMM
64 MMY
65 MY0
66 MY1
67 MYM
```

```
68 MYY
69 Y00
70 Y01
71 Y0M
72 Y0Y
73 Y10
74 Y11
75 Y1M
76 Y1Y
77 YM0
78 YM1
79 YMM
80 YMY
81 YY0
82 YY1
83 YYM
84 YYY
```

7.6 Conclusion

Our aim was to force evolution to design a parallel binary adder. At the outset we thought that evolution corresponded with text processing that can be reduced to copy-paste. Battling with a combinatorial explosion we decided to change our mind and concluded that a new ingredient was needed: the capability to process if-instructions. All that was naturally contained in the so called grammars, which are sets of replacement rules to process strings. So, with the purpose of succeed we armed evolution with a genetic code or combinatorial basis that poses the problem at one step from being solved. In this case, our combinatorial basis was formed by some few Java methods that were previously known to function. And, we succeeded. So, we claim that evolution is an artificial intelligence capable of software design for mathematical enterprises. By the way, as in the previous program WORMS as in our Binary adder, we have seen the importance of duplications of code, a fact that also has been underscored by molecular biologists. Anyway, duplications of code conform a basic feature of high programming languages that are implemented in loops of the type `for` or `while` so, duplications are backed up by the recursiveness that form the very nature of software. But in spite of our success, we feel that our honor has been offended because we have been unable to force evolution to do the task beginning from a more elementary basis. In fact, if instead of some few methods we give as combinatorial basis elementary instructions to

build the sentences of those methods, we get nothing.

In general, a hint consist in the definition of an effective combinatorial basis upon which evolution works. Two natural hints are: the genetic code, with its 20 amino acids, and the immune system. At the scale of time of an impatient human developer that is armed with a single desktop, a basis to be reliable must have very few and well chosen members. Else, he or she will fall prey of an irreducible combinatorial explosion, one against evolution is impotent. In our case, this seems to be the result of the strong discontinuity of software, of verbal instructions.

Chapter 8

Evolution of genetic codes

`Hints for hints`

125 ***Motivation*** *We learned in the last chapter with sweat and blood that to choose a reasonable combinatorial basis for an evolutionary process used for the design of software is a very difficult task. The problem is that we are compelled to begin with a basis that puts the problem at one step of being solved else to propitiate a leading to nowhere combinatorial explosion.*

So, we have every justification to say that the election of a right combinatorial basis is indeed a powerful hint given to evolution, which functions as an artificial intelligence trying to develop software. Unfortunately, all used hints until now came from the human imagination. So, we ask: Can the process of generating hints be automated anyhow? Can evolution replace us?

126 ***Purpose*** *We explore in this chapter the idea of using evolution over the space of hints to catch appropriate hints, i.e. to enumerate a combinatorial basis that could be given to a low level evolutionary process to solve a problem of software design.*

8.1 A hierarchy with two levels

An ordinary evolutionary process runs over a given combinatorial basis. If we allow the basis to evolve, we end with a problem involving a two level hierarchy : the first is at the basis and the second at the chromosomes that the basis allows to be built.

127 ***Our code*** *The implementation of a two level hierarchy is shown in the next code:*

```
//Program TwoLevelHier, B127
//We begin the study of the evolution of hints
//to enable evolution to design a
//parallel binary adder.
//A two level hierarchy is implemented.
//The very general and large hint contained in List
//and defined in motherList() gives rise
//to shorter but possibly more appropriate hints
//that are contained in Lists[i][j].
//The general algorithm is as follows:
//To begin with, n short hints are generated.
//Each hint generates a trend of evolution
//of binary adders. Hints that produce
//the best binary adders, if any, are the fittest
//and are reproduced. And so on.
package ejvol2v2p;

import java.util.Random;

public class TwoLevelsHier {
    //The next constructor is necessary when
    //we have new types or inner classes.

    public TwoLevelsHier() //constructor
    {
    }

    static TwoLevelsHier x = new TwoLevelsHier(); //x = Instance

    //****Declaration of new types as inner classes**
    //New type Codon
    class Codon {

        private int first; //0=replacement, 1=insertion
        private int second; //place or address
        private int third;  // address in a vector of strings

        //A codon can be defined given some integers
        Codon(int j, int k, int l) {
            first = j;
```

```
            second = k;
            third = l;
        }

        //A codon can be copied from another codon g
        Codon(Codon g) {
            first = g.first;
            second = g.second;
            third = g.third;
        }

    }

    //New type Chromosome
    //A chromosome is an array of codons.
    //Chromosome is synonymous of individual.
    class Chromosome {

        private int length;
        private Codon c[];

        //Everything in Java must be initialized
        //otherwise, an nillpointer error will be generated.
        //Default initialization to zero
        public Chromosome(int n) {
            length = n;
            Codon cd = new Codon(0, 0, 0);
            c = new Codon[n];
            for (int i = 0; i < n; i++) {
                c[i] = cd;
            }
        }

        //Once initialized, a chromosome can be
        //copied as a clone from another one.
        public Chromosome(Chromosome g) {
            length = g.length;

            for (int i = 0; i < length; i++) {
                c[i] = g.c[i];
```

```
        }
    }
}

//******Declaration of variables*********
//  Boolean variable to decide whether or not
//  a known solution must be tried out.
static boolean runTest;

//for exact fitting, pose ToleratedError = 0;
//You might need 10 years to accomplish your work.
private static int ToleratedError = 0;
private static int numberOfTrials;



//  strings are converted to numbers
static int nInput1, nInput2, nOutput;
//  Error = difference among expected
//  addition and output
static double error = 0;
//  Number of ciphers or bits in the inputs
static int lengthInput;
//  Number of ciphers in all of our strings
static int stringLength = 20;
static int maxNumbCodons = 100;

//  We will have a population with
//  a number of POPSIZE individuals
private static int POPSIZE = 200;
static Chromosome Pop1[];
static Chromosome Pop2[];
static double Fitness[];

// Individuals are sorted by fitness in Order[].
static int Order[];
static private int gen;
static boolean problem = true;

// CONTROL PARAMETERS
```

```
    static double pointMutationRate = 0.5;//per individual
    static double dupliRate = 0.2;//per individual
    static double deleRate = 0.2;//per individual

    //  List contains all   words from {0,1,M,Y}
    //  with 3 or less chars. This is the mother hint.
    static int numbWords;
    static String List[] = new String[85];
    //  Number of hints
    static final int NUMBEROFHINTS = 10;
    // A set of 10 shorter hints:
    static boolean Table[][] = new boolean[NUMBEROFHINTS][85];
    static boolean Tablec[][] = new boolean[NUMBEROFHINTS][85];
    //  Each hint defines its own set of words
    static String Lists[][] = new String[NUMBEROFHINTS][85];
    //  Number of words in each hint
    static int nWords[] = new int[NUMBEROFHINTS];
    static int hint;
    //  score[i] is the fitness of hint i
    static int score[] = new int[NUMBEROFHINTS];
    static int hOrder[] = new int[NUMBEROFHINTS];
    private static boolean print = false;
    private static double leastError = 0;

    // Turn on the random generator
static Random r = new Random();

    private static final long serialVersionUID = 1L;

    //********Declaration of methods*****************
    // All words with less than 3 chars are generated.
    //Alphabet = {0,1,M,Y}
    private static void motherHint() {
        String alphabet[] = {"0", "1", "M", "Y"};
        List[0] = "";

        //strings with one char
        int counter = 1;
        for (int i = 1; i < 5; i++) {
            List[counter] = alphabet[i - 1];
```

```
            counter = counter + 1;
        }

        //strings with two chars
        for (int i = 1; i < 5; i++) {
            for (int j = 1; j < 5; j++) {
                List[counter] = List[i] + alphabet[j - 1];
                counter = counter + 1;
            }
        }

        //    strings with three chars
        for (int i = 5; i < 21; i++) {
            for (int j = 1; j < 5; j++) {
                List[counter] = List[i] + alphabet[j - 1];
                counter = counter + 1;
            }
        }

        //Printing checkout
        System.out.println("The vector with all source strings");
        for (int i = 0; i < 85; i++) {
            System.out.println(i + " " + List[i]);
        }
        System.out.println();
    }

    //  NUMBEROFHINTS daughter hints are generated:
    // Word j of List[] is used in hint i
    //  if and only if Table[i][j]= true
    private static void hints() {
        for (int i = 0; i < NUMBEROFHINTS; i++) {
            int counter = 0;
            while (counter == 0) {
                double p = 0.2;
                for (int j = 0; j < 85; j++) {
                    double s = r.nextDouble();
                    if (s < p) {
                        Table[i][j] = true;
                        counter = counter + 1;
```

```
                } else {
                    Table[i][j] = false;
                }
            }
        }
    }
}

//Each hint i has its own list of words
//with nList[i] elements
private static void hintsCode() {
    for (int i = 0; i < NUMBEROFHINTS; i++) {
        int counter = 0;
        for (int j = 0; j < 85; j++) {
            if (Table[i][j] == true) {
                Lists[i][counter] = List[j];
                counter++;
            }
        }
        nWords[i] = counter;
    }
}

//**************Evolution of binary adders*************
//  A string of stringLength characters is generated
//  The alphabet is {0,1}.
private static String generateStringNumber(int stringLength) {
    String individual = "";
    //An individual is assembled char by char
    for (int j = 1; j <= stringLength; j++) {
        // We generate a random binary number as Character c
        int binary = r.nextInt(2);
        //System.out.println( "binary number = "+ binary );
        char c = (char) (binary + 48);
        //char c is converted into  string s

        String s = "" + c;
        //System.out.println( "binary string = "+ s );
        // s is concatenated to Pop1[i]
        individual = individual + s;
```

```
        }
        //System.out.println( individual);
        return individual;
    }

        private static void printCodon(Codon cd, int i) {
        System.out.println("\n" + i + " cd.first = " + cd.first
                + "  cd.second = " + cd.second
                + "  cd.third = " + cd.third );


        if (cd.first == 0)//replacement
        {
            String a = List[cd.second];
            String b = List[cd.third];
            System.out.println("Replace "  +  a  + " by " + b);


        }
        else
        {
            int k = cd.second;
            String c = List[cd.third];
            System.out.println("Insert " + c
                    + " at place n - " + k );
        }

    }



    private static void printChromosome(Chromosome ch) {
        int n = ch.length;
        System.out.println();
        for (int i = 0; i < n; i++) {
            printCodon(ch.c[i], i);
        }
    }

    private static Codon generateCodon(int numbWords) {
```

```
        //Default initialization
        Codon c = x.new Codon(0, 0, 0);
        //True initialization
        //0(replacement) else 1(insertion)
        c.first = r.nextInt(2);
        if (c.first == 0) {
            c.second = r.nextInt(numbWords);
        }

        if (c.first == 1) {
            c.second = r.nextInt(1000);
        }
        c.third = r.nextInt(numbWords);
        return c;
    }

    private static Chromosome generateChro(int n) {
        Chromosome chr = x.new Chromosome(maxNumbCodons);
        chr.length = n;
        for (int j = 0; j < n; j++) {
            chr.c[j] = generateCodon(numbWords);
        }
        return chr;
    }

    //  An initial population   is set up.
    private static void initialization() {
        numbWords = nWords[hint];
        //Formal declaration of our arrays.
        Pop1 = new Chromosome[POPSIZE];
        Pop2 = new Chromosome[POPSIZE];

        Chromosome chr = x.new Chromosome(maxNumbCodons);
        for (int i = 0; i < POPSIZE; i++) {
            Pop1[i] = chr;
            //n= Number of codons in   chromosome i
            int n = r.nextInt(maxNumbCodons);
            n = Math.max(2, n);
            //System.out.println("NumbCodons = " + n);
            Pop1[i] = generateChro(n);
```

```
            Pop2[i] = chr;
        }

        // Declarations and default initialization
        Order = new int[POPSIZE];
        for (int i = 0; i < POPSIZE; i++) {
            Order[i] = 0;
        }

    }

    //Reads the  number encoded by the binary string a
    // Unprocessed literals are ignored
    private static int decoder(String a) {
        int number = 0;
        int n = a.length();
        for (int j = 0; j < n; j++)
        {
            char c = a.charAt(j);
            switch (c)
            {
                case '0': number = number * 2; break;
                case '1': number = 1 + number * 2;    break;
            }
            //System.out.println(c + " -> " + number);
            //Literals are not caught
        }
        return number;
    }


    //  Posits zeros at the left of the number
    //  until all sites are occupied
    private static String complete(String s) {
        int l = stringLength - s.length();
        for (int i = 1; i < l; i++) {
            s = '0' + s;
        }
        return s;
    }
```

```
// Intercalates input1 with input2
// char by char and posits a separator "Y"
private static String intercalation(String input1
        , String input2) {
    int n = input1.length();
    int m = input2.length();
    int max = Math.max(n, m);
    String output = "";
    for (int i = 0; i < max; i++) {
        output = output + input1.charAt(i)
                + input2.charAt(i) + "Y";
    }
    return output;
}


// This is our genetic code.
// We use four codons

// This is our genetic code:
private static String geneticCode(Codon cd
        , String output) {

    int n = output.length();
    if (n < 120) //avoid extremely large strings
    {
        if (cd.first == 0)//replacement
        {
            String a = List[cd.second];
            String b = List[cd.third];
            //if( (n < 60) && ( a.length() >= b.length()))
            output = output.replaceAll(a, b);
            if (print)
                System.out.println("Replace "  +  a
                    + " by " + b + " -> " + output);
        }

        //Avoid very large strings
        if ((cd.first == 1) && (n < 80))//insertion
```

```
            {
                int k = cd.second;
                String c = List[cd.third];
                n = output.length();
                if (n > 0) {

                    if (n - k > 0) {
                        output = output.substring(0, n - k)
                                + c + output.substring(n - k);
                    }
                } else {
                    output = c;
                }
                if (print)
                    System.out.println("Insert " + c
                        + " at place n - " + k + " -> " + output);
            }
        }

        return output;
    }

    // This is our ribosome
    private static String interpreter(Chromosome chro,
            String output) {
        int n = chro.length;
        for (int i = 0; i < n; i++) {
            Codon c = chro.c[i];
            output = geneticCode(c, output);
        }

        return output;
    }

    //Each individual is tested for how well it adds.
    //Random inputs.
    private static double fitness(Chromosome chr) {
        error = 0;
        /*
            System.out.println("Gen = " + gen + " " +
```

```
            "chr.length = "+chr.length);
        */
        for (int i = 0; i < numberOfTrials; i++) {
            if (print)
            {
                System.out.println("\nTest number " + i);
            }
            String input1 = generateStringNumber(lengthInput);
            input1 = complete(input1);
            String input2 = generateStringNumber(lengthInput);
            input2 = complete(input2);
            nInput1 = decoder(input1);
            nInput2 = decoder(input2);

            String intercalation
                    = intercalation(input1, input2);
            intercalation = complete(intercalation);
            String interpretation
                    = interpreter(chr, intercalation);
            int result = decoder(interpretation);


            //The error is the difference between the correct
            //addition and that proposed by  chro.
            int myError = Math.abs(nInput1 + nInput2 - result);
            if (print)
            {
                System.out.println("\nRESULT OF TEST:"
                    + "\nInput1 = "
                        + input1  + " -> " + nInput1
                    + " \nInput2 = "
                        + input2   + " -> " + nInput2
                    + " \nIntercalation = " + intercalation
                    + " \nInterpretation = " +  interpretation
                    + " \nResult = " + result
                    + " \nError = "
                    + myError);
            }
            error = error + myError;
```

```
    }
if (print) System.out.println("Total error = " + error);
    return Math.abs(error);
}


// Individuals are sorted by fitness
//  The fittest is the first, with minimum error

// Individuals are sorted by fitness
//  The fittest is the first, with minimum error
private static boolean sorting() {
    //System.out.println("sorting");
    problem = true;
    int Champ;
    leastError = 0;

    Fitness = new double[POPSIZE];


    for (int i = 0; i < POPSIZE; i++) {
        Fitness[i] = fitness(Pop1[i]);
        if (print)
            System.out.println("i = " + i
                + " lengthInput = " +lengthInput
                + " Gen = " + gen
                +  " Total Errorrrr = " + Fitness[i]);
    }

    //We sort individuals by fitness
    // Fitness 0 means no errors
    for (int i = 0; i < POPSIZE; i++) {
        Champ = 0;
        for (int j = 0; j < POPSIZE; j++) {
            if (Fitness[j] < Fitness[Champ]) {
                Champ = j;
            }
        }

        //The array Order classifies individuals by  fitness.
```

```
            //the fittest is number zero
            Order[i] = Champ;
            if (i == 0) leastError = Fitness[ Order[0]];
            Fitness[Champ] = 1000000;
        }
        if (print)
            System.out.println("lengthInput = " + lengthInput
                + " Gen = " + gen
                + " Least error"  + leastError
                );
        /*
        //Print job to test sorting procedure, POPSIZE = 5

for(int i = 0; i< POPSIZE;i++)
 System.out.println("Fitness  = " + Fitness[Order[i]]);

System.out.println();
         */
        if (leastError <= ToleratedError)
        {
            problem = false;
            System.out.println(
                    "\nTask ended by generation = " + gen);
            System.out.println(
                    "POPSIZE = " + POPSIZE
                    +"\ngen * POPSIZE = " + gen * POPSIZE
            + "\nNumber of random trials per test = "
                            + numberOfTrials
            + "\nToleratedError = " + ToleratedError);
            System.out.println(
                    "\nCODONS OF THE WINNER CHROMOSOME");
            printChromosome(Pop1[Order[0]]);
            print = true;
            System.out.println(
                    "\nTEST OF THE WINNER WITH A NEW SET "
                            + "OF RANDOOM INPUTS");
            fitness(Pop1[Order[0]]);
            print = false;
        }
        return problem;
```

```
    }


    // The new generation is made
    private static void newGeneration() {
        int i = 0;
        //The fittest individuals are reproduced
        // numbCopies times
        int numbCopies = 10;
        int k = 0;  //rank, 0 is the highest

        while (i < POPSIZE) {
            int counter = numbCopies;
            while ((counter > 0) & (i < POPSIZE)) {
                //System.out.println(i + "" + l);
                Pop2[i] = Pop1[Order[k]];
                //System.out.println(i + " " + counter + " " + k);
                i++;
                counter = counter - 1;
            }
            k++;
        }

        for (int j = 0; j < POPSIZE; j++) {
            Pop1[j] = Pop2[j];
            /*
    * System.out.println( j+ " length at newGen "
                                        + Pop1[j].length);
             */
        }
    }

    // Recombination is enabled
    private static Chromosome rec(int l) {
        //the first part of "l" is copied
        int m1 = 1;
        int m = Pop1[l].length;
        if (m > 0) {
            m1 = r.nextInt(m);
            for (int i = 0; i < m1; i++) {
```

```
                Pop2[l].c[i] = Pop1[l].c[i];
            }
        }
        //The second part of "k" is copied
        int k = r.nextInt(POPSIZE);
        int n = Pop1[k].length;
        if (n > 0) {
            int n1 = r.nextInt(n);
            for (int i = m1; i < m1 + n - n1; i++) {
                if (i < 100) {
                    Pop2[l].c[i] = Pop1[k].c[n1 + i - m1];
                }
            }
        }
        return Pop2[l];
    }

    // The definitive population  is a recombinant one.
    private static void recombination() {
        //Pop2 is a copy of Pop1
        for (int i = 0; i < POPSIZE; i++) {
            Pop2[i] = Pop1[i];
        }

        //Recombination
        for (int l = 0; l < POPSIZE; l++) {
            Pop2[l] = rec(l);
        }
        //Pop2 is transferred to Pop1
        for (int i = 0; i < POPSIZE; i++) {
            Pop1[i] = Pop2[i];
        }
    }

    // Chro g a is mutated at a random codon.
    private static Chromosome pointMutation(Chromosome g) {
        //System.out.println("PointMutation");
        //Which codon must be mutated?
        int l = r.nextInt(g.length);
        //Which part of the codon must be mutated?
```

```
        int m = r.nextInt(3);
        if (m == 0) {
            g.c[l].first = 0;
            g.c[l].second = r.nextInt(numbWords);
        }

        if (m == 1) {
            g.c[l].first = 1;
            g.c[l].second = r.nextInt(1000);
        }

        if (m == 2) {
            g.c[l].third = r.nextInt(numbWords);
        }
        return g;
    }

    private static Chromosome duplication(Chromosome g) {
        //System.out.println("duplication");
        /*
if (g.length ==0)
System.out.println("Old length at dup = " + g.length);
         */
        int l = r.nextInt(g.length);
        int m = r.nextInt(g.length);
        int min = Math.min(l, m);
        int max = Math.max(l, m);
        //System.out.println("Old length at dup = " + g.length);
        //System.out.println("min = " + min + " Max " + max);
        Chromosome chr = g;
        int top = Math.min(2 * max - min, maxNumbCodons);
        for (int i = max; i < top; i++) {
            g.c[i] = chr.c[i - (max - min)];
        }
        int toptop = Math.min(chr.length + max - min, maxNumbCodons);
        if (top < maxNumbCodons) {
            for (int i = top; i < toptop; i++) {
                g.c[i] = chr.c[i - (max - min)];
            }
        }
```

```
        g.length = toptop;
        /*
  * System.out.println("min = " + min +
   "max = " + max +  "top = " + top + "toptop = " + toptop);
         */
        //System.out.println("New length at dup = " + g.length);

        //System.out.println();
        return g;
    }

    private static Chromosome deletion(Chromosome g) {
        //System.out.println("deletion");
        //System.out.println("Old length at del = " + g.length);
        int l = r.nextInt(g.length);
        int m = r.nextInt(g.length);
        int min = Math.min(l, m);
        int max = Math.max(l, m);
        //Deletes from min to max
        for (int i = max; i < g.length; i++) {
            g.c[i - max + min] = g.c[i];
        }

        //System.out.println("Max = " + max + "Min = " + min);
        g.length = g.length - (max - min);
        //System.out.println("New length at del = " + g.length);

        if (g.length < 2) {

            int n = r.nextInt(maxNumbCodons);
            n = Math.max(2, n);
            g = generateChro(n);
        }
        return g;
    }

    // Chromosome g is mutated at a random codon.
    private static Chromosome change(Chromosome g) {
        double p = r.nextDouble();
        if (p < pointMutationRate) {
```

```
            g = pointMutation(g);
        }
        double q = r.nextDouble();
        if (q < dupliRate) {
            g = duplication(g);
        }
        double s = r.nextDouble();
        if (s < deleRate) {
            g = deletion(g);
        }
        return g;
    }

    //  All individuals are mutated
    private static void mutation() {
        for (int i = 0; i < POPSIZE; i++) {
            /*System.out.println( "gen = " + gen);
int l = Pop1[i].length;
 System.out.println( i + "  oldLength at mut = " + l);
             */
            Pop1[i] = change(Pop1[i]);
            /*
int m = Pop1[i].length;
 System.out.println(i + "newLength at mut = " + m);
             */
        }
    }

    // Evolution of binary adders
    private static void evolSolution() {
        newGeneration();
        //The new population is subjected to mutation
        mutation();
        recombination();
        gen++;
        problem = sorting();
        if (gen % 100 == 0) {
            System.out.println("lengthInput = " + lengthInput
                    + " Gen = " + gen + " Least error = " +  leastErro
        }
```

```
    }

    //***********Evolution of hints*************
    // Hints are sorted by fitness
    // The fittest is the first, with max score
    private static void hSorting() {
        //System.out.println("hSorting");
        problem = true;
        int Champ;

        //scorec is a copy of score[]
        //used as workbench
        double scorec[];

        scorec = new double[NUMBEROFHINTS];
        for (int i = 0; i < NUMBEROFHINTS; i++) {
            scorec[i] = score[i];
        }

        //We sort hints by fitness
        for (int i = 0; i < NUMBEROFHINTS; i++) {
            Champ = 0;
            for (int j = 0; j < NUMBEROFHINTS; j++) {
                if (scorec[j] > scorec[Champ]) {
                    Champ = j;
                }
            }

            //The array hOrder classifies individuals by  fitness,
            //the fittest is number zero
            hOrder[i] = Champ;
            scorec[Champ] = 0;
        }
    }

    // The new generation is made
    private static void hNewGeneration() {
        int i = 0; //i = counter of hints
        //The fittest individuals are reproduced
        // numbCopies times
```

```
        int numbCopies = 3;
        int k = 0;  //rank, 0 is the highest

        while (i < NUMBEROFHINTS) {
            int counter = numbCopies;
            while ((counter > 0) & (i < NUMBEROFHINTS)) {
                //System.out.println(i + "" + l);
                for (int j = 0; j < 85; j++) {
                    Tablec[i][j] = Table[hOrder[k]][j];
                }
                //System.out.println(i + " " + counter + " " + k);
                i++;
                counter = counter - 1;
            }
            k++;
        }
        for (int j = 0; j < NUMBEROFHINTS; j++) {
            Table[hint][j] = Tablec[hint][j];
        }
    }

    // Recombination is enabled
    private static void hRec() {
        for (int l = 0; l < NUMBEROFHINTS; l++) {
            //the first part of hint l is copied
            int m1 = r.nextInt(85);
            for (int i = 0; i < m1; i++) {
                Tablec[l][i] = Table[l][i];
            }

            //The second part of hint k is copied
            int k = r.nextInt(NUMBEROFHINTS);
            int n1 = r.nextInt(85);
            for (int i = m1; i < m1 + 85 - n1; i++) {
                if (i < 85) {
                    Tablec[l][i] = Table[k][n1 + i - m1];
                }
            }
        }
    }
```

```
// The definitive population  is a recombinant one.
private static void hRecombination() {
    //Tablec is a copy of Table
    for (int i = 0; i < NUMBEROFHINTS; i++) {
        for (int j = 0; j < 85; j++) {
            Tablec[i][j] = Table[i][j];
        }
    }

    //Recombination
    hRec();
    //Tablec is transferred to Table
    for (int i = 0; i < NUMBEROFHINTS; i++) {
        for (int j = 0; j < 85; j++) {
            Table[i][j] = Tablec[i][j];
        }
    }
}

// Each hint is mutated at a random entry.
private static void hPointMutation() {
    for (int i = 0; i < NUMBEROFHINTS; i++) {
        int j = r.nextInt(85);
        Table[i][j] = !Table[i][j];
    }
}

private static void evolveHints() {
    hNewGeneration();
    hPointMutation();
    hRecombination();
    hSorting();
    gen++;
}

public static void main(String[] args) {
    /*int l = decoder("100111");
    System.out.println(l);*/
    /*
```

```
int seed = 12345;
r.setSeed(seed);
System.out.println("Seed = " + seed);
 */
//The mother hint is generated
motherHint();
//Ten daughter hints are generated.
hints();
hintsCode();
//Loop of evolution of hints
int hintsGen = 1000;

for (int a = 1; a <= hintsGen; a++) {
    System.out.println();
    System.out.println("**************");
    System.out.println("**************");
    System.out.println("GEN  OF EVOLUTION OF HINTS= " + a);
    //For each hint,
    //a new separated evolutionary branch of
    //binary adders is run.

    for (int i = 0; i < NUMBEROFHINTS; i++) {
        hint = i;
        System.out.println();
        System.out.println("**************");
        System.out.println("hint number = " + hint);
        numbWords = nWords[i];//number of words in hint i
        initialization();//binary adders are initialized
        //a number of generations is given to each hint
        int maxGen = 7000;
        lengthInput = 1;
        numberOfTrials = 10 * lengthInput;
        System.out.println("lengthInput = " + lengthInput);
        gen = 1;
        //Individuals are sorted by fitness
        problem = sorting();
        //Loop of evolution of binary adders
        while ((problem) & (gen < maxGen)) {
            evolSolution();
            if (problem == false) {
```

```
                        lengthInput++;
                        numberOfTrials = 10 * lengthInput;
                        problem = true;
                        System.out.println();
                        System.out.println("lengthInput = " + lengthInput);
                    }
                }
                //The score of a hint is the length of resolved inputs.
                score[hint] = lengthInput - 1;
                System.out.println("hint = " + hint + " score "
                        + score[hint]);
            }

            evolveHints();
        }
    }//End of main

}//End of Program TwoLevelHier, B127
```

128 ***Exercise*** *Run the program and play with the code. Play enough to agree else disagree with the claim of the Author: in the scale of an impatient developer, the program is useless as software evolutionary synthesizer.*

8.2 Three level hierarchy

Our experiment of the last section shows us that to tame evolution to design software is not a simple deal. Possibly, our previous programs with a two level hierarchy or those with no hierarchy at all would do the job if enough time could be provided. The problem is that time of cosmological order would be necessary for moderate length inputs. This is the meaning of a combinatorial explosion. So, our aim is to achieve concrete results in no later than some hours of computing work over a desktop, say, less than 100.

Let us try out a three level hierarchy to see what we achieve. Why do we prefer to try a three level hierarchy rather than to run the two level hierarchy during some 100 hours? It is because program BinaryAdderTrans, B112, that posits the problem of designing a binary adder at one step from being solved, worked with an organization that fits a three level hierarchy: a master method call subsidiary methods (exterior hierarchy), each of which consists of some few replace or insert clauses (intermediate hierarchy), whose arguments are taken from a mother combinatorial basis (inner hierarchy).

Our three level hierarchy implements modular design of software: instead of dealing with elementary instructions directly, a strategy that leads into a combinatorial explosion, one deals with modules or methods or subsets of instructions to form reduced alphabets with potential evolutionary efficacy.

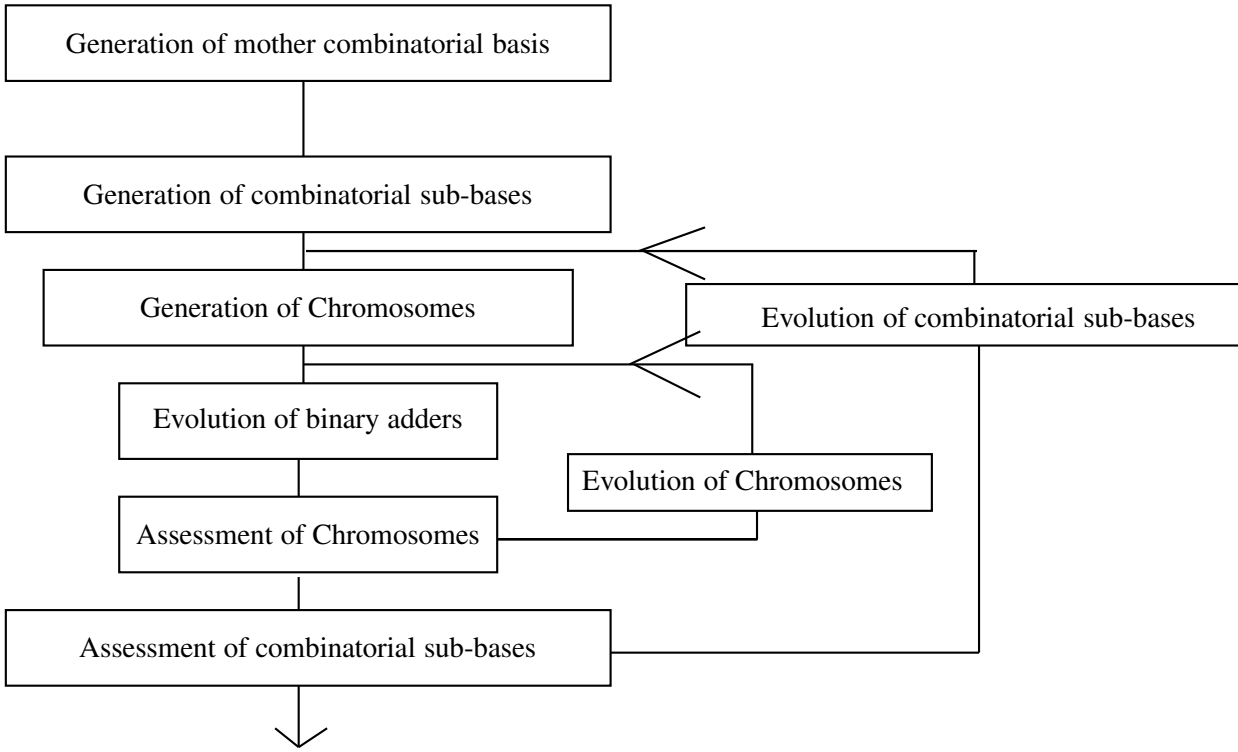


Figure 8.0. *The algorithm that obeys a three level hierarchy for the design of a parallel binary adder.*

- The elementary clauses of all binary adders are either replacement or insertion instructions. They are our codons.
- All possible arguments of elementary clauses are taken from a mother combinatorial basis. Some subsets are taken from it to define specific combinatorial sub-bases. Each sub-basis generates its own evolutionary process. A combinatorial basis plays the role of a hint for the artificial intelligence simulated by evolution. The whole hierarchy is a superhint.

- A string of elementary clauses or codons is generated at random, whose arguments are taken from a given combinatorial sub-basis. This is the equivalent of methods of program BinaryAdderTrans, B112. They are the chromosomes whose length is 10 codons as maximum. For each combinatorial sub-basis, 26 chromosomes are generated at random and are denoted by an upper case letter, A, B, ..., Z.

- A presumably binary adder is a string of chromosomes: HDABC. These are the individuals of our evolutionary environment. They play the role of a master method that calls other ones.

- Evolution is allowed at all levels. There is evolution of combinatorial sub-bases, of chromosomes, and of individuals that must function as binary adders.

Dear Reader: our enterprise is very tedious. So, it is a good idea to make it clear that software is a tool that can be used without limit of time or space: software is a tool for eternity. So, any effort is nothing in comparison with resultant benefices.

129 ***Our code*** *The next code implements evolution at a three level hierarchy:*

```
/*Program ThreeLevelsHier, B129
We continue the study of the evolution of combinatorial bases
to enable evolution to design a
parallel binary adder.
A three level hierarchy is implemented.


- The elementary clauses of all binary adders
are either replacement or insertion instructions.
They are our codons.
- All possible  arguments of elementary clauses
are taken from a mother
combinatorial basis. Some subsets are taken from it to
define specific combinatorial sub-bases.
Each sub-basis generates its own evolutionary process.
A combinatorial basis plays the role of a hint for
the artificial intelligence simulated by evolution.
The whole hierarchy is a superhint.
```

```
- A string of elementary clauses or codons
is generated at random,
whose arguments are taken from
a given  combinatorial sub-basis.
This is the equivalent of methods of
program BinaryAdderTrans, B112. They are the
chromosomes  whose length is 10 codons maximum.
For each combinatorial sub-basis, 26 chromosomes are
generated at random and are denoted
by an upper case letters, A, B, ..., Z.
- An individual, a presumably binary adder, is
a string of chromosomes: HDABC. These
are the individuals of our evolutionary environment.
They play the role of a
master method that calls other methods
with executive clauses.

Evolution is allowed at all levels.
There is evolution of   combinatorial sub-bases,
of chromosomes, and of individuals.
 */
package ejvol2v2p;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class ThreeLevelsHier {

    public ThreeLevelsHier() //constructor
    {
    }

    // x = Instance
    static ThreeLevelsHier x = new ThreeLevelsHier();

    //****Declaration of new types as inner classes**
    //New type Codon
    class Codon {
```

```
    private int first; //0=replacement, 1=insertion
    private int second; //address of word
    private int third; //address of word or place in a word
    //A codon can be defined given some integers
    Codon(int i, int j, int k) {
        first = i;
        second = j;
        third = k;
    }

    //A codon can be copied from another codon g
    Codon(Codon g) {
        first = g.first;
        second = g.second;
        third = g.third;
    }

}

//New type Chromosome
//A chromosome is an array of codons.
//Chromosome is synonymous of individual.
class Chromosome {

    private int length;
    private Codon c[];

    //Everything in Java must be initialized
    //otherwise, an nillpointer error will be generated.
    //Default initialization to zero
    public Chromosome(int n) {
        length = n;
        Codon cd = new Codon(0, 0, 0);
        c = new Codon[n];
        for (int i = 0; i < n; i++) {
            c[i] = cd;
        }
    }

    //A chromosome can be
```

```
    //copied as a clone from another one.
    public Chromosome(Chromosome g) {
        length = g.length;

        for (int i = 0; i < length; i++) {
            c[i] = g.c[i];
        }
    }
}



//******Declaration of variables*********

//For a true run put demo = false;
private static boolean demo = true;

//For exact fitting, pose ToleratedError = 0;
//You might need 10 years to accomplish your work.
private static int ToleratedError = 2;//Change by 1 or 0
//  Number of ciphers in all of our strings
private static int stringLength = 7;



//  Number of ciphers or bits in the inputs
private static int lengthInput;



//********Level one: individuals**********
//  Pop1 and Pop2 are populations of individuals
private static String Pop1[];
private static String Pop2[];

//  Max number  of letters at the beginning in an individual
private static int maxNumbLetters = 10;
// Individuals are sorted by fitness in Order[].
private static int OrderInd[];
private static double FitnessInd[];
```

```
//******* Level two: chromosomes *******
//  A chromosome is  string of codons.
//  A codon is a replacement or insertion instruction.
//  We will have a population of chromosomes with
//  a number of nChros individuals.
// NUMBCHROS <= 26, the number of upper letters
private static final int NUMBCHROS = 26;
private static Chromosome Chro1[];
private static Chromosome Chro2[];
//  The fitness of chromosomes
private static double FitnessChro[]
        = new double[NUMBCHROS];
private static double FitnessChroc[]
        = new double[NUMBCHROS];
// Chromosomes are sorted by fitness in OrderChro[].
private static int OrderChro[];
//  There are some few codons per chromosome
private static int maxNumbCodons = 10;

//***********Level three: combinatorial bases*************

//Number of words of subbasis number i
private static int numbWordsBi;
//  All  possible words from {0,1,M,Y}
//with 3 or less chars.
private static String wordsOfMotherCombBasis[]
        = new String[85];

private static int indexOfSubBasis;
//  Number of combinatorial sub-bases
private static final int NUMBOFSUBBASES = 1000;

private static boolean Table[][]
        = new boolean[NUMBOFSUBBASES][85];
private static boolean Tablec[][]
        = new boolean[NUMBOFSUBBASES][85];
//  Each combinatorial basis i defines
//its own set of words j..
private static String wordsOfSubBasis[][]
```

```
        = new String[NUMBOFSUBBASES][85];
//  Number of words in each combinatorial basis
private static int nWords[]
        = new int[NUMBOFSUBBASES];

//  The max length of resolved input
private static int maxLengthInput;
//  fitnessH[i] is the fitness of combinatorial basis i
private static int fitnessH[]
        = new int[NUMBOFSUBBASES];
//  Combinatorial bases are ordered by fitness in OrderH[]
private static int OrderH[] = new int[NUMBOFSUBBASES];

private static int gen;
private static boolean problem = true;
private static boolean printSolution = true;
private static boolean print = true;

// CONTROL PARAMETERS
//  Number of individuals: greater than 10.
private static int POPSIZE = 10;
private static double pointMutationRate = 0.5;
private static double dupliRate = 0.2;
private static double deleRate = 0.2;
//  Number of generations given to the
//evolution of combinatorial bases
private static int combBasisGen = 10;
//  Number of generations given to
//the evolution of chromosomes
private static int chroGen = 30;
//Number of generations given to
//the evolution of individuals
private static int indivGen = 20;

// Turn on the random generator
private static Random r = new Random();

private static final long serialVersionUID = 1L;

//******** Printing facility ***************
```

```
    private static void printCodon(int indexOfSubBasis
            , Codon cd, int codonNumber) {
        System.out.println("\n" + codonNumber + " cd.first = "
                + cd.first
                + "  cd.second = " + cd.second
                + "  cd.third = " + cd.third );


        if (cd.first == 0)//replacement
        {
            String a
                = wordsOfSubBasis[indexOfSubBasis][cd.second];
            String b
                = wordsOfSubBasis[indexOfSubBasis][cd.third];
            System.out.println("Replace "  +  a  + " by " + b);

        }
        else
        {
            String word = wordsOfSubBasis[indexOfSubBasis][cd.second];
            int k = cd.third;

            System.out.println("Insert " + word
                    + " at place n - " + k );
        }
    }



    private static void printChromosome(int indexOfSubBasis
            , Chromosome ch) {
        int n = ch.length;

        for (int i = 0; i < n; i++) {
            printCodon(indexOfSubBasis, ch.c[i], i);
        }
    }
```

```
private static void printIndividual(int indexOfSubBasis
        , String s)
{

    System.out.println("THIS IS THE COMBINATORIAL SUB-BASIS "
        + indexOfSubBasis);



    for (int j = 0; j < numbWordsBi; j++) {

            System.out.print(
                    wordsOfSubBasis[indexOfSubBasis][j]
                    + "   " );
    }
    System.out.println();


    System.out.println("\nTHIS IS INDIVIDUAL = " + s);
    int n = s.length();
    for (int i = 0; i < n; i++) {
        char ch = s.charAt(i);
        System.out.println("\nLIST OF CODONS "
                + "OF CHROMOSOME = " + ch);
        int number = charToNumber(ch);
        printChromosome(indexOfSubBasis, Chro1[number]) ;
    }
}


 private static void printCombinatorialSubbasis(int indexOfSubBasi
{

    System.out.println("THIS IS THE COMBINATORIAL SUB-BASIS "
        + indexOfSubBasis);



    for (int j = 0; j < numbWordsBi; j++) {
```

```
            System.out.print(
                    wordsOfSubBasis[indexOfSubBasis][j]
                    + "   " );
    }
    System.out.println();

}

////******** Combinatorial Bases ***************




//Generation of words that will be the arguments
//of replacement and insertion clauses.
//Generation of the mother combinatorial basis:
//All words with less than 3 chars are generated.
//Alphabet = {0,1,M,Y}
private static void wordsOfMotherCombBasis() {
    String alphabet[] = {"0", "1", "M", "Y"};
    //To enable deletions
    wordsOfMotherCombBasis[0] = "";

    //strings with one char
    int counter = 1;
    for (int i = 1; i < 5; i++) {
       wordsOfMotherCombBasis[counter] = alphabet[i - 1];
        counter = counter + 1;
    }

    //strings with two chars
    for (int i = 1; i < 5; i++) {
        for (int j = 1; j < 5; j++) {
           wordsOfMotherCombBasis[counter]
                   = wordsOfMotherCombBasis[i]
                   + alphabet[j - 1];
            counter = counter + 1;
        }
```

```
    }

    // Strings with three chars
    for (int i = 5; i < 21; i++) {
        for (int j = 1; j < 5; j++) {
           wordsOfMotherCombBasis[counter]
                   = wordsOfMotherCombBasis[i]
                   + alphabet[j - 1];
            counter = counter + 1;
        }
    }

    //Printing checkout
    System.out.println("WORDS OF THE MOTHER "
            + "COMBINATORIAL BASIS");
    for (int i = 0; i < 85; i++) {
        System.out.println(i + " "
                + wordsOfMotherCombBasis[i]);
    }
    System.out.println();
}


private static int distinctRandomInt(List presentNumbers)
{
    int i = r.nextInt(85);
    //System.out.println(i + "  " + d);
    if (presentNumbers.contains(i))
            distinctRandomInt(presentNumbers);
    else return i;
   return 0;
}



private static int distinctRandomInt3(int i)
{
    int newRandom = r.nextInt(85);
    //System.out.println(i + "  " + d);
    for(int j = 0; j < 85; j++)
```

```
          if (Table[i][newRandom] == true)
                distinctRandomInt3(i);

       return newRandom;
    }


    //  Generation of daughter combinatorial bases.
    //  NUMBOFSUBBASES daughter combinatorial bases
    //  are generated:
    //  Word j of wordsOfMotherCombBasis[] is used
    //  in combinatorial subbasis i
    //  if and only if Table[i][j]= true
    private static void combSubBasis() {
        List presentNumbers = new ArrayList();
        for (int i = 0; i < NUMBOFSUBBASES; i++)
        {
            //Initialization with no elements
            for (int j = 0; j < 85; j++)
                Table[i][j] = false;

            // 0 < Number of elements in Basis < 25
            //20 were shown to be enough.
            int howMany = r.nextInt(24) +1;


            //Elements are chosen and marked
            for (int k = 0; k < howMany; k++)
            {
                int indexChosenWord
                        = distinctRandomInt(presentNumbers);
                presentNumbers.add(indexChosenWord);
                Table[i][indexChosenWord] = true;
            }

            presentNumbers.clear();
            //Elements are  orderly added to subBasis i
            int counter = 0;
            //System.out.println("ELEMENTS OF SUBBASIS " + i + " -> " +
nWords[i]);
```

```
            for (int j = 0; j < 85; j++)
            {

                if ( Table[i][j] == true)
                {
                    wordsOfSubBasis[i][counter]
                        = wordsOfMotherCombBasis[j];
                    //System.out.println(counter + "   " +
wordsOfMotherCombBasis[j]);
                    counter++;
                }

            }
            nWords[i] = counter;
            //System.out.println();
        }



        for (int i = 0; i < NUMBOFSUBBASES; i++)
        {
            System.out.println("ELEMENTS OF SUBBASIS " + i
                    + " -> " + nWords[i] + " words");
            for (int j = 0; j < nWords[i]; j++)
                System.out.println( j + "    " +  wordsOfSubBasis[i][j
);
            System.out.println();
        }

    }


    //************ CODONS AND CHROMOSOMES **********



    private static Codon generateCodon(int numbWordsBi) {
        //Default initialization
        Codon c = x.new Codon(0, 0, 0);
        //True initialization
```

```
        //0(replacement) else 1(insertion)
        c.first = r.nextInt(2);
        //Replace
        if (c.first == 0) {
            //Replace what
            c.second = r.nextInt(numbWordsBi);
            //by what
            c.third = r.nextInt(numbWordsBi);
        }


        //Insert
        if (c.first == 1) {
            //index of chosen word
            c.second = r.nextInt(numbWordsBi);
            //where: at n - k (k = 0 or  1 or 2)
            c.third = r.nextInt(3);

        }
        return c;
    }


    private static Chromosome
                generateChro(int indexOfSubBasis, int n) {
        Chromosome chr = x.new Chromosome(maxNumbCodons);
        chr.length = n;
        for (int j = 0; j < n; j++) {
            chr.c[j] = generateCodon(nWords[indexOfSubBasis]);
        }
        return chr;
    }


    // An initial population  of chromosomes is set up.
    //This is subordinated to a combinatorial sub-basis.
    // Each chromosome has various replacement
    //and insertion clauses.
    //It plays the role of a method in progem
    //Program FullParallelBinaryAdder, B107.
```

```
private static void initialChros(int indexOfSubBasis) {

    //Formal declaration of our arrays.
    Chro1 = new Chromosome[NUMBCHROS];
    Chro2 = new Chromosome[NUMBCHROS];

    Chromosome chr = x.new Chromosome(maxNumbCodons);
    for (int i = 0; i < NUMBCHROS; i++) {
        Chro1[i] = chr;
        //n= Number of codons in   chromosome i
        int n = r.nextInt(maxNumbCodons);
        n = Math.max(2, n);
        //System.out.println("NumbCodons = " + n);
        Chro1[i] = generateChro(indexOfSubBasis, n);
        Chro2[i] = chr;
    }

    // Declarations and default initialization
    OrderChro = new int[NUMBCHROS];
    for (int i = 0; i < NUMBCHROS; i++) {
        OrderChro[i] = 0;
    }

}

//******Initialization of individuals ************
//A genotype is a succession of calls to chromosomes.
//Chromosomes appear in Chro1[],
//Individuals appear in Pop1[].
//Each genotype encodes for a binary adder.
//A genotype looks like this: BCDFADBFD
//Next, each letter is converted into a number, with A=0,
//and the corresponding chromosome is called for.
// A string of stringLength characters is generated
// The alphabet is {A,B,C,..,Z}.
private static String generateString(int n) {
    String individual = "";
    //An individual is assembled char by char
    for (int j = 1; j <= n; j++) {
        // We generate a random  number less than 26
```

```
            int numb = r.nextInt(26);
            //System.out.println( "Number = "+ numb);
            //numb defines a letter
            char c = (char) (numb + 65);
            //Character c is converted into  string s
            String s = "" + c;
            // s is concatenated to Pop1[i]
            individual = individual + s;
        }
        //System.out.println( individual);
        return individual;
    }

    // An initial population  of individuals is set up.
    private static void initialInd() {
        //Formal declaration of our arrays.
        Pop1 = new String[POPSIZE];
        Pop2 = new String[POPSIZE];

        for (int i = 0; i < POPSIZE; i++) {
            //n= Number of letters in Pop1[i]
            int n = r.nextInt(maxNumbLetters) + 1;
            for (int j = 0; j < n; j++) {
                Pop1[i] = generateString(n);
                Pop2[i] = Pop1[i];
            }
        }

        // Declarations and default initialization
        OrderInd = new int[POPSIZE];
        for (int i = 0; i < POPSIZE; i++) {
            OrderInd[i] = 0;
        }
    }


    //Reads the decimal number in  string a
    //Reads the  number encoded by the binary string a
    // Unprocessed literals are ignored
    private static int decoder(String a) {
```

```
        int number = 0;
        int n = a.length();
        for (int j = 0; j < n; j++)
        {
            char c = a.charAt(j);
            switch (c)
            {
                case '0': number = number * 2; break;
                case '1': number = 1 + number * 2;    break;
            }
            //System.out.println(c + " -> " + number);
            //Literals are not caught
        }
        return number;
    }



    //  Posits zeros at the left of the number
    //  until all sites are occupied
    private static String complete(String s) {
        int l = stringLength - s.length();
        for (int i = 0; i < l; i++) {
            s = '0' + s;
        }
        return s;
    }

    // Intercalates input 1 with input2
    // char by char and posits a separator "Y"
    private static String intercalation(String input1
            , String input2) {
        int n = input1.length();
        int m = input2.length();
        input1 = complete(input1);
        input2 = complete(input2);
        String intercalation = "";
        for (int i = 0; i < stringLength; i++) {
            intercalation = intercalation + input1.charAt(i)
```

```
                + input2.charAt(i) + "Y";
        }


        return intercalation;
    }




    //***********Evolution of individuals*******************
    // Char c is decoded into a number: 'A' is zero.
    private static int charToNumber(char c) {
        String s1 = "" + c;
        int number = 0;
        for (int j = 0; j < 26; j++) {
            char d = (char) (j + 65);
            String s2 = "" + d;
            if (s1.equals(s2)) {
                number = j;
            }
        }
        return number;
    }


    // This is our genetic code.
    // We use four codons
    private static String geneticCodeCodon(int indexOfSubBasis
            , Codon cd,
            String interpretation) {

        int n = interpretation.length();

        if (cd.first == 0)//replacement
        {
            String a = wordsOfSubBasis[indexOfSubBasis]
                                       [cd.second];
            String b = wordsOfSubBasis[indexOfSubBasis]
                                       [cd.third];
```

```
            /*
                System.out.println("\nInterpretation = "
                        + interpretation);
                System.out.println("cd.second = "
                        + cd.second);
                System.out.println("cd.third = "
                        + cd.third);
                System.out.println("a = " + a);
                System.out.println("b = " + b);
            */

            //Avoid very long strings
            if ((n < 70) ) {
                interpretation = interpretation.replaceAll(a, b);
            }
        }

        if (cd.first == 1)//insertion
        {
            //System.out.println(indexOfSubBasis + "   " +  cd.second)
            String word = wordsOfSubBasis[indexOfSubBasis][cd.second];
            if (n > 0) {
                int k = cd.third % n;
                if (n - k > 0) {
                    interpretation = interpretation.substring(0, n - k)
                            + word + interpretation.substring(n - k);
                }
            }
        }
        //printCodon(cd);
        //System.out.println("interpretation = " + interpretation);
        return interpretation;
    }

    // This is our ribosome
    private static String interpreterChro(int indexOfSubBasis
            , Chromosome chro,
            String intercalation) {
        int n = chro.length;
        String interpretation = intercalation;
```

```
        for (int i = 0; i < n; i++) {
            Codon codon = chro.c[i];
            interpretation = geneticCodeCodon(indexOfSubBasis
                           , codon, interpretation);

        }
        return interpretation;
    }


    //  This is the genetic code for individuals.
    //  Char chr is converted into a number. A is 0.
    //  That number points to a chromosome.
    private static String geneticCodeChro(
            int indexOfSubBasis, char chr, String intercalation) {
        int number = charToNumber(chr);
        String interpretation = interpreterChro(indexOfSubBasis
                , Chro1[number], intercalation);

        return interpretation;
    }


    //An individual is a string of choromosomes
    private static String interpreterInd(int indexOfSubBasis
            , String indiv,
            String intercalation) {
        int n = indiv.length();
        String interpretation = intercalation;
        for (int i = 0; i < n; i++) {
            char chr = indiv.charAt(i);

            interpretation = geneticCodeChro(indexOfSubBasis
                    , chr, interpretation);

        }
        return interpretation;
    }
```

```
//Each individual is tested for how well it adds.
static private long assessInd(int indexOfSubBasis
                              , String indiv) {

    int upTo = (int) Math.pow(2, lengthInput);
    long totalError = 0;

    for (Integer i = 0; i < upTo; i++)
        for (Integer j = 0; j < upTo; j++)
        {
            if (print)
            {
                System.out.println("\nTest of binary addition " + i+ "
            }
            long myError = 0;
            String input1 = Integer.toBinaryString(i);
            //input1 = complete(input1);
            String input2 = Integer.toBinaryString(j);

            //input2 = complete(input2);
            int nInput1 = decoder(input1);
            int nInput2 = decoder(input2);
            /*
            System.out.println("\nNUMBERS TO BE ADDED:"
                    + "\nInput1 = "
                        + input1  + " -> " + nInput1
                    + " \nInput2 = "
                        + input2   + " -> " + nInput2);
            */
            String intercalation = intercalation(input1
                                        , input2);
            String interpretation
                    = interpreterInd(indexOfSubBasis
                            , indiv, intercalation);

            //System.out.println();
            int result = decoder(interpretation);

            //The error is the difference between the correct
```

```
            //addition and the output proposed by indiv.
            myError = Math.abs(nInput1 + nInput2 - result);
            if (myError < 0 ) myError = - myError;
            if (print)
            {
                System.out.println("\nRESULT OF TEST:"
                    + "\nInput1 = "
                        + input1  + " -> " + nInput1
                    + " \nInput2 = "
                        + input2   + " -> " + nInput2
                    + " \nIntercalation = " + intercalation
                    + " \nInterpretation = " +  interpretation
                    + " \nResult = " + result
                    + " \nError = " + myError);
            }
            totalError = totalError +  myError;
        }
    if (print)
    {
        System.out.println("Gen = " + gen + " Total Error = "
                                        + totalError);
    }

    return totalError;
}


static private void printInd(int indexOfSubBasis
                             , String indiv) {
    System.out.println("Task ended by generation = " + gen);
    System.out.println("POPSIZE = " +   POPSIZE);
    System.out.println("Number of trials = "  + gen * POPSIZE);
    System.out.println("Solution = "
            + indiv);
    if (printSolution)
            printIndividual(indexOfSubBasis
                    , indiv);
    print = true;
    System.out.println(
            "\nTEST OF THE WINNER "
```

```
            + "\nToleratedError = " + ToleratedError);
        assessInd(indexOfSubBasis, indiv);
        print = false;
        if (demo) System.exit(0);
    }

        // Individuals are sorted by fitness.
        // the fittest is the first, with minimum error
        private static boolean sortingInd() {
            int indexChamp;
            String champion = "";
            long leastError = 0;
            //FitnessIndc[] is a copy of FitnessInd[]
            //used as workbench
            double FitnessIndc[];
            FitnessInd = new double[POPSIZE];
            FitnessIndc = new double[POPSIZE];
            for (int i = 0; i < POPSIZE; i++) {
                FitnessInd[i] = 1000000;
                FitnessIndc[i] = 1000000;
            }
            print = false;
            for (int i = 0; i < POPSIZE; i++) {
                FitnessInd[i] = assessInd(indexOfSubBasis
                        , Pop1[i]);
                FitnessIndc[i] = FitnessInd[i];
            }
            //We sort individuals by fitness
            //Fitness 0 means no errors
            for (int i = 0; i < POPSIZE; i++) {
                indexChamp = 0;
                for (int j = 0; j < POPSIZE; j++) {
                    if (FitnessIndc[j] < FitnessIndc[indexChamp]) {
                        indexChamp = j;
                    }
                }
                //The array Order classifies individuals by  fitness.
                //The fittest is number zero
                OrderInd[i] = indexChamp;
                if (i == 0)
```

```
            {
                leastError = (long) FitnessInd[ OrderInd[0]];
                champion = Pop1[OrderInd[0]];
            }
            FitnessIndc[indexChamp] = 1000000000;
        }
        //Printing job to test sorting procedure, POPSIZE = 5
        /*for(int i = 0; i< POPSIZE;i++)
System.out.println("Fitness  = " + FitnessInd[ i] );
for(int i = 0; i< POPSIZE;i++)
System.out.println(" Rank   " + i +" = "+ OrderInd[i]);

for(int i = 0; i< POPSIZE;i++)
System.out.println("Fitness  = " + FitnessInd[OrderInd[i]]);

System.out.println();
         */

        //¡¡ Large numbers may turn into negative values!!
        if ((leastError <= ToleratedError) && (leastError >= 0))
        {
            problem = false;
            System.out.println("Least Error = " + leastError );
            printInd(indexOfSubBasis, champion);
        }


        return problem;
    }



    // The new generation is made
    private static void newGenerationInd() {
        int i = 0;
        //The fittest individuals are reproduced
        // numbCopies times
        int numbCopies = 10;
        int k = 0;  //rank, 0 is the highest
```

```
        while (i < POPSIZE) {
            int counter = numbCopies;
            while ((counter > 0) & (i < POPSIZE)) {
                //System.out.println(i + "" + l);
                Pop2[i] = Pop1[OrderInd[k]];
                //System.out.println(i + " "
                //+ counter + " " + k);
                i++;
                counter = counter - 1;
            }
            k++;
        }
        for (int j = 0; j < POPSIZE; j++) {
            Pop1[j] = Pop2[j];
            /*System.out.println( j+ " length at newGen "
                                    + Pop1[j].length);
             */
        }
    }


    //Recombination is enabled
    private static String recInd(int l) {
        String s = Pop1[l];
        String s1 = "";
        int m = s.length();
        if (m > 0) {
            int m1 = r.nextInt(m);
            s1 = s.substring(0, m1);
        }

        int k = r.nextInt(POPSIZE);
        String t = Pop1[k];
        int n = t.length();
        String t2 = "";
        if (n > 0) {
            int m2 = r.nextInt(n);
            t2 = t.substring(m2);
        }
```

```
        //Recombination
        return s1 + t2;
    }

    // The definitive population  is a recombinant one.
    private static void recombinationInd() {
        //Pop1 is a recombinant version of Pop2
        for (int l = 0; l < POPSIZE; l++) {
            Pop2[l] = recInd(l);
        }
        //Pop2 is transferred to Pop1
        for (int i = 0; i < POPSIZE; i++) {
            Pop1[i] = Pop2[i];
        }
    }

    // This method generates random chars
    private static char randomChar() {
        // We generate a random   number less than 5
        int numb = r.nextInt(3);
        //System.out.println( "Number = "+ numb );
        char c = (char) (numb + 65);
        return c;
    }

    // String a is mutated at a random place.
    // Method taken from weasel by Max Scott
    private static String pointMutationInd(String a) {
        //System.out.println("Old a" + a);
        int l = r.nextInt(a.length());
        //System.out.println(l);
        String z = ((l <= 0) ? "" : a.substring(0, l))
                + randomChar()
                + ((l >= (a.length() - 1)) ? ""
                        : a.substring(l + 1));
        // System.out.println("Point Mutated a" + z);
        return z;
    }

    // A substring sub inside a is duplicated;
```

```
    //sub is chosen at random.
    private static String duplicationInd(String a) {
        //System.out.println("old a = " + a);
        if (a.length() == 1) {
            a = a + a;
        } else {
            int l = r.nextInt(a.length());
            int m = r.nextInt(a.length());
            int min = Math.min(l, m);
            int max = Math.max(l, m);
            String sub = a.substring(min, max);
            //System.out.println(l + " " + m);
            a = a.substring(0, max)
                    + sub + a.substring(max);
        }
        //System.out.println("Duplication in a = " + a);
        return a;
    }

    private static String deletionInd(String a) {
        //System.out.println("old a = " + a);
        int l = r.nextInt(a.length());
        int m = r.nextInt(a.length());
        int min = Math.min(l, m);
        int max = Math.max(l, m);
        //Deletes from min to max
        String z = a.substring(0, min) + a.substring(max);
        return z;
    }

    // Individuals are mutated
    private static void mutationInd() {
        for (int i = 0; i < POPSIZE; i++) {
            double p = r.nextDouble();
            if (p < pointMutationRate) {
                Pop1[i] = pointMutationInd(Pop1[i]);
            }
            double q = r.nextDouble();
            if (q < dupliRate) {
                Pop1[i] = duplicationInd(Pop1[i]);
```

```
            }
            double s = r.nextDouble();
            if (s < deleRate) {
                Pop1[i] = deletionInd(Pop1[i]);
            }
            while (Pop1[i].equals("")) {
                Pop1[i] = generateString(maxNumbLetters);
            }
            //Truncation
            Pop1[i] = Pop1[i].substring(0, Math.min(20
                                , Pop1[i].length()));

        }
    }

    // Given a goal, it is guessed by evolution
    private static boolean evolBinAdders() {
        //The new generation is produced
        //as copies of the fittest
        newGenerationInd();
        recombinationInd();
        mutationInd();

        problem = sortingInd();

        if (gen % 10 == 0) {
            System.out.println("lengthInput = "
                    + lengthInput
                    + " Gen = " + gen
                    + " Error  = "
                    + FitnessInd[OrderInd[0]]);
        }
        return problem;
    }

    //***********Evolution of  chromosomes*********


    //  A good chromosome is one that is called by one of the
    //  top ten binary adders, which are listed in OrderInd[].
```

```
    private static void assessChros() {
        //System.out.println("assessChro");
        for (int i = 0; i < NUMBCHROS; i++) {
            FitnessChro[i] = 0;
        }

        int n = Math.min(10, POPSIZE);
        for (int i = 0; i < n; i++) {
            int m = Pop1[OrderInd[i]].length();

            for (int j = 0; j < m; j++) {
                char c = Pop1[OrderInd[i]].charAt(j);
                int l = charToNumber(c);
                //Chromosome l is good;
                FitnessChro[l]++;
            }
        }

        //FitnessChroc[] is a copy of FitnessChro[]
        //used as workbench
        for (int i = 0; i < NUMBCHROS; i++) {
            FitnessChroc[i] = FitnessChro[i];
        }

        /*
    System.out.println( );
          */
     }

     //Chros are sorted by fitness
     //The larger the fitness the better is the chro
     private static void sortingChros() {
        //System.out.println("sorting");
        problem = true;
        int Champ;
        assessChros();
        for (int i = 0; i < NUMBCHROS; i++) {
            Champ = 0;
            for (int j = 0; j < NUMBCHROS; j++) {
                if (FitnessChroc[j] > FitnessChroc[Champ]) {
```

```
                Champ = j;
            }
        }

        //The array Order classifies individuals by  fitness,
        //the fittest is number zero
        OrderChro[i] = Champ;
        FitnessChroc[Champ] = 0;
    }
}

//The new generation is made
private static void newGenerationChro() {

    //The fittest individuals are reproduced
    // numbCopies times
    int numbCopies = 10;
    int k = 0;  //rank, 0 is the highest
    int i = 0;
    while (i < NUMBCHROS) {
        int counter = numbCopies;
        while ((counter > 0) & (i < NUMBCHROS)) {
            //System.out.println(i + "" + l);
            Chro2[i] = Chro1[OrderChro[k]];
            /*System.out.println(i + " "
                + counter + " " + k);*/
            i++;
            counter = counter - 1;
        }
        k++;
    }

    for (int j = 0; j < NUMBCHROS; j++) {
        Chro1[j] = Chro2[j];
        /*
* System.out.println( j+ " length at newGen "
                                + Chro1[j].length);
         */
    }
}
```

```
//Recombination is enabled
private static Chromosome recChro(int l) {

    //the first part of "l" is copied
    int m1 = 1;
    int m = Chro1[l].length;
    //The second part of "k" is copied
    int k = r.nextInt(NUMBCHROS);
    int n = Chro1[k].length;
    int min = Math.min(m,n);


    if (min > 0) {
        m1 = r.nextInt(min);
        for (int i = 0; i < m1; i++) {
            Chro2[l].c[i] = Chro1[l].c[i];
        }
    }

    if (n > 0) {

        for (int i = m1; i < n ; i++) {
            if (i < maxNumbCodons) {
                Chro2[l].c[i] = Chro1[k].c[i];
            }
        }
    }
    Chro2[l].length = n;
    return Chro2[l];
}


//The definitive population  is a recombinant one.
private static void recombinationChro() {
    //Recombination
    for (int l = 0; l < NUMBCHROS; l++) {
        Chro2[l] = recChro(l);
    }
    //Chro2 is transferred to Chro1
```

```
        for (int i = 0; i < NUMBCHROS; i++) {
            Chro1[i] = Chro2[i];
        }
    }

    //Chro g a is mutated at a random codon.
    private static Chromosome pointMutationChro(Chromosome g) {
        //System.out.println("PointMutation");
        //Which codon must be mutated?
        int l = r.nextInt(g.length);
        //Which part of the codon must be mutated?
        int m = r.nextInt(3);
        if (m == 0) //Swap replacement and insertion
        {
            if (g.c[l].first == 0) //Replacement
            {
                g.c[l].first = 1;//turn to Insertion
                //where: at n - k (k = 0 or  1 or 2)
                g.c[l].third = r.nextInt(3);
            }
            else //Turn to replacement
            {
                g.c[l].first = 0;
                g.c[l].third = r.nextInt(numbWordsBi);
            }

        }

        if (m == 1) //change second, a word
        {

            g.c[l].second = r.nextInt(numbWordsBi);
        }

        if (m == 2) //change third
        {
            if (g.c[l].first == 0) //Change word by word
               g.c[l].third = r.nextInt(numbWordsBi);
            else//Change place of insertion by a new one
                g.c[l].third = r.nextInt(3);
```

```
        }
        return g;
    }


    private static Chromosome duplicationChro(Chromosome g) {
        //System.out.println("duplication");
        /*
if (g.length ==0)
System.out.println("Old length at dup = " + g.length);
         */
        int l = r.nextInt(g.length);
        int m = r.nextInt(g.length);
        int min = Math.min(l, m);
        int max = Math.max(l, m);
        //System.out.println("Old length at dup = " + g.length);
        //System.out.println("min = " + min + " Max " + max);
        Chromosome chr =  g;
        int top = Math.min(2 * max - min, maxNumbCodons);
        for (int i = max; i < top; i++) {
            g.c[i] = chr.c[i - (max - min)];
        }
        int toptop = Math.min(chr.length + max - min
                , maxNumbCodons);
        if (top < maxNumbCodons) {
            for (int i = top; i < toptop; i++) {
                g.c[i] = chr.c[i - (max - min)];
            }
        }
        g.length = toptop;
        /*
* System.out.println("min = " + min +
 "max = " + max +  "top = " + top
            + "toptop = " + toptop);
         */
        //System.out.println("New length at dup = " + g.length);

        //System.out.println();
```

```
        return g;
    }

    private static Chromosome deletionChro(Chromosome g) {
        //System.out.println("deletion");
        //System.out.println("Old length at del = " + g.length);
        int l = r.nextInt(g.length);
        int m = r.nextInt(g.length);
        int min = Math.min(l, m);
        int max = Math.max(l, m);
        //Deletes from min to max
        for (int i = max; i < g.length; i++) {
            g.c[i - max + min] = g.c[i];
        }

        //System.out.println("Max = " + max + "Min = " + min);
        g.length = g.length - (max - min);
        //System.out.println("New length at del = " + g.length);

        if (g.length < 2) {

            int n = r.nextInt(maxNumbCodons);
            n = Math.max(2, n);
            g = generateChro(indexOfSubBasis, n);
        }
        return g;
    }

    //Chromosome g is mutated at a random codon.
    private static Chromosome changeChro(Chromosome g) {
        double p = r.nextDouble();
        if (p < pointMutationRate) {
            g = pointMutationChro(g);
        }
        double q = r.nextDouble();
        if (q < dupliRate) {
            g = duplicationChro(g);
        }
        double s = r.nextDouble();
        if (s < deleRate) {
```

```
            g = deletionChro(g);
        }
        //Truncation
        g.length = Math.min(20, g.length);
        return g;
    }

    // All individuals are mutated
    private static void mutationChro() {
        for (int i = 0; i < NUMBCHROS; i++) {
            /*System.out.println( "gen = " + gen);
int l = Chro1[i].length;
 System.out.println( i + "  oldLength at mut = " + l);
             */
            Chro1[i] = changeChro(Chro1[i]);
            /*
int m = Chro1[i].length;
 System.out.println(i + "newLength at mut = " + m);
             */
        }
    }

    //Evolution of Chromosomes
    private static void evolChros() {
        newGenerationChro();
        //The new population is subjected to mutation
        mutationChro();
        recombinationChro();
        sortingChros();
    }

    //**********Evolution of combinatorial bases***********
    // Combinatorial bases are sorted by fitness
    // The fittest is the first, with max fitnessH
    private static void sortingCB() {
        //System.out.println("sortingH");
        problem = true;
        int Champ;

        //fitnessHc is a copy of fitnessH[]
```

```
        //used as workbench
        double fitnessHc[];

        fitnessHc = new double[NUMBOFSUBBASES];
        for (int i = 0; i < NUMBOFSUBBASES; i++) {
            fitnessHc[i] = fitnessH[i];
        }

        //We sort combinatorial bases by fitness
        for (int i = 0; i < NUMBOFSUBBASES; i++) {
            Champ = 0;
            for (int j = 0; j < NUMBOFSUBBASES; j++) {
                if (fitnessHc[j] > fitnessHc[Champ]) {
                    Champ = j;
                }
            }

            //The array OrderH classifies individuals by  fitness.
            //The fittest is number zero
            OrderH[i] = Champ;
            fitnessHc[Champ] = 0;
        }
    }

    //The new generation is made
    private static void newGenerationCB() {
        int i = 0; //i = counter of combinatorial basiss
        //The fittest individuals are reproduced
        // numbCopies times
        int numbCopies = 3;
        int k = 0;  //rank, 0 is the highest

        while (i < NUMBOFSUBBASES) {
            int counter = numbCopies;
            while ((counter > 0) & (i < NUMBOFSUBBASES)) {
                //System.out.println(i + "" + l);
                for (int j = 0; j < 85; j++) {
                    Tablec[i][j] = Table[OrderH[k]][j];
                }
                i++;
```

```
                counter = counter - 1;
            }
            k++;
        }


        for (int m = 0; m < NUMBOFSUBBASES; m++)
            for (int n = 0; n < 85; n++)
        {
            Table[m][n]
                    = Tablec[m][n];
        }
    }

    //Recombination of tables is enabled
    private static boolean[][] recCB() {
        for (int l = 0; l < NUMBOFSUBBASES; l++) {
            //the first part of combinatorial basis l is copied
            int m1 = r.nextInt(85);
            for (int i = 0; i < m1; i++) {
                Tablec[l][i] = Table[l][i];
            }

            //The second part of combinatorial basis k is copied
            int k = r.nextInt(NUMBOFSUBBASES);

            for (int i = m1; i <  85; i++) {
                if (i < 85) {
                    Tablec[l][i] = Table[k][  i ];
                }
            }
        }
        return Tablec;
    }


    /*
    //Recombination is enabled
    private static Chromosome recChro(int l) {
```

```
        //the first part of "l" is copied
        int m1 = 1;
        int m = Chro1[l].length;
        //The second part of "k" is copied
        int k = r.nextInt(NUMBCHROS);
        int n = Chro1[k].length;
        int min = Math.min(m,n);


        if (min > 0) {
            m1 = r.nextInt(min);
            for (int i = 0; i < m1; i++) {
                Chro2[l].c[i] = Chro1[l].c[i];
            }
        }

        if (n > 0) {

            for (int i = m1; i < n ; i++) {
                if (i < maxNumbCodons) {
                    Chro2[l].c[i] = Chro1[k].c[i];
                }
            }
        }
        Chro2[l].length = n;
        return Chro2[l];
    }
    */
    //The definitive population  is a recombinant one.
    private static void recombinationCB() {


        //Recombination
        Tablec = recCB();
        //Tablec is transferred to Table
        for (int i = 0; i < NUMBOFSUBBASES; i++) {
            for (int j = 0; j < 85; j++) {
                Table[i][j] = Tablec[i][j];
            }
        }
```

```
    }

    //Each combinatorial basis is mutated at a random entry.
    private static void pointMutationCB() {
        for (int i = 0; i < NUMBOFSUBBASES; i++) {
            int j = r.nextInt(85);
            Table[i][j] = !Table[i][j];
        }
    }


    //Performace of combinatorial bases
    //is evaluated through chromosomes, intermediate hierarchy.
    private static void performanceEvalulationOfCB()
    {
        for (int i = 0; i < NUMBOFSUBBASES; i++) {
            //number of words in combinatorial basis i
            numbWordsBi = nWords[i];
            indexOfSubBasis = i;
            System.out.println();
            System.out.println("**************");
            System.out.println("Combinatorial basis  number = "
                    + indexOfSubBasis);


            printCombinatorialSubbasis(indexOfSubBasis);
            initialChros(indexOfSubBasis);
            evolveChromosomes();


            //The fitness of a combinatorial basis is
            //the maximum length of
            //resolved input.
            fitnessH[indexOfSubBasis] = maxLengthInput - 1;
            System.out.println("combinatorial basis = "
                    + indexOfSubBasis + " fitnessH = "
                    + fitnessH[indexOfSubBasis]);
        }
    }
```

```
//Combinatorial bases evolve
private static void evolveCombinatorialSubBases() {
    performanceEvalulationOfCB();
    sortingCB();
    newGenerationCB();
    pointMutationCB();
    recombinationCB();
}



//Loop of evolution of individuals (binary adders)
private static void evolveIndividuals() {

    while ((problem) && (gen <= indivGen)) {
        problem = evolBinAdders();
        gen++;
        if (problem == false) {
            lengthInput++;
            problem = true;
            System.out.println();
            System.out.println("lengthInput = "
                    + lengthInput);
        }

    }
}


//Loop of evolution of chromosomes
private static void evolveChromosomes()
{
    //Performance is evaluated through individuals (binary adders)
    for (int b = 0; b < chroGen; b++) {
        System.out.println();
        System.out.println("Gen of Chros = " + b);
        initialInd(); //Individuals are initialized
        evolveIndividuals();
```

```
            System.out.println("lengthInput = "
                    + lengthInput);
            gen = 1;
            //Individuals are sorted by fitness
            problem = sortingInd();

            if (lengthInput > maxLengthInput) {
                maxLengthInput = lengthInput;
            }
            // A good chromosome is one that
            // is called by one of the
            // top ten binary adders,
            // which are listed in OrderInd[].
            sortingChros();

        }
        evolChros();
    }




    //General loop of evolution:
    //Evolution of combinatorial bases is visible,
    //it is the outer hierarchy
    private static void evolution(){
        for (int a = 0; a < combBasisGen; a++) {
            System.out.println();
            System.out.println("**************");
            System.out.println("**************");
            System.out.println( "\nCOMBINATORAL BASIS,  "
                            + "GENERATION  = "  + a);
            //For each combinatorial basis,
            //a new separated evolutionary branch of
            //chromosomes and
            //binary adders is run.
            //maxlengthInput resolved by the  combinatorial basis
            maxLengthInput = 0;
```

```
            //Allowed number of generations for the Bases loop
            // More time is given to advanced stages of evolution
            indivGen = indivGen + a * 500;
            evolveCombinatorialSubBases();
        }
    }


    public static void main(String[] args) {
        /*int l = decoder("100111");
            System.out.println(l);*/
           /*
           int seed = 12345;
           r.setSeed(seed);
           System.out.println("Seed = " + seed);
         */
        //The mother combinatorial basis is generated
        wordsOfMotherCombBasis();
        //Generation of daughter combinatorial subbases.
        combSubBasis();
        //We begin by adding combinations of 0 and 1
        lengthInput = 1;
        evolution();
    }//End of main

}//End of Program ThreeLevelsHier, B129
```

130 ***Exercise*** *Run the code and study its possibilities as software synthesizer. This is a time consuming program so, if you fall short of time and want to see a demo, put* `toleratedError = 2;` *and* `demo = true;`*. With more time you can set* `toleratedError = 1;` *or* `toleratedError = 0;`*. If you have plenty of time, and want to explore additions of binary numbers with 2 or more bits, set* `demo = false;`

8.3 In hindsight

With the purpose of proving that evolution is a software developer we have devised various programs that use evolution to develop the code for a binary adder. Let us add some reflections about this theme.

131 *A half cooked product*

The Author has verified that the program `ThreeLevelsHier, B129` can correctly produce the code to add binary numbers with one bit, i.e., it correctly adds 0 + 0, 1 + 0, 0 + 1 and 1 + 1. But because his impatience he has been unable to see the code for the binary addition with two bits, say, 11 + 10. So, a doubt remains: is our failure due to a bug? Our instinctive reaction is to deny the doubt and to point to the postulated intrinsic complexity of the problem. One ingredient is that we fall prey of a combinatorial explosion and the other is that seemingly the combinations that do the job are too few in number. So, the combinatorial explosion cannot be effectively circumvented, a belief that seems to be valid for the binary adder that adds numbers of one bit. In a case like this we speak of `irreducible complexity`, a concept that makes sense in relation with the employed evolutionary environment.

Let us delve a bit into the employed strategy to try to circumvent the encountered combinatorial explosion.

132 *Alphabets, super-alphabets and grammars*

Genetic programming is the design of software by means of simulated evolution. At the outset we thought that evolution corresponded with text processing that can be reduced to copy-paste. Battling with a combinatorial explosion we decided to change our mind and concluded that a new ingredient was needed: the capability to process if-instructions. All that was naturally contained in the so called grammars. More formally:

A **simulation of evolution** begins with the definition of an **alphabet**, which is a set of letters of the English alphabet. The simulation continues with the definition of a **grammar**, i.e., of a set of replacement rules to process substrings. In our case, the input for the grammar consists of two binary numbers and the grammar processes the input to get the correct output, the addition of the two numbers. Grammars are tested for how well they add and reproduced according to its fitness.A new turn begins and so on until the criterion for halting is satisfied.

The strategy to design Java software by means of evolution consists in arming evolution with a battery of Java methods that define the **alphabet**. A set of methods is suitable to solve a problem when a concatenation of some of its members may form a functional unit, program or class that solves the problem and such that there are irrelevant functional redundancy to avoid unnecessary magnification of combinatorial explosions.

Instead of alphabets, one can also use superalphabets. A **super-alphabet** is an alphabet whose letters represent strings or words of the alphabet: a super-alphabet implements recursion over methods.

Example: We have four Java methods, $\{A,B,C,D\}$ in our alphabet. A string could be ABBCA, another ABBBBCCCCACACACDCACAC. A string represents a Java program or class. A super-alphabet is an alphabet, say, $\{X,Y,W,Z\}$ but each letter represents a substring composed with letters of the alphabet, say:

$\{X = AAB; Y = ABBCDCD; W = DBDCDA; Z = CADBCADCADA\}$.

So, a super-alphabet is indeed a library of Java programs.

Our simulations have shown us that because of exaggerated combinatorial explosions the use of an alphabet composed of elementary instructions as the raw shuffling material for evolution is not a viable strategy to pursue genetic programming. By contrast, the use of **super-alphabets** seems to be technologically reliable, at least in some cases, although this method is also expensive specially in computing time.

Since the design of software is a sign of intelligence, we can say that evolution functions as an artificial intelligence while suitable alphabets and super-alphabets function as hints and we have proved that evolution can, up to some point, devise its own hints. Certainly, we as yet have not been enabled evolution to devise super-alphabets in a completely independent way: some know-how must be inserted at the very beginning. In our case, that was done with the mother alphabet plus the code of interpretation, what corresponds in molecular biology to the genetic code.

133 *The importance of super-alphabets*

We proved that evolution is capably of developing the code for a binary adder if we pose the problem at one step from being solved, i.e., if we use an appropriate super-alphabet. Is this something special, unique to our approach?

Let us share right now the experiences obtained in regard with super-alphabets in other areas related with computation: linguistics, computing science, electronics, biotechnology and genomics: the use of super-alphabets corresponds with modular structure and design.

Linguistics: the alphabet is the alphabet of the language, say, A,B,C..Z, the super-alphabet is composed of roots that are combined to form words. For instance, the word `etymology` is derived from two Greek roots: `etymon = true sense`, and `logia = the study of`. So, words represent super-super-alphabets. Words are used to compose sentences but libraries of sentences figure more like a curiosity than as a matter of surviving. People with high IQ move amidst the super-alphabet of roots and the super-super-alphabet of words with some incursions into the realm of libraries of sentences. The elementary alphabet is dealt with by specialists, say, in phonetics.

Computing science: a computer receives instructions in machine language, which is very simple indeed. Nevertheless, we cannot imagine today a computer without an operational system. What is that? It is a library of programs, written in machine code, that fulfill specific functions that look simple for the user but that might be very complex for the computer. So, an operational system is a super-alphabet built over the alphabet of the machine language. Computers are useful in ordinary life thanks to super-alphabets.

Electronics: the fundamental alphabet of solid state technology is the transistor. The corresponding super-alphabet is composed of chips, i.e. of crafts that contain complete circuits with various transistors and other elements, say, resistances and capacitances, in the same silicon plate. Today, anyone can assemble a computer using a processor that is contained in a chip with some hundreds of millions of transistors. Thus, one can say that in electronics, organismic evolution can proceed at fast pace thanks to modular design at the scale of super-super-.. ..- super-alphabets. But, chips are not easy to invent: they are produced by very competitive companies whose job is to use the elementary alphabet of transistors to assemble functional units. One problem with this methodology is the extremely high quality of standardization in the synthesis processes needed to make complex circuits to fulfill a predefined function. A very modern alternative consists in evolvable hardware (Higuchi et al, [28] 1997). It is a fault tolerant strategy that uses a simulation of evolution over real electronic circuits: its various elementary units are not composed of transistors but of circuits that calculate, say, polynomials or trigonometric functions. Thus, their success is also boosted by super-alphabets.

Biotechnology: when genetic programming is specialized to protein or DNA design in the laboratory, it is called **lab evolution**. The elementary alphabet is composed of codons or by aminoacids. Evolution over the alphabet of codons or aminoacids is a factual possibility today but this technique seems to be superseded by evolution over super-alphabets, whose letters are composed of strings of codons or **schemes**. These super-alphabets are called in biotechnology libraries and techniques for producing effective libraries are already at work (Bacher et al, [4] 2002).

Genomics: The alphabet is composed of bases. Codons appear as components of a superalphabet, which in turn can integrate a higher superalphabet of exons, pieces of DNA that are expressed and translated in toto. The DNA structure of living beings is best described by a reshuffling of exons and genes and not by a recombination of codons. A super-super-alphabet composed of genes (associations built from various exons and circumventing material) is also a good candidate to

explain part of the DNA structure of living creatures. The hottest version of this fact is given by the unveiling of various chimeras (Furla et al, [23] 2005; Raymond et al, [42] 2003; Goldenfeld at al, l.c., [25] 2007).

By looking at these similarities across so diverse fields, we conclude that the excessive combinatorial explosions caused by the use of alphabets are indeed a tremendous load in various areas of culture, industry and in nature altogether, but super-alphabets or modules do it much better because we expect evolution aimed at the assembling of high level functional structures to proceed more swiftly.

134 ***Exhortation***

An alphabet plus a genetic code for interpretation plus a form to try out diverse super-alphabets is called an **evolutionary environment**. If one has an evolutionary environment plus infinite computing power and time, evolution can be run automatically.

Because the immanent exasperating restrictions of time and raw material, the election of effective alphabets and super-alphabets is a fine and delicate art reserved for people and companies with great expertise because most alphabets and super-alphabets lead to nowhere else are very effective but without expressive power, so they are good for a concrete task but not for solving slightly different problems. So, evolution functions like an artificial intelligence that tries to solve a problem and good evolutionary environments represent hints given from above: evolution without effective hints is just noise and smoke. Or equivalently, effective hints transform evolution from noise and smoke into an effective artificial intelligence.

One always cherishes the desire that evolution could alleviate the hard work demanded by design. Our experience shows that we are very far from that state of development. So, alphabets and super-alphabets must be devised with great care otherwise nothing is achieved. This can be vividly seen in the history of electronics leading to the design and assembling of modern computers:

First computers were made with vacuum tubes but because of exaggerated needs in space and energy they represent what is known as an evolutionary blind alley. The exit from that black hole came after the invention of the transistor in 1947 by John Bardeen and Walter Brattain, working at Bell Telephone Laboratories. The transistor is the mother of a new alphabet, whose singular importance was recognized by the Academy with the Nobel Prize in Physics of 1956. Transistors can form a super-alphabet in various ways, say, printed circuits. But these also are ineffective for a very large design. The idea for building a really effective super-alphabet was invented in 1958 and 1959 by Jack Kilby, at Texas Instruments, and Robert Noyce, at Fairchild Camera, who proposed the integrated circuit that could

be as small as a transistor but with complete circuits with various transistors inside and therefore it could have many legs, unlike the transistor that has three. The grandiosity of this idea was awarded the Nobel Prize in Physics of 2000. You can see more details of this amazing history in Internet at nobelprize.org ([38] 2008). In modern computers one can find chips that contain micro-processors with some hundreds of millions of transistors and with a density of one million transistors per square centimeter. These circuits with very large scale integration (VLSI) are the letters of the high level super-alphabets of modern electronics.

The corresponding situation in biotechnology is sincerely embarrassing: apart from some exceptions, the raw material for the super-alphabets of modern lab evolution has been stolen from living beings. This has happened as follows: proteins and enzymes fulfill specific functions in living beings. But, in general, the function of a complete enzyme results as the coherent work of subunits that can be given in a consistent way a specific sub-function. These subunits are called **domains**. Now, how can recombination and mutation work without disturbing function? It is because many proteins admit variations and therefore the important functional concept is not that of protein but of the scheme behind them (Voigt et al, [50] 2002). Now, the raw material for enzyme evolutionary biotechnology consists mostly in schemes that encode for protein domains of extant living beings (Joern et al, l.c., [30] 2002). Soil bacteria have enough variability in functional domains so that they will remain forever as the fundamental source of libraries of schemes and domains for every kind of medical, industrial or ecological needs. Nevertheless, biotechnology is so wide that in many works bacteria are also used as letters of alphabets in some evolutionary quests. All this is a matter of intense research.

Thus, the state of the art of evolutionary programming be it in genetic programming, electronics, computing science or in biotechnology is reminiscent of the situation described by the words that close the Origin of Species:

'There is grandeur in this view of life, with its several powers, having been originally breathed into a few forms or into one; and that, whilst this planet has gone cycling on according to the fixed law of gravity, from so simple a beginning endless forms most beautiful and most wonderful have been, and are being, evolved.'

We see in this description that evolution is promised to be capable of many wonders if a good beginning is correctly set up. The good beginning is nowadays composed of a good evolutionary environment with a good alphabet and a very good super-alphabet together with an excellent genetic code. The problem is that good evolutionary environments must be breathed from above. But contrary to Darwin, we are anymore ready to accept that our medical, industrial or ecological

problems will be solved just by firing evolution and waiting during some time: the load of combinatorial explosions heavily pressures over our back. In more clear words, modern world is urged by the need of every kind of proficient professionals in all areas of applied evolution that could dress evolution in suitable hints.

Our exhortation needs a further explanation: evolution is extremely demanding in every kind of resources: time, memory, processors, and design effort. So, why one must worry about evolution if one can go around with suitable heuristics? The reason is this: heuristics might solve a problem but evolution can sharply improve performance: the new output could be thousands of times better than the initial one.

135 *On the scarcity of genetic codes*

The genetic code is the set of rules to translate the information of RNA strands into polypeptide sequences. To achieve this, it assigns an amino acid to codons, say:

UUU $--> $ Phe

We see that the genetic code can be modeled as a set of replacement rules so, from the syntactic point of view, the **genetic code** is a `grammar`. Therefore, our exercise with a three level hierarchy is indeed a simulation of the evolution of genetic codes. This hierarchy was devised with the intention of killing a combinatorial explosion and we are sure that we succeed because we were able to see how the program can develop the code for a binary adder of numbers one bit long, a fact that was absent with other designs. Anyway, because of a lack of patience (or maybe a bug), we have not as yet seen the arising of adders of numbers two bits long. So, we consider that the achieved reduction of the combinatorial explosion is important but anyway unreachable.

What is the case in Biology?

Let us see:

All aminoacids have the same head, $NH_2 - (CHR) - COOH$, but different R, radical. So, the number of potential aminoacids diverges towards infinite because one can put there as many radicals as desired. Nevertheless, there are only some 500 known aminoacids, the half of which occur free in nature (Wagner et al, [51] 1983) while 20 go in the genetic code of the nuclei of our cells. Now and in relation with combinatorial explosions, how many genetic codes with 20 elements can be built?

$$\binom{250}{20} = \frac{250!}{20! \times 230!} > \frac{231^{20}}{20^{20}} > 11^{20} > 10^{20}$$

So, these big numbers together with the fact that the evolution of grammars is what we are doing every day induce us to expect a great variability of genetic codes in nature plus the ensuing variability in forms of life and evolutionary trees. It is the same that happens with human languages: similar languages are grouped in families, but there are families that have nothing in common apart from cultural contamination, say, the caucasic language Georgian has nothing in common with Russian, which is an slavic language, and both are very different than English, a language that belongs in the germanic group.

So, what? How many genetic codes are there in nature? There are less than 30 (Elzanowski and Ostell, [18] 1996). An immediate question springs: How can we pass from 10^{20} to 30? There are two types of weapons to try put. We have functional constraints in first place. Next, we have population genetics effects.

Additionally, we can advocate that any change in the genetic code implies a change in a significant proportion of proteins, many of which are monomorphic. From this stand, evolution of the natural genetic code seems an unexpected marvel so, evolution must be found to be severely restricted, if any. What we do find? On one hand, biotechnology has shown that the natural genetic code can be extended (Mehl et al, [33] 2003) so, evolution must be really possible. On the other hand, we share with most species the same nuclear genetic code although mitochondria, chloroplasts along with some other few species have variations that are peacefully incorporated into a nice tree-like graph with soft changes (Weberndorfer, [53] 2002). If we take this fact at its face value, we must predict that our grammar approach to GP is a desperate option and we must get content with very humble results. If this is proven to be true, everyone has a license to claim that life was designed by a very smart Creator. That is why we must call population genetics to witness:

What can we do to accommodate results into the evolutionary program of accumulation of small change along a landscape with just one big summit? To begin with, we can depart from a simple beginning, whatever, and scale towards complexity thanks to a powerful selection factor:

"Starting with very simple codes, and hence greatly reduced amino acid alphabets, we observe a diversification of the codes in most simulation runs. The driving force behind this effect is the possibility to produce fitter proteins when the repertoire of amino acids is enlarged." (Weberndorfer, l.c., [53] 2002).

Anyway, the beginning and the evolutionary path can even be specified up to some degree (Wong, [60] 1975). Moreover, the final state must not be a surprise:

the genetic code of alive beings appears to be finely tunned to minimize the effects of mistranslation and point mutation (Freeland et al, l.c., [21] 1998). Besides, homogenizing processes could had been working even in the prebiotic era (Vetsigian et altri, [49] (2006).

We has seen how easy has been for evolution to explain the observed facts of nature in regard with the genetic code. Or maybe, have been we cheating? Most probably it is so: we interpret our personal failures in the study of the evolution of grammars as a direct evidence of the extreme complexity of this problem. In fact, the design of a genetic code is a difficult problem: it is a multilevel, multiobjective optimization problem (compare, Bollenbach et al, [7] 2007; Gilis et al, [24] 2001). Now, we know quite well that achieved solutions to complex problems are in general numerous, varied, quite different one from another and far from being optimal. Nothing like that is seen on Earth. This is clearly a falsification of the Evolutionary Theory and of its extensions to the origin of life.

8.4 Conclusion

From the syntactic point of view, the genetic code is a grammar. So, the correct frame of the question of the evolution of the genetic code is the evolution of grammars. We have been working with this problem in relation with the evolutionary synthesis of the code for a binary adder. We succeeded in our task of taming evolution to have it designing the Java code for a binary adder if we pose the problem at one step of being solved. Nevertheless, we have proposed that a three level hierarchy evolutionary environment is necessary to solve the problem when it is less restricted and this choice allowed us to see evolution successfully developing the code for a binary adder of numbers one bit long. But for two bits a combinatorial explosion made prey of us (although bugs as primary causes of our failure cannot be excluded). So, our position in regard with the evolution of the genetic code is as follows: it is not excluded that the problem could be depicted by a landscape with one dominant summit but since the principal factor could be co-adaptation, many more hills shall be present making the evolutionary rate to be very low. Hence, were evolution to cause of life, that evolution must be observable today with the presence of thousand of suboptimal genetic codes on earth. But this is not found. This amounts to a falsifying of the Evolutionary Theory and of its extension to cover the origin of life.

Chapter 9

Conclusions

```
Verified:  Evolution is a software developer.
```

1. The teaching of Darwin about evolution plus the proposals of Mendel that received plain justification with the discovery of DNA and genes can be summarized in the following form: evolution is an automatic mechanism that enables 1) random mutation leading to variability and 2) accumulation of those small random changes that are beneficial for surviving and reproduction (Dawkins, 1986).

2. A new insight has been maturing since the 1930s and it is so important that it could be the fundamental contribution of our time to the understanding of evolution. That insight reads: the genome is software and evolution is a software developer. Software is composed of a set of strings in an environment that contains a code of interpretation, an interpreter and an executer that can associate to each string a specific action. The genome is the natural example of software: the code of interpretation is the genetic code, the interpreter is the ensemble of t-RNA's, and the primary executer is the ribosome that assembles proteins.

3. The Evolutionary Theory now reads: evolution is the software developer that is responsible for the genome.

4. One difference between the vision of evolution as accumulation of small change and evolution as software developer is that according to the new vision the primary target of evolution is to design tools not things. This is due to the fact that software programs are tools to process information or to solve problems. Additionally, software is possibly the most sophisticated

and difficult tool to devise because minor variations might produce no function else change it completely. With this insight, we hope that we at last have made justice to Biologists that instinctively consider that life is the summit of marvels.

5. This new insight, that evolution is both a software developer (fact) and the responsible for the genome (theory), has arrived to us through a rich history, whose minimal version might include the next points:

   (a) Everyone uses fingers and pebbles to count. Commerce impulsed the construction of refined helping machines: the oldest surviving counting board is the Salamis tablet, used by the Babylonians circa 300 B.C., discovered in 1846 on the island of Salamis (Wikipedia, [56] 2016). Incas had their yupana (Wikipedia, [57] 2016).

   (b) Charles R. Darwin proposes in 1859 (l.c., [12]) his theory on the origin of species by means of natural selection and of adaptation by means of accumulation of small change.

   (c) The British mathematician and inventor Charles Babbage worked out in the 19th century the principles of the modern digital computer. Alan Turing and John von Neumann worked out around 1930 the mathematical fundamentals of computation such as they are known and applied today. A truly digital computer was built around 1940 by Howard Aiken, a Harvard University mathematician. The ENIAC, Electronic Numerical Integrator And Computer, was built by John Presper Eckert, Jr. and John William Mauchly in 1946 (Encarta, [19] 1994).

   (d) Friedberg ([22] 1958) used a blind random search in the space of programs to find a program performing certain task as adding two bits.

   (e) Arthur Samuel ([45] 1959) expressed the need of enabling computers with the possibility of learning from experience.

   (f) The spatial structure of DNA is worked by Watson and Crick ([52] 1953) showing that DNA was the perfect candidate to be the carrier of the inheritable genetic information.

   (g) Marshal Nirenberg ([37] 1961, 1964) deciphers the genetic code showing that the genome is software.

   (h) John Holland ([29] 1975) proves that the theory of evolution encloses a methodology of automatic learning that is applicable to the generality of adaptive systems. So, evolution is anymore a patrimony of nature but a tool to be used in ordinary problem solving.

(i) John Koza (l.c., [32] 1980) publishes his *Genetic programming*. That work showed that the design of software by means of evolution is eventually more of the same evolutionary game. He highlights the importance of making sure that evolution is better than randomness for the design of software. He verifies that this is so for simple problems but his studies transmit a negative feeling about high levels of complexity.

(j) A DNA computer specifically prepared to solve an abstract mathematical program was built and run by Leonard Adleman ([1] 1994).

(k) DNA universal computers (with the possibility of solving every kind of computable abstract problems) have been imagined (Beaver, [6] 1996) and it is now clear that DNA **universal computation** possibly never would reach the industrial production.

(l) In the 1990ies, ARPANET, the Defense Department's inter-computer system, morphed into the National Science Foundation's network and then it was released to the public as Internet (Shurkin, [47] 2008).

(m) The most crucial point menacing the security of the communication among computers (a frontal attack upon hosting processors) was solved in 1995 with the adoption of the Java virtual machine concept developed by Sun Microsystems (Deitel, [13] 2004). The code for Java was opened by 2006.

(n) The evolJava community is born by 2009: this is a community of persons, not of computers, committed to undertake the study of evolution both conceptually and operationally with the possibility of running Java simulations on a word wide scale.

6. The developer of software could be a conscious being, like a human, or an unconscious process, like evolution. If the conscious being has a purpose, we speak of design. Evolution has no purpose, so it is not a designer, but acquires an emergent purpose: to augment differential surviving and reproduction. Therefore, all what we know about software design is also valid for natural evolution.

7. There are computer programs that need no input to get functioning, say, a program that prints a welcome message. But others programs process inputs that are usually restricted to certain bounds and they fail if those bounds are violated. The genome of alive beings belongs in this last class: if we consider the enzymes as a fair projection of the genome, then we can say that our organism process vegetables and meat but fails to process plastics or soil. For this reason, the genome per se is nothing and contains no life in

itself. Instead, life needs an appropriate co-adaptation of the genome and the milieu, both extracellular and intracellular.

8. Human software cannot be imagined if not as recursive calling to certain units. This is so compelling that the electing and use of libraries of well tested software is considered as one of the elementary duties of a developer. Biological software observes the same tendency: duplications of parts of the genome abound everywhere (Wen-Hsiung, l.c., [54] 1997).

9. If evolution is the software developer that is responsible for the origin of species, we must be able to use evolution to design software. This is the founder idea of GP (Genetic programming). This is a first mandatory test. To that aim, we have tested two theories. The first is that evolution corresponds with text processing that can be reduced to copy-paste. Battling with a combinatorial explosion we decided to change our mind and concluded that a new ingredient was needed: the capability to process if-instructions. This second theory is naturally implemented by the so called grammars, which are sets of replacement rules to process strings. In our worked examples, the input is given by two binary numbers and the rules are intended to produce an output that shall represent its addition. We succeeded when we put the problem at one step of being solved but in contrary case we witnessed only the development by evolution of an adder of two numbers of the form 1+1.

10. From the syntactic point of view, the genetic code is a grammar because it consists of replacement rules of codons by amino acids. So, the correct frame of the question of the evolution of the genetic code is the evolution of grammars, which is precisely what we do in our approach to GP. Our community has the sacred duty of piling up mountains of examples to infer robust generalizations in order to support else reject accepted evolutionary explanations of the origin of the genetic code. Precisely, a computer simulation allowed its Author to claim: “Starting with very simple codes, and hence greatly reduced amino acid alphabets, we observe a diversification of the codes in most simulation runs. The driving force behind this effect is the possibility to produce fitter proteins when the repertoire of amino acids is enlarged.” (Weberndorfer, l.c., [53] 2002). Let us notice that the results of this simulation are in perfect agreement with the theory of complexity: achieved solutions to complex problems are in general numerous, varied, quite different one from another and far from being optimal. Nothing like that is seen on Earth. This is clearly a falsification of the Evolutionary Theory and of its extensions to the origin of life.

11. A key point in our grammar approach to GP is the combinatorial bases that one uses to trigger evolution. A good combinatorial basis functions as a powerful hint that is given to evolution that plays the role of an artificial intelligence. Our original aim was to prove that the evolution of hints or grammars is just one more natural problem for evolution. We failed but still consider that this is true if only cosmological time is furnished.

12. Human intelligence seems to be more than enough to tame evolution to have it designing software. Of course, a price must be paid: it is not at all simple to do that but once done, it is no more than the exercise of techniques that are routine today, such as a hierarchical attack to complex problems. The importance of this technique was already stressed by Wilson Steward ([58] 1987) and its use is now a normal routine. On the other hand, we recognize that we cannot guarantee that our methods are better than chance, especially for large inputs. In fact, the design of software with the use of evolution is such a challenging business that it is despised or mostly unknown in many professional circles and in any case it is reserved for developers with great expertise, powerful computers, plenty of time and that can appreciate the tremendous importance of this new world.

13. Evolution directed to the design of software can be simulated over any programming language and we have used Java grammars. Nevertheless LISP has some powerful advantages over Java, say, one can consider in LISP a string of symbols as an object to be manipulated by a program or, with a very simple command, view the same string as a LISP program. It is, therefore, very intelligent from our part to see in LISP a potential allied (Koza, [32] 1980 and 1996).

14. All human beings use evolution by instinct. In fact, they mutate and recombine ideas toward the better all the time. In particular, GP is the natural way of life of any human developer. Precisely, the correction of bugs and the updating of software awakes an evolutionary process. So, it is widely believed that used language play an important role in the quality and quantity of produced software. That is why all programming languages evolve. Nevertheless, involved restrictions are great and so, no language can be good for everything. That is why we need a community with a great degree of diversification. The language that best fits this directive for Java Users is .NET.

Chapter 10

Answers

Problems of Chapter 2

13, page 15. To guess a goal with n letters by mere randomness, one must guess each letter among an alphabet with 26 entries. So, the probability is $1/26$. Because all letters are guessed independently, the total probability is $p = (1/26)^n$, henceforth, the expected number of trials to make a guessing is $1/p = 26^n$. The program to verify this is the following:

```
//Program randomSearch2, B13
//A string of length k is declared as goal
//and it is guessed by random searching.
//The number of trials done to
//achieve the goal is reported.

package ejvol2p;


import java.util.Random;

public class RandomSearch2
{
    //All string have the same length
    static int stringLength=0;
    //A particular string
    static String individual = "";
    //Boolean condition to continue else halt
    static boolean problem=true;
```

```
//The string to b e guesses
static String goal = "";


// Turn on of the random generator
static Random r  = new Random();

//  This method generates random chars
//  only lower case letters of the alphabet are used
private  static  char  randomChar()
{
 return  ( char  ) ( r.nextInt( 26 ) + 65  );
}

//  A string of stringLength characters is generated
private static String generateString(int stringLength)
{
    individual = "";
    //An individual is assembled char by char
    for(int j = 1; j<= stringLength; j++)
    {
        // We generate a random char
        char c =  randomChar();
        //System.out.println( "char = " + c);
        // char c is converted into a string s
        String s = ""+c;
        // s is concatenated to Individual[i]
        individual  = individual +s;
    }
    return individual;
}


//  The goal is defined
private static void   defineProblem(int n)
{
    goal = generateString(n);
    stringLength = n;
    System.out.println( "goal = " + goal);
}
```

```
    //  The goal is achieved by guessing at random
    private static void  randomSolution(int n)
    {
        long trial = 0;
        String a;
        while (problem)
        {
            trial = trial + 1;
            a=generateString(n);
            /*
                System.out.println("Try number = " +
                    trial   + " individual " + individual);
            */
            if (a.equals(goal))
            {
             //A  tabulator sign is included
             System.out.print( "\t" + trial );
             problem =false;
            }
        }
    }

    public static void main(String[] args)
    {
        for(int k = 1; k<=15; k++)
        {
            System.out.print( k);
            defineProblem(k);
            while (problem)
              randomSolution(k);
            problem = true;
            System.out.println();
        }
    }
}////End of main class RandomSearch2,  B13
```

The output of this program produces a clearly exponential graphic, whose nature can be corroborated by taking logarithm to the number of trials and drawing the result against the length of the string: points are well represented by a line.

17, page 24. It is immediate to decide which is the better: while randomness with difficulty can pass the problem of guessing string 7 chars long, evolution has no problem in going as far as 40.

23, page 49. Intelligent design is good to solve a problem when patterns are useful to solve them. Nevertheless, there are important problems whose solution is not tamable by intelligence. They are most like checking infinitely many cases one by one. Intelligent people hate these problems. Evolution too. Now, there are problems midway: with too many families of subproblems that must be checked one but one but with the plus that inside each family evolution can provide satisfactory solutions: it is here that evolution comes to our rescue.

25, page 50. Program *randomSearch*(B11) includes: random generation of chars, insertions, comparing, copying. Program *checkOut* (B21) is entirely based on copying, recognizing of specific sites and comparing. Thus we see that these two programs, in spite of its appearance, classify as text editing so, they are also simulations of evolution. Nevertheless, from the organismic stand point they do not look as simulations of evolution because they lack reproduction of the fittest which is considered to be the distinctive feature of evolution in that field.

Problems of Chapter 4

50, page 92. Each link in its own color:

```
//Program WormsColor, B50
//Twenty worms at the start line
//OOP
//Each link in its own color
package ejvol2p;



//Needed tools
import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;
import java.util.Random;


public class WormsColor extends Applet
```

```
{

    public WormsColor()
    {}

    static  WormsColor wormsColor = new WormsColor();

    static Random r  = new Random();

    private static final long serialVersionUID = 1L;

    private  class Move
    {
        int x;
        int y;

        Move(int a, int b)
        {
            x= a;
            y = b;
        }
    }


    private   class Exon
    {
        int link ;
        Move move;

        Exon(int a, Move m)
        {
            link= a;
            move = m;
        }

        Exon(int a, int b, int c)
        {
            link= a;
            move.x = b;
```

```
        move.y = c;
    }
}


private   class LinkAddress
{
    int x;
    int y;

    LinkAddress(int a, int b)
    {
        x= a;
        y = b;
    }

}

private class Worm
{

    Exon chromosome[ ] ;
    int color[];
    LinkAddress  linkAddress[ ];

    private Worm()
    {
        chromosome = new Exon[20];
        color = new int[3];
        linkAddress = new LinkAddress[20];
    }
}



static Move vectMoves[];
static Worm worm[];
static Worm wormTwo[];
static int colors[][];
```

```
    static int nWorms;

    static int initialRow;
    static int nLinks, nExons ;

    private  void constructors()
    {
        System.out.println("constructors began");
        vectMoves = new Move[10];
        worm = new Worm[21];
        colors = new int [21][4];
    }



static private void initialPosition(Graphics g, Worm worm[]  )
{
    int k, a,b;

    for(int  W = 1; W<= nWorms; W++)
    {
        Worm  worm1;
              worm1 = wormsColor.new Worm();



        for(int  l = 1; l <= nLinks; l++)
        {
            k = initialRow - nLinks;
            a = 20 + 20 * W;
            b = k + 12*l;
            LinkAddress   ad = wormsColor.new LinkAddress (a,b);
            worm1.linkAddress [l] = ad;

            // Each link is colored
            int c1,c2,c3;
            c1 = r.nextInt(226);
            c2 = r.nextInt(226);
            c3 = r.nextInt(226);
```

```
            //colors are memorized.
            colors[W][1]= c1;
            colors[W][2]= c2;
            colors[W][3]= c3;

            g.setColor( new Color(c1,c2,c3));
            g.fillRect(a,b,4,12);
        }
        worm[W] = worm1 ;
    }
}


@Override
public  void  paint(Graphics g)
{
 nWorms = 20;
 nLinks = 12;
 initialRow = 100;
 g.drawString("Please, resize the Applet", 20,20);
 constructors();

 initialPosition (g,worm);
}
}//End of Program WormsColor, B50
```

54, page 105. In the method ribosome, change the instruction

```
g.setColor( new Color(
     worm.color[m][1],worm.color[m][2],worm.color[m][3] ));
```

by the following:

```
g.setColor( new Color(
     worm.color[n][1],worm.color[n][2],worm.color[n][3] ));
```

The reason of the bug is that the instructions are executed in the order that they appear in the chromosome, but this is done automatically, so, its index m is not necessary. What is important is what link must be moved, and which is its color. The index of the link to be moved is n.

64. With the parameters defining as in the setParameters method reported with the whole program, the output was:

```
WORST CASE MOTIVATION
Initial length = 2
Link number + waiting time
3  3
4  2
5  98
6  350
7  3480
8  12919
9  151461
10  333387
```

In a sheet of Calc, a graphic of the base 10 log of waiting time against the link index clearly renders a linear regression, with a slope approximately equal to 0.8 and a determination coefficient greater than 0.9. Hence, we have an exponential dependence, of the form $w = A10^{0.8l}$, where l is the link index and w is the waiting time.

Problems of Chapter 5

72, page 175. Recombination causes a small insignificant perturbation in the velocity of evolution to solve the posed problem. In both cases, with and without recombination, the arrival time to a solution is an exponential function in the number of links.

75, page 178. Duplication of DNA always comes with errors. The cell has various mechanisms to alleviate the problem, say, proof-reading enzymes and a sophisticated machinery for repairing. These enzymes may have allosteric modulation of their activity. This amounts to a regulation of the mutation rate.

Problems of Chapter 6

77, page 187. We need a polynomial with a root very far away from the range visited by the searching machine of the program. This is easily done: the program can visit the region in within -10^{10} and 10^{10}, so we take a root beyond 10^{10}, say 10^{15} . And to cause trapping at a local minim, we design an 'almost root' near zero. So, our polynomial looks like this: $x^2(x-10^{15})+1$. Hence, the algorithm must get trapped near zero. To see this, one can replace the method `functionValue()` of program polynomial, B76, by this new version:

```
private static double functionValue(double b)
{
    double c = 1E15; // ten to the 15th power
    return (b*b)*(b-c)+1;
}
```

To verify that everything works perfect, one can put `double c = 1E6;`. The root is in this case in within the span of the searching machine and so it can be found within an approximation of 10 significant figures.

80, page 197. Recombination is included in the following program:

```
// Program varMutation2, B80
// Evolution of a population
// of strings that finds a root
// of a fifth degree polynomial.
// Mutation rate is variable in order to
// escape local minima.
//Recombination has been added.
//It is defined char by char.
package ejvol2v2p;

import java.util.Random;

public class Polynomial3 {

    // Strings are at the same time
    // genotype and phenotype.
    // They are like ribozymes,
    // but more advanced.
    // Pre-declaration of global variables.
    // They are used all throughout the whole class.
```

```
   // Individuals are kept in the array
   // Individual[]. It is an array of strings.
   // The number of individuals must be
   // less than limit
   private static final int POPSIZE = 100;
   static double Fitness[];
   static String Individual[];//Predeclaration
   static String PatternParasite, PatternFood;
   // Order  is an array of integers.
   static int Order[];//pre-declaration
   private static int Champ;
   static String b;
   static int generation;
   private  static int numberOfClones ;
   static int nGen;
   static int ReportMin[], ReportMax[];
   static double mutationRate;
   static double recombinationRate;
   static double oldError, newError;
   static double deltaError;
   private static final boolean  PRINT = false;

   //  Turn on of the random generator
   static Random r = new Random();

   private static char randomChar() //  This method generates random chars
   // that represent digits
   {
       return (char) (r.nextInt(10) + 48);
   }

   private static void Initialization() {
       /* We generate POPSIZE individuals (strings)
15 characters long.
Sequences are completely random */

       //Formal declaration of our array.
       Individual = new String[POPSIZE];
       //  System.out.println("ORIGINAL POPULATION");
       for (int i = 0; i < POPSIZE; i++) {
```

```
            //Initialization in blank.
            Individual[i] = "";

            //An individual is assembled char by char
            for (int j = 0; j < 9; j++) {
                // We generate a random numeric char
                char c = randomChar();
                //System.out.println( "char = " + c);
                // Char c is converted into a string s
                String s = "" + c;
                // s is concatenated to Individual[i]
                Individual[i] = Individual[i] + s;
            }
            //We add the place for a point.
            char n = randomChar();
            String s = "" + n;
            Individual[i] = s + Individual[i];

            // We add a sign: zero or one.
            n = (char) (r.nextInt(2) + 48);
            s = "" + n;
            Individual[i] = s + Individual[i];
            //System.out.print("Individual " +  i + "\t");
            // System.out.println( Individual[i]);
        }

        // Declarations and default initializations
        // of other arrays.
        Order = new int[POPSIZE];
        ReportMin = new int[POPSIZE];
        ReportMax = new int[POPSIZE];
        Fitness = new double[POPSIZE];
        for (int i = 0; i < POPSIZE; i++) {
            Order[i] = 0;
            ReportMin[i] = 0;
            ReportMax[i] = 0;
            Fitness = new double[POPSIZE];
        }
    }
```

```
private static double decoder(String s) {
    // This method transforms our strings in numbers
    // We get the sign:
    char s1 = s.charAt(0);
    int sign = Character.getNumericValue(s1);
    // We get the place of the point,
    // which implies multiples of ten.
    s1 = s.charAt(1);
    int point = Character.getNumericValue(s1);
    // We build the number with the other chars
    double number = 0;
    for (int j = 2; j < 11; j++) {
        char s3 = s.charAt(j);
        int l = Character.getNumericValue(s3);
        int k = j - 1;
        double number1 = (int) l * Math.pow(10, point - k);
        number = number + number1;
    }
    if (sign == 0) {
        return (-1) * number;
    } else {
        return number;
    }
}

//The method encodes the equation = 0 to solve.
//Perfect solution must ne a root and output zero.
//Other  numbers represents an error
private static double functionValue(double b)
{
    //Root = 0.75;
    //return 4*b+3;
    //Root = root = -4.71817809;
    return b*b*b*b*b + 4*b*b*b*b - 3*b*b*b + 2*b*b +b +1;
    //Root = 1;
    //return b*b - 2*b + 1;
}


//The fitness of each individual is calculated
```

```
private static void fitness()
{
   for(int i = 0; i< POPSIZE; i++)
    {
        // Individual[i] is a string,
        // it is a genotype.
        // b is a number: it is the phenotype
        double b = decoder(Individual[i]);
        double  fV = functionValue(b);
        //Fitness(root) = 1
        //Others have Fitness less than 1
        Fitness[i] =    1/(1 + fV * fV);
    }
}


//Strings are sorted by fitness
private static void Sorting()
{

    fitness();

    for(int i = 0; i < POPSIZE;i++)
    {
        int champ = 0;
        for(int j = 0; j< POPSIZE; j++)
            if (Fitness[j] >= Fitness[champ])   champ = j;
        //The array Order classifies individuals by fitness.
        //The fittest, with less errors, go to the 0-th entry.
        Order[i] = champ;
        Fitness[champ] = 0;
    }

    Champ = Order[0];

    String s = Individual[Champ];
    if (generation % 100 == 0)
    {
        double b = decoder(s);
        double p = functionValue(b);
```

```
        System.out.println("Gen = "
            + generation + " best solution  = "
            + b + " Error = " + p );
    }

}
```

```
//The top   fill the world
private  static void Cloning()
{
    //Array to host cloned individuals
String Individual2[ ] = new String[POPSIZE];
int   i=0;
int   k=0;         //rank, 0 is the highest
//Cloning
while (i < POPSIZE)
{
    int counter = numberOfClones;
    while ((counter > 0) & (i < POPSIZE))
    {
        Individual2[i] = Individual[Order[k]];
        i++;
        counter= counter-1;
    }
    k++;
}


//Cloned individuals become the official population
for(int j = 0; j < POPSIZE; j++)
    Individual[j] = Individual2[j];
if (PRINT) report(generation );
}
```

```
    private static void Reproduction()
    {
        //Individuals are sorted by fitness
        Sorting();
        //for(int i = 0; i<  10;i++)
        //System.out.PRINTln(  Individual(Order(i))

        //Fitest individuals are cloned
        Cloning();
    }


    private static void report(int gen) {
        for(int j = 0; j< POPSIZE; j++)
        {
            System.out.print("gen " + gen + " Ind " + j );
            System.out.println(" " +  Individual[j]);
        }
        String c = Individual[POPSIZE]; //champ reported
        double d = decoder(c);
        double error = functionValue(d);
        System.out.println("genotype = " + c);
        System.out.println("Phenotype" + b);
        System.out.println(" Error = " + error);
    }


    private static void Mutation() {
        //We consider only one type of mutation:
        //substitution of one char by another one.


        char c, n;
        String s;
        String sign;
        String place;
        //All the individuals mutate except the champ = 0;
        for (int j = 1; j < POPSIZE; j++) {
            //Mutation of digits
            // w = mutated individual.
```

```
String w = "";

for (int k = 2; k <= 10; k++) {
    //Change digit k?
    double p = r.nextDouble();
    if (p < mutationRate) {
        c = randomChar();
    } else { //no mutation
        c = Individual[j].charAt(k);
    }
    // Char c is converted into a string s
    s = "" + c;
    // s is concatenated to w
    w = w + s;
}

// Change place of point
double p = r.nextDouble();
if (p < (3 / 3) * mutationRate) //mutation
{
    // place redefined
    n = randomChar();
} //no mutation
else {
    n = Individual[j].charAt(1);
}
place = "" + n;
w = place + w;
// Change sign?
p = r.nextDouble();
if (p < (1 / 3) * mutationRate) //mutation
{
    // Sign redefined
    n = (char) (r.nextInt(2) + 48);
} else { //no mutation
    n = Individual[j].charAt(0);
}
sign = "" + n;
w = sign + w;
```

```
            Individual[j] = w;
            /*System.out.println(j + "\t "
    +v+"\t "+ Individual[j]); */
        }
    }

    private static void mutationRedef(int gen) {
        String c = Individual[0];
        double d = decoder(c);
        double error = Math.abs(d);
       System.out.println("gen " + gen +
  " Error " + error);
        /* System.out.println("mutation rate = " +
    mutationRate);*/
        newError = error;
        deltaError = Math.abs(newError - oldError);
        /*
        System.out.println("Error= " + error +
" deltaError= " + deltaError);
        */
        oldError = newError;
        if (deltaError < 0.001) {
            mutationRate = 1.1 * mutationRate;
            if (mutationRate > 1) {
                mutationRate = 1;
            }

        System.out.println("gen " + gen
                    + " mutation was redefined");
        } else;
    }



private static String recombinant(int i)
{
    String recombinant = "";
    //A partner for i is found
    int j = r.nextInt(POPSIZE);
```

```
    int length = Individual[i].length();
    for(int place = 0; place < length; place++)
    {
        double p = r.nextDouble();
        if (p < recombinationRate)
            recombinant = recombinant + Individual[i].charAt(place);
        else recombinant = recombinant + Individual[j].charAt(place);
    }


   /*
    System.out.println( "Inputs = " + Individual[i]
            + "    " + Individual[j]
    + " Recombinant = " + recombinant);
   */
    return recombinant;
}


//Two strings are taken as input
//and one recombinant is produced
private static void Recombination()
{


    //Array with recombinant individuals
    String Individual2[ ] = new String[100];
    for(int i = 0; i< POPSIZE; i++)
    {
        Individual2[i] = recombinant(i);
    }
    //Recombinants become the official population
    for(int i = 0; i< POPSIZE; i++)
    {
        Individual[i] = Individual2[i];
    }

}
```

```
    private static void dynamics(int gen) {
        generation = gen;
        //The top ten are allowed to  reproduce:
        //a child occupies the place
        //of the shortest individual
        Reproduction( );
        //every 100 generations, stagnation is revised
        if (gen % 100 == 0) {
            mutationRedef(gen);

        }
        //The new population is subjected to mutation
        if (gen == nGen) {
            report(gen);
        }
        Mutation();
        Recombination();

    }



    public static void main(String[] args) {
        //To modify POPSIZE,
        // the method Copying()
        // must also be modified.
        System.out.println("Running ");
        Initialization();
        nGen = 200000;
        oldError = 1000;
        mutationRate = 0.1;
        recombinationRate = 0.5;
        numberOfClones  = 10;
        for (int gen = 0; gen <  nGen; gen++) {
            dynamics(gen);
        }
    }
}//End of Program varMutation, B80
```

81, page 197. The evolutionary rate could be taken as the derivative of the fitness function with respect to time (generations).

84, page 224. Nothing special was detected. A run cast the next results:

```
Link number + waiting time
3  3
4  10
5  378
6  947
7  10464
8  6564
9  131762
10  1690122
```

85, page 224. No special effect was detected.

Problems of Chapter 7

94, page 241.
Our program shows that an intelligent human being doing a good effort can program evolution as bad as desired: our program is too slow to deserve any further attention.

96, page 241. Recombination is included in the next program, in which a new individual or chromosome is formed by taking a first part of one and the last part of another. Our implementation of recombination rendered no improvement in the synthesis of a binary adder.

```
//Program AdditionRec, B96
//The task of devising a binary adder
//is given to evolution.
//Mutation is implemented with copyPaste.

package ejvol2v2p;

import java.util.Random;
```

```
public class BinaryAdder2 {

    public BinaryAdder2() {

    }

    //AdditionRec is a class, a prototype.
    //x is an instance of AdditionRec
    //that will be used in the sequel.
    static BinaryAdder2 x = new BinaryAdder2();

    //****Declaration of new types as inner classes**
    //Any class in Java may become a new type,
    //much as integer and string.
    //We need to define codons and chromosomes.
    //A codon is an instruction to
    //a) choose one  string among some options,
    //b) choose a place in that string,
    //c) copy a number of chars since that place on,
    //d) insert the copied substring into the output.
    class Codon {

        int fromWhat; //from which string
        int fromWhere; //from which place  on
        int howManyChars;
        int toPlace; //in which place of the output

        //A codon can be defined given some integers
        public Codon(int j, int k, int l, int m) {
            fromWhat = j;
            fromWhere = k;
            howManyChars = l;
            toPlace = m;
        }

        //A codon can be copied from another codon g
        public Codon(Codon g) {
            fromWhat = g.fromWhat;
            fromWhere = g.fromWhere;
            howManyChars = g.howManyChars;
```

```
        toPlace = g.toPlace;
    }

}

//A chromosome is an array of codons.
//Chromosome is synonymous of individual
class Chromosome {

    int chromosomeLength;
    Codon c[];

    //Everything in Java must be initialized
    //otherwise, a nillpointer error will be generated.
    //Default initialization to zero
    public Chromosome(int n) {
        Codon cd = new Codon(0, 0, 0, 0);
        c = new Codon[n];
        for (int i = 0; i < n; i++) {
            c[i] = cd;
        }
    }

    //Once initialized, a chromosome can be
    //copied as a clone from another one.
    public Chromosome(Chromosome g) {
        chromosomeLength = g.chromosomeLength;

        for (int i = 0; i < chromosomeLength; i++) {
            c[i] = g.c[i];
        }
    }
}

//******Declaration of variables*********
//******Declaration of variables*********
//  Strings that enconde numbers to be added
private static String input1 = "1111";
private static String input2 = "1111";
private static String output = "0";
```

```
//  strings are converted to numbers
private static int nInput1, nInput2, nOutput;
//  Error = difference among expected
//  addition and output
private static int error = 0;
//  Number of ciphers or bits in the inputs
private static int lengthInput;
//  Number of ciphers in all of our strings
private static int stringLength = 10;
private static int maxNumbCodons = 20;

//  We will have a population with
//  a number of POPSIZE individuals
private static final int POPSIZE = 1000;
private static Chromosome Pop1[];
private static Chromosome Pop2[];

// Individuals are sorted by fitness in Order[].
private static int Order[];
private static int gen;
private static boolean problem = true;
private static double mutationRate = 0.3;//per codon

//  Data contains the two input strings
//  plus the output.
private static final int LISTLENGTH = 3;
private static String Data[] = new String[LISTLENGTH];

// Turn on of the random generator
private static Random r = new Random();

private static final long serialVersionUID = 1L;

//********Declaration of methods*****************
//  A string of stringLength characters is generated
//  The alphabet is {0,1}.
private static String generateStringNumber(int stringLength) {
    String individual = "";
    //An individual is assembled char by char
    for (int j = 1; j <= stringLength; j++) {
```

```
        // We generate a random binary number as Character c
        int binary = r.nextInt(2);
        //System.out.println( "binary number = "+ binary );
        char c = (char) (binary + 48);
        //Character c is converted into  string s
        String s = Character.toString(c);
        //System.out.println( "binary string = "+ s );
        // s is concatenated to Pop1[i]
        individual = individual + s;
    }
    //System.out.println( individual);
    return individual;
}

private static void printCodon(Codon c) {
    System.out.println("c.fromWhat = " + c.fromWhat);
    System.out.println("c.fromWhere = " + c.fromWhere);
    System.out.println("c.howManyChars = " + c.howManyChars);
    System.out.println("c.toPlace = " + c.toPlace);
    System.out.println();
}

private static void printChromosome(Chromosome ch) {
    int n = ch.chromosomeLength;
    System.out.println();
    for (int i = 0; i < n; i++) {
        printCodon(ch.c[i]);
    }
}

private static Codon generateCodon() {
    //Default initialization
    Codon c = x.new Codon(0, 0, 0, 0);
    //True initialization
    c.fromWhat = r.nextInt(LISTLENGTH);
    int n = stringLength;
    c.fromWhere = r.nextInt(n);
    int m = c.fromWhere;
    c.howManyChars = r.nextInt(n - m);
    c.toPlace = r.nextInt(n);
```

```
        //printCodon(c);
        return c;
    }

    //  An initial population   is set  up.
    private static void initialization(int k) {
        //Formal declaration of our arrays.
        Pop1 = new Chromosome[POPSIZE];
        Pop2 = new Chromosome[POPSIZE];

        for (int i = 0; i < POPSIZE; i++) {
            //n= Number of codons in   chromosome i
            int n = r.nextInt(maxNumbCodons) + 1;

            Chromosome chr = x.new Chromosome(n);
            Pop1[i] = chr;
            Pop1[i].chromosomeLength = n;
            //System.out.println(Pop1[i].chromosomeLength);
            Pop2[i] = chr;
            Pop2[i].chromosomeLength = n;
            //System.out.println("*");
            for (int j = 0; j < n; j++) {
                Pop1[i].c[j] = generateCodon();
                Pop2[i].c[j] = generateCodon();
            }
            //printCodon(Pop1[i].c[n-1]);
        }

        // Declarations and default initialization
        Order = new int[POPSIZE];
        for (int i = 0; i < POPSIZE; i++) {
            Order[i] = 0;
        }
    }

    // This method takes substring a
    // and posits it at place Start inside c
    private static String Insert(String a, int Start, String c) {
        //System.out.println("original c " + c);
        //System.out.println("original a " + a);
```

```
    //System.out.println("Start " + Start);
    String s1, s2;
    // We copy  the left part of c
    s1 = c.substring(0, Start);
    // We copy  the right part of c
    s2 = c.substring(Start);
    // We concatenate the left part of c  to
    // the right part of c
    String z = s1 + a + s2;
    // System.out.println("Inserted c " + z);
    return z;
}

// Reads the decimal number in  string a
private static int decoder(String a) {
    int i = 0;
    int k = 0;
    int n = a.length();
    for (int j = 0; j < n; j++) {
        char c = a.charAt(j);
        if (c == '1') {
            k = 1;
        } else {
            k = 0;
        }
        i = k + i * 2;
    }
    return i;
}

//  Posits zeros at the left of the number
//  until all sites are occupied
private static String complete(String s) {
    int l = stringLength - s.length();
    for (int i = 1; i < l; i++) {
        s = '0' + s;
    }
    return s;
}
```

```
    //  The problem is defined
    private static void defineProblem(int k) {
        input1 = generateStringNumber(k);
        input1 = complete(input1);
        input2 = generateStringNumber(k);
        input2 = complete(input2);
        output = "0";
        output = complete(output);
        Data[0] = input1;
        Data[1] = input2;
        Data[2] = output;
        nInput1 = decoder(input1);
        nInput2 = decoder(input2);
        /*System.out.println("input1= " + input1  );
System.out.println("input2= " + input2 );
System.out.println("output= " + output );*/
    }


    //  Each individual is tested for how well it adds
    private static int fitness(Chromosome chr) {
        //Checking out of the algorithm with
        //some random inputs.
        error = 0;
        for (int i = 0; i < 20 * lengthInput; i++) {
            defineProblem(lengthInput);
            for (int h = 0; h < chr.chromosomeLength; h++) {
                //System.out.println(h);
                int j = chr.c[h].fromWhat;
                int k = chr.c[h].fromWhere;
                int l = chr.c[h].howManyChars;
                int m = chr.c[h].toPlace;
                String fromWhat = Data[j];
                //printCodon(chr.c[h]);
                String a = "";
                a = fromWhat.substring(k, k + l);
                output = Insert(a, m, output);
                int v = output.length();
                //System.out.println("output.length()= "+v);
                //The last lengthString chars of output are copied
```

```
            output = output.substring(v + 1 - stringLength);
            //System.out.println("Output = " + output);
            //The List of sources is updated:
            Data[2] = output;
        }
        //System.out.println();
        nOutput = decoder(output);
        //The error is the difference between the correct
        //addition and that proposed by the individual.
        error = error + Math.abs(nInput1 + nInput2 - nOutput);
    }
    //System.out.println("Gen = " + gen + "Error = " + error);
    return error;
}

// Individuals are sorted by fitness.
//  The fittest is the first, with minimum error
private static void sorting() {
    int Champ;
    int Fitness[];
    //Fitnessc[] is a copy of Fitness[]
    //used as workbench
    int Fitnessc[];
    Fitness = new int[POPSIZE];
    Fitnessc = new int[POPSIZE];
    for (int i = 0; i < POPSIZE; i++) {
        Fitness[i] = 1000000;
        Fitnessc[i] = 1000000;
    }
    for (int i = 0; i < POPSIZE; i++) {
        Fitness[i] = fitness(Pop1[i]);
        Fitnessc[i] = Fitness[i];
    }

    //We sort individuals by fitness
    // Fitness 0 means no errors
    for (int i = 0; i < POPSIZE; i++) {
        Champ = 0;
        for (int j = 0; j < POPSIZE; j++) {
            if (Fitnessc[j] < Fitnessc[Champ]) {
```

```
                    Champ = j;
                }
            }

            //The array Order keeps a record of fitness
            //the fittest is number zero
            Order[i] = Champ;
            Fitnessc[Champ] = 1000000;
        }
        //System.out.print("Min Error = " + Fitness[Order[0]]);
        //Print job to test sorting procedure
        /*
        for (int i = 0; i < POPSIZE; i++) {
            System.out.println("Fitness  = " + Fitness[Order[i]]);
        }

        System.out.println();
        */
        if (Fitness[Order[0]] == 0) //if error = 0
        {
            problem = false;
            System.out.println();
            System.out.println("Task ended by generation =" + gen);
            System.out.println("Number of trials = " + gen * POPSIZE);
            printChromosome(Pop1[Order[0]]);
            fitness(Pop1[Order[0]]);
        }
    }

    // The new generation is made
    private static void newGeneration() {
        int i = 0;
        //The fittest individuals are reproduced
        // numbCopies times
        int numbCopies = 100;
        int k = 0;  //rank, 0 is the highest

        while (i < POPSIZE) {
            int counter = numbCopies;
            while ((counter > 0) & (i < POPSIZE)) {
```

```
            //System.out.println(i + "" + l);
            Pop2[i] = Pop1[Order[k]];
            //System.out.println(i + " " + counter + " " + k);
            i++;
            counter = counter - 1;
        }
        k++;
    }

    for (int j = 0; j < POPSIZE; j++) {
        Pop1[j] = Pop2[j];
        //System.out.println( j+ " length at newGen " + Pop1[j].length);
    }
}

// Recombination of chro i is enabled
private static Chromosome rec(int i) {
    //A zeroth chromosome is created
    Chromosome g = x.new Chromosome(maxNumbCodons);
    //An initial part of   chro i is copied to g
    int l = Pop1[i].chromosomeLength;
    int m = r.nextInt(l);
    for (int k = 0; k < m; k++) {
        g.c[k] = Pop2[i].c[k];
    }

    //A partner is chosen at random
    int j = r.nextInt(POPSIZE);
    //A final part of the chro is taken
    l = Pop2[j].chromosomeLength;
    if (m < l) {
        for (int k = m; k < l; k++) {
            g.c[k] = Pop2[j].c[k];
        }
    }

    g.chromosomeLength = Math.max(m, l);
    return g;
}
```

```
// The definitive population  is a recombinant one.
private static void recombination() {
    //Pop1 is a recombinant version of Pop1
    for (int i = 0; i < POPSIZE; i++) {
        Pop2[i] = rec(i);
    }

    //Pop2 is transferred to Pop1
    for (int i = 0; i < POPSIZE; i++) {
        Pop1[i] = Pop2[i];
    }

}

// Individuals are mutated with prob mutationRate
//  Mutation= a whole codon is redefined
//  Intrigue: Are our mutations too large to be useful?
private static void mutation() {
    for (int i = 0; i < POPSIZE; i++) {
        double p = r.nextDouble();
        if (mutationRate > p) {
            int l = r.nextInt(Pop1[i].chromosomeLength);
            Pop1[i].c[l] = generateCodon();
        }
    }
}

// Given a goal, it is guessed by evolution
private static void evolSolution(int n) {
    //Individuals are sorted by fitness
    sorting();
    //The new generation is produced
    // as copies of the fittest
    newGeneration();
    recombination();
    mutation();
    gen++;
    /*System.out.println("Length of input = "
     + lengthInput+"GEN= " + gen);*/
}
```

```
    public static void main(String[] args) {
        /*int l = decoder("100111");
System.out.println(l);*/

        //r.setSeed(Int) causes the same sequence of
        //random numbers to be  always be generated.
        //r.setSeed(1);
        int seed = 54321;
        r.setSeed(seed);
        System.out.println("Seed = " + seed);

        for (int k = 1; k <= 10; k++) {
            // k is the number of places or bits of inputs
            lengthInput = k;
            System.out.print("Length of input = " + lengthInput);
            gen = 1;
            initialization(lengthInput);
            while (problem) {
                evolSolution(lengthInput);
            }
            problem = true;
            System.out.println();
        }
    }
}//End of Program AdditionRec, B96
```

99, page 242.

In the method sorting, one uses the instruction

```
if (Fitness[Order[0]] == 0)
```

to aim at a perfect computation. But one can pass to an imperfect one using the next instruction

```
if (Fitness[Order[0]] < 2)
```

or the next one

```
if (Fitness[Order[0]] < 3*lengthInput*0.05)
```

These new settings run as inefficiently as the previous ones.
103, page 244.

```
//Program SerialBinaryAdder, B103
//Binary adder that implements
//the algorithm that one learned at school.
package ejvol2v2p;

public class SerialBinaryAdder {
    //  Strings that encode numbers to be added

    static String input1, input2;

    //  strings are converted to numbers
    static int nInput1, nInput2;
    static int nOutput;
    //  Error = difference among expected
    //  addition and output
    static int error;
    //  Number of ciphers or bits in the inputs

    static int stringLength;
    static int lengthInput;

    private static final long serialVersionUID = 1L;

    // Reads the decimal number in  string a
    private static int decoder(String a) {
        int i = 0;
        int k = 0;
        int n = a.length();
        for (int j = 0; j < n; j++) {
            char c = a.charAt(j);
            if (c == '1') {
                k = 1;
            } else {
                k = 0;
            }
            i = k + i * 2;
        }
```

```
    return i;
}

private static String solution() {
    int m = input1.length();
    int n = input2.length();
    int max = Math.max(m, n);
    //Puts zeros at the left of the shorter number.
    for (int i = m; i < max; i++) {
        input1 = '0' + input1;
    }
    for (int i = n; i < max; i++) {
        input2 = '0' + input2;
    }
    String output = "";
    char carrier = '0';
    //adds bit by bit from right to left
    for (int i = 0; i < max; i++) {
        char c = input1.charAt(max - i - 1);
        char d = input2.charAt(max - i - 1);

        //this is the table for addition when
        //the carrier is included
        if ((carrier == '1') & (c == '1') & (d == '1')) {
            carrier = '1';
            output = '1' + output;
        } else if ((carrier == '1') & (c == '1') & (d == '0')) {
            carrier = '1';
            output = '0' + output;
        } else if ((carrier == '1') & (c == '0') & (d == '1')) {
            carrier = '1';
            output = '0' + output;
        } else if ((carrier == '1') & (c == '0') & (d == '0')) {
            carrier = '0';
            output = '1' + output;
        } else if ((carrier == '0') & (c == '1') & (d == '1')) {
            carrier = '1';
            output = '0' + output;
        } else if ((carrier == '0') & (c == '1') & (d == '0')) {
            carrier = '0';
```

```
                output = '1' + output;
            } else if ((carrier == '0') & (c == '0') & (d == '1')) {
                carrier = '0';
                output = '1' + output;
            } else if ((carrier == '0') & (c == '0') & (d == '0')) {
                carrier = '0';
                output = '0' + output;
            }
        }
        return output;
    }

    public static void main(String[] args) {
        input1 = "10101011";
        input2 = "1101";
        String output = solution();
        System.out.println("input1 =  " + input1);
        System.out.println("input2 =  " + input2);
        System.out.println("output = " + output);
        //Strings are transformed to int
        nInput1 = decoder(input1);
        nInput2 = decoder(input2);
        nOutput = decoder(output);
        System.out.println("input1 =  " + nInput1);
        System.out.println("input2 =  " + nInput2);
        System.out.println("output = " + nOutput);
        //The error is the difference between the correct
        //addition and that proposed by the algorithm.
        error = Math.abs(nInput1 + nInput2 - nOutput);
        System.out.println("Error = " + error);
    }
}//Program SerialBinaryAdder, B103
```

106, page 253. The relation $error3/error1$ is approximately constant and equal to 0.1. This shows that more contractions produce better results. Nevertheless, involved errors are so huge that improvement seems to be irrelevant.

108, page 260. The more contractions are included, the better is the performance of the algorithm: the number of bits of the inputs over which the algorithm

seems to function perfectly is enhanced with the number of contractions. Thus, we have every reason to think that our parallel binary adder presents indeed an approximating scheme.

117, page 287. The algorithm is very fast at short inputs but slows down for moderate ones and one begins to observe that the temp of evolution of our populations seems to be described very simply: long periods of ecstasy are interrupted by fast changes.

118, page 287. Some runs of the code below seem to suffer no change in regard with velocity but others get definitively poisoned. The expected solution, that was devised by an intelligent being, is not found by evolution. Instead, surprising solutions are proposed. As a rule, this always happen in simulations: evolution is more creative than a human can imagine and so it is not excluded that more efficient algorithms than those proposed by humans may appear. The code follows:

```
//Program AdderWHintEvol, B118
//A hint given to evolution in program
//BinAdderHintEvol, B116, is cut off.
//In that program, the method finalContraction
//was executed outside the evolutionary
//part of the program. Now, evolution must find
//the right order of incorporating it.

package ejvol2v2p;


import java.util.Random;

public class AdderWHintEvol {
    //Strings that encode numbers to be added

    static String input1 = "1111";
    static String input2 = "1111";
    static String output = "0011";
    //Strings are converted to numbers
    static int nInput1, nInput2, nOutput;
    //Error = difference among expected addition
    //and output
```

```
static int error = 0;
//Number of ciphers or bits in the inputs
static int lengthInput;
//Max lengthInput
static int stringLength = 30;
//Chromosomes have a varying number of codons
static int numbCodons = 8;

static int numberOfPairsOfContractions = 15;
static int n;

//Chro1 and Chro2 are populations of individuals
static String Chro1[];
static String Chro2[];
//limit is the number of individuals
static int limit = 30;

//Individuals are sorted by fitness in Order[].
static int Order[];
static int gen;
static boolean problem = true;

//VARIABLE PARAMETERS
static double pointMutationRate = 0.9;//per individual
static double dupliRate = 0.3;//per individual
static double deleRate = 0.3;//per individual

//Turn on of the random generator
static Random r = new Random();

private static final long serialVersionUID = 1L;

//********Declaration of methods*****************
//A string of stringLength characters is generated
//The alphabet is {A,B,C,D}. D is newbrand.
private static String generateString(int n) {
    String individual = "";
    //An individual is assembled char by char
    for (int j = 0; j < n; j++) {
        // We generate a random  number less than 4
```

```
            int numb = r.nextInt(4);
            //System.out.println( "Number = "+ numb);
            //numb defines a letter
            char c = (char) (numb + 65);
            //Character c is converted into  string s
            String s = "" + c;
            // s is concatenated to Chro1[i]
            individual = individual + s;
        }
        //System.out.println( individual);
        return individual;
    }

    //An initial population   is set up.
    private static void initialization() {
        //Formal declaration of our arrays.
        Chro1 = new String[limit];
        Chro2 = new String[limit];

        for (int i = 0; i < limit; i++) {
            //n= Number of codons in   chromosome i
            int l = r.nextInt(numbCodons) + 1;
            for (int j = 0; j < l; j++) {
                Chro1[i] = generateString(l);
                Chro2[i] = generateString(l);

            }
        }

        // Declarations and default initialization
        Order = new int[limit];
        for (int i = 0; i < limit; i++) {
            Order[i] = 0;
        }
    }

    // A string of stringLength characters is generated
    // The alphabet is {0,1}.
    private static String generateStringNumber(int lengthInput) {
        String individual = "";
```

```
        //An individual is assembled char by char
        for (int j = 1; j <= lengthInput; j++) {
            // We generate a random binary number as Character c
            int binary = r.nextInt(2);
            //System.out.println( "binary number = "+ binary );
            char c = (char) (binary + 48);
            //Character c is converted into  string s
            String s = "" + c;
            //System.out.println( "binary string = "+ s );
            // s is concatenated to individual
            individual = individual + s;
        }
        //System.out.println( individual);
        return individual;
    }

    //Reads the decimal number in  string a
    private static int decoder(String a) {
        int i = 0;
        int k ;
        int l = a.length();
        for (int j = 0; j < l; j++) {
            char c = a.charAt(j);
            if (c == '1') {
                k = 1;
            } else {
                k = 0;
            }
            i = k + i * 2;
        }
        return i;
    }

    //Posits zeros at the left of the number
    //until all sites are occupied
    private static String complete(String s) {
        int j = stringLength - s.length();
        for (int i = 1; i <= j; i++) {
            s = '0' + s;
        }
```

```
    return s;
}

//The problem is defined
private static void defineProblem(int lengthInput) {
    input1 = generateStringNumber(lengthInput);
    input1 = complete(input1);
    input2 = generateStringNumber(lengthInput);
    input2 = complete(input2);
    nInput1 = decoder(input1);
    nInput2 = decoder(input2);
    // System.out.println("input1= " + input1  );
    //System.out.println("input2= " + input2 );
}

//Intercalates input 1 with input2
//char by char and posits a separator "Y"
private static String intercalation() {
    int l = input1.length();
    int m = input2.length();
    stringLength = Math.max(l, m);
    input1 = complete(input1);
    input2 = complete(input2);
    String result = "";
    for (int i = 0; i < stringLength - 1; i++) {
        result = result + input1.charAt(i) + input2.charAt(i) + "Y";
    }
    result = result + input1.charAt(stringLength - 1)
            + input2.charAt(stringLength - 1);
    return result;
}

private static String verticalSum(String result) {
    result = result.replaceAll("10", "01");
    result = result.replaceAll("11", "10");
    return result;
}

private static String contraction1(String result) {
    result = result.replaceAll("0Y0", "M00");
```

```
        result = result.replaceAll("1Y0", "M01");
        result = result.replaceAll("0Y1", "M01");
        result = result.replaceAll("1Y1", "M10");
        n = result.length();
        //A "M0" is inserted
        result = result.substring(0, n - 1)
                + "M0" + result.substring(n - 1);

        return result;
    }

    private static String contraction2(String result) {
        result = result.replaceAll("0M0", "Y00");
        result = result.replaceAll("1M0", "Y01");
        result = result.replaceAll("0M1", "Y01");
        result = result.replaceAll("1M1", "Y10");
        //A "Y0" is inserted
        n = result.length();
        result = result.substring(0, n - 1)
                + "Y0" + result.substring(n - 1);
        return result;
    }

    private static String finalContraction(String result) {
        result = result.replaceAll("01", "1");
        result = result.replaceAll("00", "0");
        //In case of uncooked solutions
        result = result.replaceAll("10", "1");
        result = result.replaceAll("Y", "");
        result = result.replaceAll("M", "");
        return result;
    }

    //This is our genetic code.
    //We use four codons
    private static String geneticCode(char c, String result) {
        switch (c) {
            case 'A':
                result = verticalSum(result);
                break;
```

```
            case 'B':
                result = contraction1(result);
                break;
            case 'C':
                result = contraction2(result);
                break;
            case 'D':
                result = finalContraction(result);
                break;
        }
        return result;
    }

    //This is our ribosome
    private static String interpreter(String chro, String result) {
        int l = chro.length();
        for (int i = 0; i < l; i++) {
            char c = chro.charAt(i);
            result = geneticCode(c, result);
        }
        return result;
    }

    //Just for reference:
    //This is the old solution2;
    private static String solution2() {
        String result = intercalation();
        result = verticalSum(result);
        for (int l = 0; l < numberOfPairsOfContractions; l++) {
            result = contraction1(result);
            result = contraction2(result);
        }
        //finalConntraction is deleted from the program.
        //It is incorporated as D in the alphabet, the genetic code
        //and in rnadomChar()
        //result = finalContraction(result);
        return result;
    }

    //Just for reference:
```

```
//This is solution2 in evolutionary language
private static String genSolution2() {
    String result = intercalation();
    String chromosome1 = "ABCBCBCBCBCBCBCBCBCBCBCBCBCBCBCD";
    result = interpreter(chromosome1, result);
    return result;
}

//Each individual is tested for how well it adds
static public int fitness(String chr) {
    //Checking out of the algorithm with
    //some random inputs.
    error = 0;
    for (int i = 0; i < 15 * lengthInput; i++) {
        defineProblem(lengthInput);
        String result = intercalation();
        result = interpreter(chr, result);
        //System.out.println();
        nOutput = decoder(result);
        //The error is the difference between the correct
        //addition and the result proposed by chr.
        error = error + Math.abs(nInput1 + nInput2 - nOutput);
    }
    //System.out.println("chr = " + chr);
    //System.out.println("Gen = " + gen + "Error = " + error);
    return error;
}

//Individuals are sorted by fitness.
//the fittest is the first, with minimum error
private static boolean sorting() {
    int Champ;
    int Fitness[];
    //Fitnessc[] is a copy of Fitness[]
    //used as workbench
    int Fitnessc[];
    Fitness = new int[limit];
    Fitnessc = new int[limit];
    for (int i = 0; i < limit; i++) {
        Fitness[i] = 1000000;
```

```
    Fitnessc[i] = 1000000;
}
for (int i = 0; i < limit; i++) {
    Fitness[i] = fitness(Chro1[i]);
    Fitnessc[i] = Fitness[i];
}
//We sort individuals by fitness
// Fitness 0 means no errors
for (int i = 0; i < limit; i++) {
    Champ = 0;
    for (int j = 0; j < limit; j++) {
        if (Fitnessc[j] < Fitnessc[Champ]) {
            Champ = j;
        }
    }
    //The array Order classifies individuals by  fitness.
    //The fittest is number zero
    Order[i] = Champ;
    Fitnessc[Champ] = 1000000;
}
/*System.out.print("Champ= " + Chro1[Order[0]]);
System.out.print("   Min Error = " + Fitness[Order[0]]);
System.out.println();  */

       //Printing job to test sorting procedure, limit = 5
       /*for(int i = 0; i< limit;i++)
System.out.println("Fitness  = " + Fitness[ i] );
for(int i = 0; i< limit;i++)
System.out.println(" Rank   " + i +" = "+ Order[i]);

for(int i = 0; i< limit;i++)
System.out.println("Fitness  = " + Fitness[Order[i]]);

System.out.println();
 */
if (Fitness[Order[0]] == 0) //if error = 0
{
    problem = false;
    System.out.println("Task ended by generation =" + gen);
    System.out.println("Number of trials = " + gen * limit);
```

```
            System.out.println("Solution= " + Chro1[Order[0]]);
            System.out.println();
        }
        return problem;
    }

    //The new generation is made
    private static void newGeneration() {
        int i = 0;
        //The fittest individuals are reproduced
        // numbCopies times
        int numbCopies = 10;
        int k = 0;  //rank, 0 is the highest

        while (i < limit) {
            int counter = numbCopies;
            while ((counter > 0) & (i < limit)) {
                //System.out.println(i + "" + l);
                Chro2[i] = Chro1[Order[k]];
                //System.out.println(i + " " + counter + " " + k);
                i++;
                counter = counter - 1;
            }
            k++;
        }

        for (int j = 0; j < limit; j++) {
            Chro1[j] = Chro2[j];
            //System.out.println( j+ " length at newGen " + Pop1[j].le
        }
    }

    //Recombination is enabled
    private static String rec(int l) {
        String s = Chro2[l];
        String s1 = "";
        int m = s.length();
        if (m > 0) {
            int m1 = r.nextInt(m);
            s1 = s.substring(0, m1);
```

```
        }

        int k = r.nextInt(limit);
        String t = Chro2[k];
        int nn = t.length();
        String t2 = "";
        if (n > 0) {
            int m2 = r.nextInt(nn);
            t2 = t.substring(m2);
        }

        //Recombination
        return s1 + t2;
    }

    //The definitive population  is a recombinant one.
    private static void recombination() {
        //Chro2 is a copy of Chro1
        for (int i = 0; i < limit; i++) {
            Chro2[i] = Chro1[i];
        }

        //Chro1 is a recombinant version of Chro2
        for (int l = 0; l < limit; l++) {
            Chro2[l] = rec(l);
        }
        //Chro2 is transferred to Chro1
        for (int i = 0; i < limit; i++) {
            Chro1[i] = Chro2[i];
        }
    }

//This method generates random chars
    private static char randomChar() {
        // We generate a random   number less than 5
        int numb = r.nextInt(4); //D must be included
        //System.out.println( "Number = "+ numb );
        char c = (char) (numb + 65);
        return c;
    }
```

```
//String a is mutated at a random place.
//Method taken from weasel by Max Scott
    private static String pointMutation(String a) {
        //System.out.println("Old a" + a);
        int l = r.nextInt(a.length());
        //System.out.println(l);
        String z = ((l <= 0) ? "" : a.substring(0, l))
                + randomChar()
                + ((l >= (a.length() - 1)) ? ""
                        : a.substring(l + 1));
        // System.out.println("Point Mutated a" + z);
        return z;
    }

    private static String duplication(String a) {
        //System.out.println("old a = " + a);
        if (a.length() == 1) {
            a = a + a;
        } else {
            int l = r.nextInt(a.length());
            int m = r.nextInt(a.length());
            int min = Math.min(l, m);
            int max = Math.max(l, m);
            String sub = a.substring(min, max);
            //System.out.println(l + " " + m);
            a = a.substring(0, max)
                    + sub + a.substring(max);
        }
        //System.out.println("Duplication in a = " + a);
        return a;
    }

    private static String deletion(String a) {
        //System.out.println("old a = " + a);
        int l = r.nextInt(a.length());
        int m = r.nextInt(a.length());
        int min = Math.min(l, m);
        int max = Math.max(l, m);
        //Deletes from min to max
```

```
    String z = a.substring(0, min) + a.substring(max);
    return z;
}

private static void mutation() {
    for (int i = 0; i < limit; i++) {
        double p = r.nextDouble();
        if (Chro1[i].equals("")) {
            Chro1[i] = generateString(numbCodons);
        }
        if (p < pointMutationRate) {
            Chro1[i] = pointMutation(Chro1[i]);
        }
        double q = r.nextDouble();
        if (q < dupliRate) {
            Chro1[i] = duplication(Chro1[i]);
        }
        double s = r.nextDouble();
        if (s < deleRate) {
            Chro1[i] = deletion(Chro1[i]);
        }
    }
}

//Given a goal, it is guessed by evolution
private static void evolSolution(int n) {
    problem = true;
    //Individuals are sorted by fitness
    problem = sorting();

    while (problem) {
        //The new generation is produced
        //as copies of the fittest
        newGeneration();
        recombination();
        //The new population is subjected to mutation
        mutation();
        gen++;
        problem = sorting();
    }
```

```
        /*System.out.println("Length of input = "
 + lengthInput+"GEN= " + gen);*/
    }

    public static void main(String[] args) {
        /*int l = decoder("100111");
 System.out.println(l);*/

        //r.setSeed(Int) causes the same sequence of
        //random numbers to be  always be generated.
        //r.setSeed(1);
        //int seed = 54321;
        //r.setSeed(seed);
        //System.out.println("Seed = " + seed);
        initialization();
        for (int k = 1; k <= 30; k++) {
            // k is the number of places or bits of inputs
            lengthInput = k;
            System.out.println("Length of input = " + lengthInput);
            gen = 1;
            evolSolution(lengthInput);
        }
    }
}//End of class B118
```

124, page 309. The code below includes the proof that the solution is in the span of the used combinatorial basis. The program was run during 100000 generations with a population of 200 but it did not arrived to the discovery of addition of binary numbers with two figures. This failure programs our mind to believe that the we face to the usual properties of software: it is discontinuous and chaotic and therefore you get the thing else you get nothing.

```
//Program FineAdderEvol, B124
//The design of a binary adder is given as a task
//to evolution.
//A low level hint is used.
//Fine mutation is enabled:
//Instead of changing a codon as a whole,
//one elementary part of it is changed.
```

```
//Additionally, a known solution is encoded
//in the formalism given by the class.
//In that way we show that a solution is
//also a possibility allowed to the program.
package ejvol2v2p;

import java.util.Random;

public class FineAdderEvol {
    //The next constructor is necessary when
    //we have new types or inner classes.

    public FineAdderEvol() //constructor
    {
    }

    static FineAdderEvol x = new FineAdderEvol();

    //****Declaration of new types as inner classes**
    //New type Codon
    class Codon {

        private int first; //0=replacement, 1=insertion
        private int second; //place or address
        private int third;  // address in a vector of strings

        //A codon can be defined given some integers
        Codon(int j, int k, int l) {
            first = j;
            second = k;
            third = l;
        }

        //A codon can be copied from another codon g
        Codon(Codon g) {
            first = g.first;
            second = g.second;
            third = g.third;
        }
```

```
}

//New type Chromosome
//A chromosome is an array of codons.
//Chromosome is synonymous of individual.
class Chromosome {

    private int length;
    private Codon c[];

    //Everything in Java must be initialized
    //otherwise, an nillpointer error will be generated.
    //Default initialization to zero
    public Chromosome(int n) {
        length = n;
        Codon cd = new Codon(0, 0, 0);
        c = new Codon[n];
        for (int i = 0; i < n; i++) {
            c[i] = cd;
        }
    }

    //Once initialized, a chromosome can be
    //copied as a clone from another one.
    public Chromosome(Chromosome g) {
        length = g.length;

        for (int i = 0; i < length; i++) {
            c[i] = g.c[i];
        }
    }
}


//******Declaration of variables*********

//True:  verify a known solution.
//False: put evolution to work
private static final boolean RUNTEST = false;
```

```
//for exact fitting, pose ToleratedError = 0;
//You might need 10 years to accomplish your work.
private static int ToleratedError = 0;
private static int numberOfTrials;

//  Boolean variable to decide whether or not
//  a known solution must be tried out.



//  strings are converted to numbers
static int nInput1, nInput2, nOutput;
//  Error = difference among expected
//  addition and output
static double error = 0;
//  Number of ciphers or bits in the inputs
static int lengthInput;
//  Number of ciphers in all of our strings
static int stringLength = 20;
static int maxNumbCodons = 120;

//  We will have a population with
//  a number of POPSIZE individuals
private static int POPSIZE = 200;
static Chromosome Pop1[];
static Chromosome Pop2[];
static double Fitness[];

// Individuals are sorted by fitness in Order[].
static int Order[];
static private int gen;
static boolean problem = true;

// CONTROL PARAMETERS
static double pointMutationRate = 0.5;//per individual
static double dupliRate = 0.2;//per individual
static double deleRate = 0.2;//per individual

//  List contains all   words from {0,1,M,Y}
//  with 3 or less chars
```

```
private static final int NLIST = 85;
private static String List[] = new String[NLIST];
private static boolean print = false;

// Turn on of the random generator
static Random r = new Random();

private static final long serialVersionUID = 1L;


//:::::::::::: TEST KNOWN SOLUTION ::::::::::::::


// A known solution is encoded,
//from Program BinAdderHintEvol, B107:
//  "ABCBCBCBCBCBCBCBCBCBCBCBCBCBCBCD";
private static Chromosome encodeSolution() {


    Chromosome mySolution = x.new Chromosome(95);

    System.out.println("TEST OF  THE SOLUTION OF "
        + "\nProgram BinAdderHintEvol, B107"
        + "\nTranscription to the new genetic code");
    //A = verticalSum
    //0: replaceAll("10", "01")
    mySolution.c[0] = x.new Codon(0, 9, 6);
    //1: replaceAll("11", "10")
    mySolution.c[1] = x.new Codon(0, 10, 9);

    //B = contraction1
    //2: replaceAll("0Y0", "M00")
    mySolution.c[2] = x.new Codon(0, 33, 53);
    //replaceAll("1Y0", "M01")
    mySolution.c[3] = x.new Codon(0, 49, 54);
    //replaceAll("0Y1", "M01")
    mySolution.c[4] = x.new Codon(0, 34, 54);
    //replaceAll("1Y1", "M10")
    mySolution.c[5] = x.new Codon(0, 50, 57);
    //Insert M0 at n-2
```

```
mySolution.c[6] = x.new Codon(1, 2, 13);
//replaceAll("Y", "");
mySolution.c[7] = x.new Codon(0, 4, 0);

//C = contraction2
//replaceAll("0M0", "Y00")
mySolution.c[8] = x.new Codon(0, 29, 69);
//replaceAll("1M0", "Y01")
mySolution.c[9] = x.new Codon(0, 45, 70);
//replaceAll("0M1", "Y01")
mySolution.c[10] = x.new Codon(0, 30, 70);
//replaceAll("1M1", "Y10")
mySolution.c[11] = x.new Codon(0, 46, 73);
//Insert Y0 at n - 1
mySolution.c[12] = x.new Codon(1, 1, 17);

int counter = 12;
//Gene amplification associated to contration()
for (int j = 0; j < 7; j++) {

    for (int i = 0; i < 5; i++) {
        counter++;
        mySolution.c[counter]
                = x.new Codon(mySolution.c[2 + i]);
    }
    //System.out.println("Counter1 = " + counter);

    //Insert M0 at n-1
    mySolution.c[counter] = x.new Codon(1, 1, 13);

    counter++;

    //replaceAll("Y", "");
    mySolution.c[counter] = x.new Codon(0, 4, 0);



    for (int i = 0; i < 5; i++) {//C
        counter++;
        mySolution.c[counter]
```

```
                    = x.new Codon(mySolution.c[8 + i]);

        //System.out.println("counter2 = " + counter);
        }
    }

    //finalContraction
    //replaceAll("01", "1")
    mySolution.c[90] = x.new Codon(0, 6, 2);
    //replaceAll("00", "0")
    mySolution.c[91] = x.new Codon(0, 5, 1);
    //replaceAll("10", "1")
    mySolution.c[92] = x.new Codon(0, 9, 2);
    //replaceAll("Y", "")
    mySolution.c[93] = x.new Codon(0, 4, 0);
    //replaceAll("M", "")
    mySolution.c[94] = x.new Codon(0, 3, 0);

    //mySolution.length = 88;
    return mySolution;
}


    public static void testKnownSolution() {


    Chromosome mySolution = encodeSolution();
    printChromosome(mySolution);

    numberOfTrials = 3;
    for (int k = 1; k <= numberOfTrials; k++) {
        // k is the number of places or bits of inputs
        lengthInput = k;
        System.out.println("\nTEST BEGINS "
                + "\nLength of input = " + lengthInput);

        print = true;
        fitness(mySolution);

        System.out.println();
```

```
    }
}


//:::::::::::: SEARCH FOR EVOLUTIONAY SOLUTION ::::::::

// All words with less than 3 chars are generated.
//Alphabet = {0,1,M,Y}
private static void generateList() {
    String alphabet[] = {"0", "1", "M", "Y"};
    List[0] = "";

    //strings with one char
    int counter = 1;
    for (int i = 1; i < 5; i++) {
        List[counter] = alphabet[i - 1];
        counter = counter + 1;
    }

    //strings with two chars
    for (int i = 1; i < 5; i++) {
        for (int j = 1; j < 5; j++) {
            List[counter] = List[i] + alphabet[j - 1];
            counter = counter + 1;
        }
    }

//    strings with three chars
    for (int i = 5; i < 21; i++) {
        for (int j = 1; j < 5; j++) {
            List[counter] = List[i] + alphabet[j - 1];
            counter = counter + 1;
        }
    }

    //Printing checkout
    System.out.println(
            "The vector with all source strings");
    for (int i = 0; i < 85; i++) {
        System.out.println(i + " " + List[i]);
```

```
        }
        System.out.println();
    }

    //  A string of stringLength characters is generated
    //  The alphabet is {0,1}.
    private static String
        generateStringNumber(int stringLength) {
        String individual = "";
        //An individual is assembled char by char
        for (int j = 1; j <= stringLength; j++) {
            // We generate a random binary number
            int binary = r.nextInt(2);
            //System.out.println( "binary number = "+ binary );
            char c = (char) (binary + 48);
            //char c is converted into  string s

            String s = "" + c;
            //System.out.println( "binary string = "+ s );
            // s is concatenated to Pop1[i]
            individual = individual + s;
        }
        //System.out.println( individual);
        return individual;
    }

    private static void printCodon(Codon cd, int i) {
        System.out.println("\n" + i + " cd.first = " + cd.first
                + "  cd.second = " + cd.second
                + "  cd.third = " + cd.third );


        if (cd.first == 0)//replacement
        {
            String a = List[cd.second];
            String b = List[cd.third];
            System.out.println("Replace "  +  a  + " by " + b);


        }
```

```
    else
    {
        int k = cd.second;
        String c = List[cd.third];
        System.out.println("Insert " + c
                + " at place n - " + k );
    }

}



private static void printChromosome(Chromosome ch) {
    int n = ch.length;
    System.out.println();
    for (int i = 0; i < n; i++) {
        printCodon(ch.c[i], i);
    }
}

private static Codon generateCodon() {
    //Default initialization
    Codon c = x.new Codon(0, 0, 0);
    //True initialization
    //0(replacement) else 1(insertion)
    c.first = r.nextInt(2);

    if (c.first == 0) {
        //Replace what
        c.second = r.nextInt(85);
        //by what
        c.third = r.nextInt(85);
    }

    //Insert
    if (c.first == 1) {
        //where: at n - k (k = 1or  2 or 3)
        c.second = r.nextInt(3);
        //what
        c.third = r.nextInt(85);
```

```
        }


        return c;
    }

    private static Chromosome generateChro(int n) {
        Chromosome chr = x.new Chromosome(maxNumbCodons);
        chr.length = n;
        for (int j = 0; j < n; j++) {
            chr.c[j] = generateCodon();
        }
        return chr;
    }


//  An initial population   is set up.
    private static void initialization() {

        //Formal declaration of our arrays.
        Pop1 = new Chromosome[POPSIZE];
        Pop2 = new Chromosome[POPSIZE];

        Chromosome chr = x.new Chromosome(maxNumbCodons);
        for (int i = 0; i < POPSIZE; i++) {
            Pop1[i] = chr;
            //n= Number of codons in   chromosome i
            int n = r.nextInt(maxNumbCodons);
            n = Math.max(2, n);
            //System.out.println("NumbCodons = " + n);
            Pop1[i] = generateChro(n);
            Pop2[i] = chr;
        }

        // Declarations and default initialization
        Order = new int[POPSIZE];
        for (int i = 0; i < POPSIZE; i++) {
            Order[i] = 0;
        }
```

```
}

// Reads the decimal number in  string a
//Reads the  number encoded by the binary string a
// Unprocessed literals are ignored
private static int decoder(String a) {
    int number = 0;
    int n = a.length();
    for (int j = 0; j < n; j++)
    {
        char c = a.charAt(j);
        switch (c)
        {
            case '0': number = number * 2; break;
            case '1': number = 1 + number * 2;    break;
        }
        //System.out.println(c + " -> " + number);
        //Literals are not caught
    }
    return number;
}

//  Posits zeros at the left of the number
//  until all sites are occupied
private static String complete(String s) {
    int l = stringLength - s.length();
    for (int i = 1; i < l; i++) {
        s = '0' + s;
    }
    return s;
}

// Intercalates input1 with input2
// char by char and posits a separator "Y"
private static String intercalation(String input1
        , String input2) {
    int n = input1.length();
    int m = input2.length();
    int max = Math.max(n, m);
    String output = "";
```

```
        for (int i = 0; i < max; i++) {
            output = output + input1.charAt(i)
                    + input2.charAt(i) + "Y";
        }
        return output;
    }



    // This is our genetic code.
    // We use four codons

    // This is our genetic code:
    private static String geneticCode(Codon cd
            , String output) {

        int n = output.length();
        if (n < 120) //avoid extremely large strings
        {
            if (cd.first == 0)//replacement
            {
                String a = List[cd.second];
                String b = List[cd.third];
                //if( (n < 60) && ( a.length() >= b.length()))
                output = output.replaceAll(a, b);
                if (print)
                    System.out.println("Replace "  +  a
                        + " by " + b + " -> " + output);
            }

            //Avoid very large strings
            if ((cd.first == 1) && (n < 80))//insertion
            {
                int k = cd.second;
                String c = List[cd.third];
                n = output.length();
                if (n > 0) {

                    if (n - k > 0) {
                        output = output.substring(0, n - k)
```

```
                                + c + output.substring(n - k);
                    }
                } else {
                    output = c;
                }
                if (print)
                    System.out.println("Insert " + c
                        + " at place n - " + k + " -> " + output);
            }
        }

        return output;
    }


    // This is our ribosome
    private static String interpreter(Chromosome chro
                    ,String intercalation) {
        //System.out.println("Intercalation = " + intercalation);
        String output = intercalation;
        int n = chro.length;
        for (int i = 0; i < n; i++)
        {
            if (print)
            {
                System.out.println("\nCodon number " + i);
                printCodon(chro.c[i], i);
            }

            output = geneticCode(chro.c[i], output);

        }

        return output;
    }


    //Each individual is tested for how well it adds.
    //Random inputs.
    private static double fitness(Chromosome chr) {
```

```
error = 0;
/*
    System.out.println("Gen = " + gen + " " +
        "chr.length = "+chr.length);
*/
for (int i = 0; i < numberOfTrials; i++) {
    if (print)
    {
        System.out.println("\nTest number " + i);
    }
    String input1 = generateStringNumber(lengthInput);
    input1 = complete(input1);
    String input2 = generateStringNumber(lengthInput);
    input2 = complete(input2);
    nInput1 = decoder(input1);
    nInput2 = decoder(input2);

    String intercalation
            = intercalation(input1, input2);
    intercalation = complete(intercalation);
    String interpretation
            = interpreter(chr, intercalation);
    int result = decoder(interpretation);


    //The error is the difference between the correct
    //addition and that proposed by  chro.
    int myError = Math.abs(nInput1 + nInput2 - result);
    if (print)
    {
        System.out.println("\nRESULT OF TEST:"
            + "\nInput1 = "
                + input1  + " -> " + nInput1
            + " \nInput2 = "
                + input2   + " -> " + nInput2
            + " \nIntercalation = " + intercalation
            + " \nInterpretation = " +  interpretation
            + " \nResult = " + result
            + " \nError = "
            + myError);
```

```
        }
        error = error + myError;

    }
if (print) System.out.println("Total error = " + error);
    return Math.abs(error);
}

// Individuals are sorted by fitness
//  The fittest is the first, with minimum error

// Individuals are sorted by fitness
//  The fittest is the first, with minimum error
private static boolean sorting() {
    //System.out.println("sorting");
    problem = true;
    int Champ;
    double bestFitness = 0;

    Fitness = new double[POPSIZE];


    for (int i = 0; i < POPSIZE; i++) {
        Fitness[i] = fitness(Pop1[i]);
        if (print)
            System.out.println("i = " + i
                + " lengthInput = " +lengthInput
                + " Gen = " + gen
                +  " Total Errorrrr = " + Fitness[i]);
    }

    //We sort individuals by fitness
    // Fitness 0 means no errors
    for (int i = 0; i < POPSIZE; i++) {
        Champ = 0;
        for (int j = 0; j < POPSIZE; j++) {
            if (Fitness[j] < Fitness[Champ]) {
                Champ = j;
            }
        }
```

```
            //The array Order classifies individuals by  fitness.
            //the fittest is number zero
            Order[i] = Champ;
            if (i == 0) bestFitness = Fitness[ Order[0]];
            Fitness[Champ] = 1000000;
        }
        System.out.println("lengthInput = " + lengthInput
                + " Gen = " + gen
                + " Best fitness = "  + bestFitness
                );
        /*
        //Print job to test sorting procedure, POPSIZE = 5

for(int i = 0; i< POPSIZE;i++)
 System.out.println("Fitness  = " + Fitness[Order[i]]);

System.out.println();
         */
        if (bestFitness <= ToleratedError)
        {
            problem = false;
            System.out.println(
                    "\nTask ended by generation = " + gen);
            System.out.println(
                    "POPSIZE = " + POPSIZE
                    +"\ngen * POPSIZE = " + gen * POPSIZE
            + "\nNumber of random trials per test = "
                            + numberOfTrials
            + "\nToleratedError = " + ToleratedError);
            System.out.println(
                    "\nCODONS OF THE WINNER CHROMOSOME");
            printChromosome(Pop1[Order[0]]);
            print = true;
            System.out.println(
                    "\nTEST OF THE WINNER WITH A NEW SET"
                            + "OF RANDOOM INPUTS");
            fitness(Pop1[Order[0]]);
            print = false;
        }
```

```
    return problem;
}
// The new generation is made
private static void newGeneration() {
    int i = 0;
    //The fittest individuals are reproduced
    // numbCopies times
    int numbCopies = 10;
    int k = 0;  //rank, 0 is the highest

    while (i < POPSIZE) {
        int counter = numbCopies;
        while ((counter > 0) & (i < POPSIZE)) {
            //System.out.println(i + "" + l);
            Pop2[i] = Pop1[Order[k]];
            //System.out.println(i + " " + counter + " " + k);
            i++;
            counter = counter - 1;
        }
        k++;
    }

    for (int j = 0; j < POPSIZE; j++) {
        Pop1[j] = Pop2[j];
        /*
System.out.println( j+ " length at newGen " +
                        Pop1[j].length);
         */
    }
}

// Recombination is enabled
private static Chromosome rec(int l) {
    //the first part of "l" is copied
    int m1 = 1;
    int m = Pop1[l].length;
    if (m > 0) {
        m1 = r.nextInt(m);
        for (int i = 0; i < m1; i++) {
            Pop2[l].c[i] = Pop1[l].c[i];
```

```
            }
        }
        //The second part of "k" is copied
        int k = r.nextInt(POPSIZE);
        int n = Pop1[k].length;
        if (n > 0) {
            int n1 = r.nextInt(n);
            for (int i = m1; i < m1 + n - n1; i++) {
                if (i < maxNumbCodons) {
                    Pop2[l].c[i] = Pop1[k].c[n1 + i - m1];
                }
            }
        }
        return Pop2[l];
    }

    // The definitive population  is a recombinant one.
    private static void recombination() {
        //Pop2 is a copy of Pop1
        for (int i = 0; i < POPSIZE; i++) {
            Pop2[i] = Pop1[i];
        }

        //Recombination
        for (int l = 0; l < POPSIZE; l++) {
            Pop2[l] = rec(l);
        }
        //Pop2 is transferred to Pop1
        for (int i = 0; i < POPSIZE; i++) {
            Pop1[i] = Pop2[i];
        }
    }

    // Chro g a is mutated at a random codon.
    private static Chromosome pointMutation(Chromosome g) {
        //System.out.println("PointMutation");
        //Which codon must be mutated?
        int l = r.nextInt(g.length);
        //Which part of the codon must be mutated?
        int m = r.nextInt(3);
```

```
        if (m == 0) {
            g.c[l].first = 0;
            g.c[l].second = r.nextInt(85);
        }

        if (m == 1) {
            g.c[l].first = 1;
            g.c[l].second = r.nextInt(1000);
        }

        if (m == 2) {
            g.c[l].third = r.nextInt(85);
        }
        return g;
    }

    private static Chromosome duplication(Chromosome g) {
        //System.out.println("duplication");
        /*
if (g.length ==0)
System.out.println("Old length at dup = "
                + g.length);
         */
        int l = r.nextInt(g.length);
        int m = r.nextInt(g.length);
        int min = Math.min(l, m);
        int max = Math.max(l, m);
        //System.out.println("Old length at dup = " + g.length);
        //System.out.println("min = " + min + " Max " + max);
        Chromosome chr = g;
        int top = Math.min(2 * max - min, maxNumbCodons);
        for (int i = max; i < top; i++) {
            g.c[i] = chr.c[i - (max - min)];
        }
        int toptop = Math.min(chr.length + max - min
                , maxNumbCodons);
        if (top < maxNumbCodons) {
            for (int i = top; i < toptop; i++) {
                g.c[i] = chr.c[i - (max - min)];
            }
```

```
        }
        g.length = toptop;
        /*
  * System.out.println("min = " + min +
   "max = " + max +  "top = " + top + "toptop = " + toptop);
         */
        //System.out.println("New length at dup = " + g.length);

        //System.out.println();
        return g;
    }

    private static Chromosome deletion(Chromosome g) {
        //System.out.println("deletion");
        //System.out.println("Old length at del = " + g.length);
        int l = r.nextInt(g.length);
        int m = r.nextInt(g.length);
        int min = Math.min(l, m);
        int max = Math.max(l, m);
        //Deletes from min to max
        for (int i = max; i < g.length; i++) {
            g.c[i - max + min] = g.c[i];
        }

        //System.out.println("Max = " + max + "Min = " + min);
        g.length = g.length - (max - min);
        //System.out.println("New length at del = " + g.length);

        if (g.length < 2) {

            int n = r.nextInt(maxNumbCodons);
            n = Math.max(2, n);
            g = generateChro(n);
        }
        return g;
    }

    // Chromosome g is mutated at a random codon.
    private static Chromosome change(Chromosome g) {
        double p = r.nextDouble();
```

```
    if (p < pointMutationRate) {
        g = pointMutation(g);
    }
    double q = r.nextDouble();
    if (q < dupliRate) {
        g = duplication(g);
    }
    double s = r.nextDouble();
    if (s < deleRate) {
        g = deletion(g);
    }
    return g;
}

//  All individuals are mutated
private static void mutation() {
    for (int i = 0; i < POPSIZE; i++) {
        //System.out.println( "gen = " + gen);
        int l = Pop1[i].length;
        //System.out.println( i + "  oldLength at mut = " + l);
        Pop1[i] = change(Pop1[i]);
        int m = Pop1[i].length;
        //System.out.println(i + "newLength at mut = " + m);
    }
}

// Given a goal, it is guessed by evolution
private static void evolSolution() {
    numberOfTrials = 10 * lengthInput;
    //Individuals are sorted by fitness
    problem = sorting();
    while (problem) {
        newGeneration();
        //The new population is subjected to mutation
        mutation();
        recombination();
        gen++;
        if (gen % 1000 == 0) {
            System.out.println("lengthInput = "
                    + lengthInput
```

```
                        + "Gen = " + gen + "Fitness  = "
                        + Fitness[Order[0]]);
            }
            problem = false;
            problem = sorting();
        }
    }



    public static void main(String[] args) {
        /*int l = decoder("100111");
 System.out.println(l);*/

        generateList();

        int POPSIZE1 = POPSIZE;
        if (RUNTEST == true) {
            testKnownSolution();
            System.out.println("Test done");
        }
        //To make  an ordinary run
        if (RUNTEST == false) {
            POPSIZE = POPSIZE1;
            int seed = 1000000;
            r.setSeed(seed);
            System.out.println("Seed = " + seed);
            initialization();
            for (int k = 1; k <= 10; k++) {
                // k is the number of places or bits of inputs
                lengthInput = k;
                numberOfTrials = 10 * lengthInput;
                System.out.println("Length of input = "
                        + lengthInput);
                gen = 1;
                problem = true;
                while (problem) {
                    evolSolution();
                }
```

```
                System.out.println();
            }
        }// end of if
    }
}//End of Program FineAdderEvol, B124
```

Problems of Chapter 8

128, page 339. Hours and hours of continuous execution produced no result.

130, page 381. Previous programs were able to sum binary numbers with one cipher but in hours and hours of computation no further progress was made. By contrasts, this new code surpasses the barrier of summing numbers with one bit almost immediately and solves the problem for two ciphers in some few minutes but further advance was very slow: it arrived to the synthesis of binary adders with three ciphers in 23 hours, required other 6 additional hours to arrive to the synthesis of binary adders with 4 ciphers and no further advance was observed in the next 10 hours. A desktop with an Intel microprocessor of 3.2 GH was used.

Chapter 11

Glossary

Algorithm An ordered set of instructions for solving a problem.

Allele A variant form of the same gene.

Alphabet A set of letters or sounds (phonemes). In nature, the fundamental alphabet is $\{A,T,C,G\}$. In genetic programming, the letters correspond to Java methods. In most applications, alphabets are useless because of combinatorial explosions, so one prefers super-alphabets, whose letters are composed of words built with letters of the alphabet.

Amino acid A building block of a protein. The genetic code deals with 20 amino acids but there are more than 150 amino acids in the biochemistry of plants. In the proteins of humans there are 25 amino acids, the 20 of the genetic code plus other five that result as modifications of the first.

Applet A type of program in Java specially suited to deal with graphics. They are the enlivening actor of the web.

Application program A computer program to fulfill a set of functions.

AWT (Abstract Window Toolkit) The initial facility of Java to make GUIs (graphic user interfaces). It has been superseded by Swing, which is completely based on OOP.

Base pair A building block of DNA: A (adenine) , T (thymine) , C (cytosine) and G (Guanine). In RNA the T is changed by U (Uracil).

Binary The quality of having only two possibilities.

Binary code A message written in an alphabet consistent of 0 and 1. The rules to decipher such a message.

Biocomputation Computation done by biological processes.

Bioinformatics The study of how information is represented and transmitted in biological systems.

Bit A quantity and measure of information that corresponds to a yes else not answer.

Boolean operators The mathematical operators AND, OR, and NOT.

Bug Misbehavior of a program.

Cell The building block of any living organism.

Chromosome The structure formed by DNA and the proteins that cover it.

Chip Encapsulated electronic circuit.

Clone A population of genetically identical beings. Quantum mechanics forbids the existence of exact clones.

Cloning The formation of clones or exact replicas.

Coding region A portion of DNA that encodes for a protein.

Code A portion of a program.

Codon A sequence of three adjacent nucleotides that encodes for an amino acid or a signal to begin or stop transcription. In general, it is a substring that contains an elementary instruction.

Combinatorial basis Is an alphabet for an evolutionary process.

Combinatorial explosion The generation of so many possibilities to test that a life would not be enough. Such an explosion is evidenced because if one augments the length of the input, the number of possibilities augments in an exponential form.

Complex The quality of an object that cannot be described with few words.

Compiler A program that translates a portion of code written in Java to a machine-language program, which can be understood by the CPU.

Computer program A set of instructions that tells a computer which operations to perform.

Computer A machine that with the appropriate instructions can run programs.

Conditional probability The probability of an event that depends on the occurrence of another event.

Convergence The quality of approaching a final state.

Creationism The belief claiming that living beings were created by a supernatural spirit, God. Some variants change God by aliens.

CPU (Central processing unit) The brain of the computer.

Datum The record of an observation.

Debugger A program that helps the developer to find bugs.

Debugging The process of cleaning bugs from a program.

Deletion The operation of cutting some substrings from a string.

Design To conceive in the mind. To have as a goal or purpose. To plan by making a preliminary sketch, outline, or drawing. To create or execute in an artistic or highly skilled manner(Webster's II Dictionary). We restrict the application of design to actors with free will.

Designer A rational entity that is committed to design.

Developer Any entity, rational or not, that is committed to the fabrication of something. Evolution is a software developer.

Digital Class of systems that are based on all or nothing outputs.

Distributed The quality of being partitioned in pieces.

DNA (deoxyribonucleic acid) The molecule that contains the genetic information in most living beings.

Domain A polypeptide (a portion of a protein) that can be assigned a specific function. The corresponding DNA code is called scheme.

eJGrid The Grid of the evolJava community.

Environment The world outside the studied system.

Epistemology The division of philosophy that studies the nature and origin of knowledge.

Eukaryote A cell whose nucleus is bounded by a membrane and that has organelles.

Evolution In epistemology, it is the scientific project whose aim is to prove that variations along living beings can be explained by a natural mechanism from just some few origins or maybe from one or maybe from no origin at all. In evolutionary biology, it is the change along lineages that add together to give rise to the origin of new species and phyla. In the genome project, evolution is the union of all sort of factors that affect genomes of individuals or of populations, be it mutations from parents to offspring or changes from a germ cell to a somatic one or transfer of genetic information from an organism to another. In sociobiology, it is the fundamental force that programs instincts.

Evolutionary environment It is an ambient of work composed of an alphabet plus a genetic code for interpretation plus a form to try out diverse super-alphabets is called an . If one has an evolutionary environment plus infinite computing power and time, evolution can be run automatically.

Evolutionism The philosophical vision that evolution is the correct and final explanations to our existence.

EvolJava community The community of all persons that currently use Java as a precious tool to study own and foreign ideas about evolution. Since our task surpasses the possibilities of one single person, our community will soon get organized. Please, visit http://www.evoljava.com or write to the Author: jose@evoljava.com

Execute To perform instructions.

Exon A region of DNA that is transcribed to an amino acid substring.

Failure An inability to perform the assigned function.

Gedanken A German word that means imagined, say, a gedanken experiment. This was the only form of attacking in ancient times the fundamentals of science. The new one is simulation. Simulation with Java all along the wide world is possibly the future of simulation. Warning: simulation alone is insufficient to account with all questions of biology, so we need mathematical models and very clever gedanken experiments.

Gene A segment of DNA that encodes a protein and all the signals that regulate its transcription.

Genetic code The correspondence among codons and aminoacids + start and stop signal for transcription.

Genome The genetic material of an organism.

genome project The systematic study of all forms of genomes.

Genotype The specific genetic information of an individual.

Google An Internet tool to search documents on the web. It does not demand fees but invades the world of the user with advertisements and moreover it sells to others his or her electronic address and usual searched motifs.

Grammar Set of replacement rules to process strings or words formed with the letters of an alphabet.

Grid The union of many desktop computers in a single structure to exploit the potentialities of parallel computation.

GUI (graphic user interface) The nice face of programs with windows, menus, buttons and the like.

Haploid An organism whose cell contain only one set of chromosomes.

Hardware The wiring of a computer.

Information theory The science of communication.

Interface A piece make to connect two other parts of a system.

Intron A part of DNA that is non-coding.

Lab evolution Artificial evolution in the laboratory with real DNA, RNA and/or proteins.

Letterzyme A string of characters taken from a given alphabet that simulate an individual.

Machine language The set of instructions in binary code that are understood and executed by a computer processor.

Markov chain A probabilistic model in which a system can occupy some states and that has constant probabilities to pass from one state into another.

Memory The capability of storing and retrieving information. Chips that are used to store programs and data.

Model An abstraction of some part of the world.

Modeling The art of making models.

Mutation A change in the genetic information.

Neural network A computing devise that imitates nervous system.

OOP (Object-oriented programming) The style of programming in which data, operators and protocols of communication are encapsulated into a single entity.

Occam's razor The advise to prefer simplicity when it explains all things.

OS (Operating system) The software that enlivens a computer. It consists in a library of programs fulfilling functions that are necessary for the user. It is a super-alphabet.

Operator A machine that transforms some input.

Output The result produced by a process or program.

Parameter Variable of a model whose value can be changed by the user.

Phenotype Any observable feature of an organism whose existence can be traced to some few genes.

Point mutation A change of a single nucleotide in a DNA sequence.

Programming language A set of reserved words and rules to specify how to command a computer.

Prokaryote An organism whose cells lacks a membrane-bounded nucleus.

Prototype A model of something to be built.

RAM (Random-access memory) A portion of electronic memory that can be used to write and retrieve information.

Recombinant The product resulting from the interchange of different portions of two initial items.

Recombination A string operation that receives as input two or more strings and produces one or more strings by interchanging their parts.

Recursion The repeated call to the same procedure.

Ribozyme RNA chain that can catalyze its own reproduction.

Scheme A DNA pattern. In general, it is word of an alphabet.

Selection The connection of the changes of the environment with those in the genome or in the frequencies of their representatives is called **selection**.

Simulation A program that represents a model.

SOFT The problem of developing software. The developer could be a conscious being, like a human, or an unconscious process, like evolution. If the conscious being has a purpose, we speak of design. The evolutionary theory reads: evolution is the developer that is responsible for the existence of the genome, the natural software. Evolution has no purpose so, it is not a designer, but acquires an emergent purpose: to gain more and more surviving capability. Thus, its developed software must comply with a function: augment differential surviving and reproduction. Therefore, all what human designers know about software design is also valid for natural evolution.

Software Computer programs.

Structured programming The style of programming in which modular approach and reuse is enabled and re-enforced.

Super-alphabet An alphabet whose letters correspond to words composed with letters of another alphabet. In genetic programming, a super-alphabet is a library of programs.

Swing A package to generate GUIs in Java. It is the very modern toolkit for making a Java Interface.

Systematics The identification of unique characteristics that are shared by certain organisms and not others. From this, it is found who is most closely related to whom and so it is retrodicted the relation by descent and the evolutionary tree of life.

Thread A specific sequencing of execution. Java can run many threads at the same time.

Transcription The operation of making mRNA, which is a RNA photocopy of some DNA information.

t-RNA (Transfer RNA) A small RNA molecule that serves to the cell to decipher a genetic sequence thanks to the genetic code: in one extreme it has a codon and in the other it has the corresponding amino acid.

Translation The process of converting mRNA to protein.

Universal computation The property enjoyed by those computers that can compute whatever is computable.

Bibliography

[1] ADLEMAN L (1994) Molecular computation of solution to combinatorial problems. *Science* 266: 1021–1024(Nov 11).
http://www.usc.edu/dept-00/dept/molecular-science/papers/adleman-sc
Revised 14/X/2016 393

[2] ADAMI C (2007) Biological programming *Nature*, 446: 263–264.
http://www.nature.com/nature/journal/v446/n7133/full/446263a.html
Revised 14/X/2016 65

[3] AYALA F AND OTHER 14 AUTHORS (2008) Science, Evolution and Creationism, National Academy of Science USA, Institute of Medicine
http://books.nap.edu/html/11876/SECbrochure.pdf.
Cited 10 Jan 2008.
Revised 14/X/2016 7

[4] BACHER J, REISS D, ELLINGTON A (2002) Anticipatory evolution and DNA shuffling *Genome Biology* 2002, 3(8):reviews1021.1[U+0096]-1021.4
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC139397/.
Revised 14/X/2016 384

[5] BALL P (1995) Towards the synthetic dragonfly. *Nature* 375(6527):101–102.
http://www.nature.com/nature/journal/v375/n6527/pdf/375101a0.pdf
Revised 14/X/2016 11

[6] BEAVER D (1996) A universal Molecular computer. DNA based computers. DIMACS Vol 27, Editors Linpton and Baum. AMS. 29–36. 393

[7] BOLLENBACH T, VETSIGIAN K AND KISHONY R (2007) Evolution and multilevel optimization of the genetic code. *Genome Res. 17: 401–404;* doi:10.1101/gr.6144007.
http://genome.cshlp.org/content/17/4/401.abstract
Revised 14/X/2016 389

[8] DARNELL J, LODISH H, BALTIMORE D (1986) *Molecular cell biology*. Scientific American Books. 8

[9] DAWKINS R(1976) *The selfish gene* Oxford: Oxford University Press.
http://download.bioon.com.cn/view/upload/201109/12125304_287
Revised 14/X/2016 11

[10] DAWKINS R (1986) The Blind Watchmaker. W.W.Norton & Company, NY, London.
http://terebess.hu/keletkultinfo/The_Blind_Watchmaker.pdf
Revised 14/X/2016 12

[11] DAWKINS R (2006) The God delusion.Houghton Mifflin Company.
https://luptaanticapitalista.files.wordpress.com/2011/04/the
Revised 14/X/2016 10

[12] DARWIN C (1859) *The origin of species* Murrap, Londres.
http://darwin-online.org.uk/content/frameset?itemID=F373&vie
Revised 14/X/2016 8, 392

[13] DEITEL H, DEITEL P (2004) Java How to program, Firth Edition. Prentice Hall Inc.
http://staff.cs.psu.ac.th/iew/cs344-481/Java%20How%20to%20Pr
Revised 14/X/2016 393

[14] DEMBSKI W (1998) *The Design Inference: Eliminating Chance through Small Probabilities*, Cambridge U. Press, New York. 10

[15] DETTMAN J, SIRJUSINGH C, KOHN L, ANDERSON J (2007) Incipient speciation by divergent adaptation and antagonistic epistasis in yeast. *Nature*, 447(7144):585-8.
http://www.nature.com/nature/journal/v447/n7144/full/nature0
Revised 14/X/2016 8

[16] THEODOSIUS DOBZHANSKY, MARCH 1973
"Nothing in biology makes sense except in the light of evolution."
The American Biology Teacher,
https://www.pbs.org/wgbh/evolution/library/10/2/text_pop/l_102_01.h
Revised 14/X/2016 11

[17] EDITOR OF NATURE (2008) Editorial: Spread the word, *Nature* 451, 108.
http://www.nature.com/nature/journal/v451/n7175/full/451108b.html
Revised 14/X/2016 11

[18] ANDRZEJ (ANJAY) ELZANOWSKI AND JIM OSTELL (1996) The Genetic Codes Compiled by National Center for Biotechnology Information (NCBI), Bethesda, MD
http://www.bioinformatics.org/jambw/2/3/TranslationTables.html.
Revised 14/X/2016 388

[19] MICROSOFT (R)(1994) Encarta, *Computer*. 392

[20] FINKEL T, SERRANO M, BLASCO M (2007) The common biology of cancer and ageing *Nature*, 448: 767-774.
http://www.nature.com/nature/journal/v448/n7155/full/nature05985.htm
Revised 14/X/2016 11

[21] FREELAND S, HURST L (1998) The Genetic Code Is One in a Million *J Mol Evol* 47:238-248.
https://www.ncbi.nlm.nih.gov/pubmed/9732450
Revised 14/X/2016 8, 389

[22] FRIEDBERG R, (1958) A learning machine. Part I. IBM *Journal of Research and Development* 2(1):2–13.
http://ieeexplore.ieee.org/document/5392654/?arnumber=5392654
Revised 14/X/2016 392

[23] FURLA P, ALLEMAND D, SHICK J, FERRIER-PAGES C, RICHIER S, PLANTIVAUX A, MERLE P AND TAMBUTTE S (2005) The Symbiotic Anthozoan: A Physiological Chimera between Alga and Animal. Integr. Comp. Biol. (2005) 45 (4): 595-604. doi: 10.1093/icb/45.4.595
http://icb.oxfordjournals.org/content/45/4/595.full
Revised 14/X/2016 385

[24] GILIS D, MASSAR S, CERF N, AND ROOMAN M (2001) Optimality of the genetic code with respect to protein stability and amino acid frequencies. arXiv:physics/0102044.
https://arxiv.org/pdf/physics/0102044.pdf
Revised 14/X/2016 389

[25] GOLDENFELD N, C WOESE (2007) Biology's next revolution *Nature*, 445: 369.
http://www.nature.com/nature/journal/v445/n7126/full/445369a
Revised 14/X/2016 9, 385

[26] HAAG-LIAUTARD C, DORRIS M, MASIDE X, MACASKILL S, HALLIGAN D, CHARLESWORRTH B, KEIGHTLEY P (2007) Direct estimation of per nucleotide and genome deleterious mutation rates in Drosophila. *Nature*, 445: 82–85.
http://www.nature.com/nature/journal/v445/n7123/full/nature0
Revised 14/X/2016 8

[27] HERMANN J, MARTI-ARBONA M, FEDOROV A, FEDOROV E, ALMO S, SHOICHET B, RAUSHEL F (2007) Structure-based activity prediction for an enzyme of unknown function *Nature*, 448: 775–779.
http://www.nature.com/nature/journal/v448/n7155/abs/nature05
Revised 14/X/2016 11

[28] HIGUCHI T., IWATA M., AND LIU W., (EDITORS), (1997) Proceedings of The First International Conference on Evolvable Systems: From Biology to Hardware (ICES96).Lecture Notes in Computer Science, Vol. 1259, Springer-Verlag, Heidelberg, 1997. 384

[29] HOLLAND J (1975) Adaptation in natural and artificial systems. University of Michigan Press.
392

[30] JOERN J, MEINHOLD P AND ARNOLD F (2002) Analysis of Shuffled Gene Libraries. *J. Mol. Biol.* (2002) 316, 643–656.
http://www.cheme.caltech.edu/groups/fha/publications/Joern_e
Revised 14/X/2016 386

[31] KAUFFMAN (1995) At home in the universe. Oxford University Press, NY. 11

[32] KOZA J, (1980, 1996) Genetic programming. On the Programming of Computers by Means of Natural Selection. A Bradfort Book. The MIT Press. First printing: 1980; Fifth Printing: 1996. 393, 395

[33] MEHL R, ANDERSON J, SANTORO S, WANG L, MARTIN A, KING D, HORN D AND SCHULTZ P (2003) Generation of a Bacterium with a 21 Amino Acid Genetic Code *J. Am. Chem. Soc., 125 (4), 935–939.*
http://pubs.acs.org/doi/abs/10.1021/ja0284153
Revised 14/X/2016 388

[34] MELOTT A (2002) Intelligent Design Is Creationism in a Cheap Tuxedo, Physics Today Online, Vol 55, June 2002, Opinion,
http://scitation.aip.org/content/aip/magazine/physicstoday/article/
Revised 14/X/2016 10

[35] NetBeans Developing Applications with NetBeans IDE. (2014)
http://docs.oracle.com/cd/E50453_01/doc.80/e50452/whatsnew.htm
Revised 14/X/2016 2

[36] NetBeans IDE Java Quick Start Tutorial (2015)
https://netbeans.org/kb/docs/java/quickstart.html
Revised 14/X/2016 2

[37] NIRENBERG M W AND MATTHAEI J H (1961). The dependence of cell-free protein synthesis in e. coli upon naturally occurring or synthetic polyribonucleotides. *Proc. Natl. Acad. Sci. USA* 47:1588–1602.
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC223178/
Revised 14/X/2016 392

[38] NOBEL PRIZE ORGANIZATION (2008) The Transistor in a Century of Electronics
http://nobelprize.org/educational_games/physics/transistor/history/index.html
Revised 14/X/2016 386

[39] PADIAN K (2005) The testimony of Kevin Padian in Kitzmiller v. Dover, Edited by Nick Matzke, National Center for Science Education.
http://tinyurl.com/2nlgar.
Revised 14/X/2016 10

[40] PATEL N H (2006) Evolutionary biology: How to build a longer beak. *Nature* 442, 515–516 (3 August 2006)

http://www.nature.com/nature/journal/v442/n7102/full/442515a.html
Revised 14/X/2016 9

[41] RADMAN M, TADDEI F, MATIC I (2000) Evolution-driving genes. *Res. Microbiol.* 151 (2000) 91-95.
https://www.ncbi.nlm.nih.gov/pubmed/10865953
Revised 14/X/2016 8

[42] RAYMOND J, BLANKENSHIP R (2003) Horizontal gene transfer in eukaryotic algal evolution *PNAS* 100, 7419-7420.
http://www.pnas.org/content/100/13/7419.short
Revised 14/X/2016 385

[43] RAY T (1990) Evolution and optimization of digital organisms. In Brown H (Editor), *Proceedings of the 1990 IBM Supercomputing Competition: Large Scale Computing Analysis and Modeling Conference.* MIT Press. See also:
http://life.ou.edu/tierra/
Revised 14/X/2016 9

[44] RODRIGUEZ J (2008) Electromagnetism and geometry, arXiv:0806.1492
http://front.math.ucdavis.edu/0806.1492
Revised 14/X/2016 11

[45] SAMUEL A, (1959) Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development* 3(3):210–219.
http://researcher.watson.ibm.com/researcher/files/us-beygel/samuel
Revised 14/X/2016 392

[46] SCHMID G (1982) The chemical basis of life. Translated to Spanish: Química Biológica, Emalsa(1986), Madrid.
Revised 14/X/2016 272

[47] SHURKIN J (2008) From spies to servers. *Nature* 452, 533 (3 April 2008).
http://www.nature.com/nature/journal/v452/n7187/full/452533a.html
Revised 14/X/2016 393

[48] THE CHIMPANZEE SEQUENCING AND ANALYSIS CONSORTIUM (25 AUTHORS) (1995) Initial sequence of the chimpanzee genome and comparison with the human genome. *Nature* 437, 69-87 (1 September 2005)
http://www.nature.com/nature/journal/v437/n7055/abs/nature04072.ht
Revised 14/X/2016 9

[49] VETSIGIAN K, WOESE C, AND GOLDENFELD N (2006) Collective evolution and the genetic code. *Proc. Nat. Acad. Sci. USA* Vol.103, No 28: 10696–10701.
http://www.pnas.org/content/103/28/10696.full
Revised 14/X/2016 389

[50] VOIGT C, C MARTINEZ C, WANG Z, MAYO S & ARNOLD F (2002) Protein building blocks preserved by recombination, *Nature Structural Biology* 9, 553 - 558 (2002) doi:10.1038/nsb805
http://www.nature.com/nsmb/journal/v9/n7/abs/nsb805.html
Revised 14/X/2016 386

[51] Wagner I, Musso H (November 1983)"New Naturally Occurring Amino Acids". Angewandte Chemie International Edition in English. 22 (11): 816–28. doi:10.1002/anie.198308161
http://onlinelibrary.wiley.com/doi/10.1002/anie.198308161/abstract
Revised 14/X/2016 387

[52] WATSON J D, CRICK F H C (1953) A structure for deoxyribose nucleic acid. *Nature* 171: 737–738.
http://www.nature.com/nature/journal/v171/n4356/abs/171737a0.html
Revised 14/X/2016 392

[53] WEBERNDORFER G, P HOFACKER, F. STADLER (2002) On the Evolution of Primitive Genetic Codes. See also the original dissertation cited there.
http://www.santafe.edu/media/workingpapers/02-08-034.pdf
Revised 14/X/2016 388, 394

[54] WEN-HSIUNG LI (1997) Molecular evolution, Sinauer Associates, Inc, Sunderland Massachusetts. 262, 394

[55] WIKIBOOKS CONTRIBUTORS, (2012) "An Introduction to Molecular Biology," The Free Textbook Project,
https://en.wikibooks.org/w/index.php?title=An_Introduction_to_Molecular_Bi
Revised 14/X/2016 65

[56] WIKIPEDIA CONTRIBUTORS. SALAMIS TABLET [INTERNET]. (2016) Wikipedia, The Free Encyclopedia; 2016 Oct 5, 00:37
https://en.wikipedia.org/w/index.php?title=Salamis_Tablet&oldid=742657928
Revised 14/X/2016 392

[57] WIKIPEDIA CONTRIBUTORS. "YUPANA," (2016) Wikipedia, The Free Encyclopedia; 2016 Oct 5, 00:37

https://en.wikipedia.org/w/index.php?title=Yupana&oldid=735427622
Revised 14/X/2016 392

[58] WILSON S (1987) Classifier Systems and the animat problem. Machine Learning 3(2): 199-228.
http://link.springer.com/article/10.1007/BF00058679
Revised 14/X/2016 395

[59] WHITFIELD J (2007) We are a family *Nature*, 446: 247–249.
http://www.nature.com/nature/journal/v446/n7133/full/446247a.html
Revised 14/X/2016 10

[60] WONG J. (1975) A Co-Evolution Theory of the Genetic Code. *Proc. Nat. Acad. Sci. USA* Vol. 72, No. 5, pp. 1909–1912, May.
http://www.pnas.org/content/72/5/1909.abstract
Revised 14/X/2016 388

Index

alphabet
    super, 322
computer, 49
hardware, 49

algorithm
    probabilistic , 207
alphabet, 49, 322
    super, 322
approximating scheme, 208, 211

cell, 51
chaos, 52
chromosome, 195
class, 2
code, 49
code of interpretation, 49
codon, 51, 195
combinatorial basis, 234
combinatorial basis, 234
combinatorial explosion, 266
completeness problem, 157
computation
    universal, 334
console, 3
copy, 4
creationism, 7
crossing over, 123

debugging, 50
delete, 4
designer, 50

developer
    evolution, 122
digital, 52
domain, 325

Eclipse, 1, 2
evolJava, 208
evolJava community, 3
Evolution, 265
evolution, 5, 97
    basic items, 39
evolutionary environment, 324, 394
evolutionism, 7

false evolutionary solutions, 149
founder effect, 328
frustrating scheme, 208

genetic code, 51
genetic programming, 322
genome, 50
genome project, 6
grammar, 322
grid, 224

interface, 10

Java, 1, 51
    installing, 1

lab evolution, 324
language, 49

maximum
    global , 149
    local , 149
mutation rate
    regulated , 150
    self regulated , 165
    variable, 149
    varying , 157

operating system, 49

pasting, 4
program, 49
programming language, 49

recombinant operator, 123
recombination, 123, 207
robot, 50

scheme, 324
schemes, 325
select all, 4
selection, 6, 396
selection of the fittest, 40
self-organization, 97
semantic content, 49
simulation, vii
simulation of evolution, 322
Software, 49
software developer, 50
surviving, 51
systematics, 6

verbal instruction, 49

weasel problem, 235