DNA computing: arrival of biological mathematics

Lila Kari[1]

"Come forth into the light of things
Let nature be your teacher."
(Wordsworth, [72])

1 Biological mathematics: the tables turned

The field usually referred to as *mathematical biology* is a highly interdisciplinary area that lies at the intersection of mathematics and biology. Classical illustrations include the development of stochastic processes and statistical methods to solve problems in genetics and epidemiology. As the name used to describe work in this field indicates, with "biology" the noun, and "mathematical" the modifying adjective, the relationship between mathematics and biology has so far been one–way. Typically, mathematical results have emerged from or have been used to solve biological problems (see [34] for a comprehensive survey). In contrast, Leonard Adleman, [1], succeeded in solving an instance of the directed Hamiltonian path problem solely by manipulating DNA strings. This marks the first instance of the connection being reversed: a mathematical problem is the end toward which the tools of biology are used. To be semantically correct, instead of categorizing the research in DNA computing as belonging to mathematical biology, we should be employing the mirror–image term *biological mathematics* for the field born in November 1994.

Despite the complexity of the technology involved, the idea behind biological mathematics is the simple observation that the following two processes, one biological and one mathematical, are analogous:

(a) the very complex structure of a living being is the result of applying simple operations (copying, splicing, etc.) to initial information encoded in a DNA sequence,

(b) the result $f(w)$ of applying a computable function to an argument w can be obtained by applying a combination of basic simple functions to w (see Section 4 or [65] for details).

If noticing this analogy were the only ingredient necessary to cook a computing DNA soup, we would have been playing computer games on our DNA

[1] Department of Computer Science, University of Western Ontario, London, Ontario, N6A 5B7 Canada, email:lila@csd.uwo.ca, http://www.csd.uwo.ca/~lila

laptops a long time ago! It took in fact the ripening of several factors and a renaissance mind like Adleman's, a mathematician knowledgeable in biology, to bring together these apparently independent phenomena. Adleman realized that not only are the two processes similar but, thanks to the advances in molecular biology technology, one can use the biological to simulate the mathematical. More precisely, DNA strings can be used to encode information while enzymes can be employed to simulate simple computations, in a way described below. (See [39] for more detailed explanations of molecular biology terms.)

DNA (deoxyribonucleic acid) is found in every cellular organism as the storage medium for genetic information. It is composed of units called nucleotides, distinguished by the chemical group, or base, attached to them. The four bases, are *adenine, guanine, cytosine* and *thymine*, abbreviated as A, G, C, and T. (The names of the bases are also commonly used to refer to the nucleotides that contain them.) Single nucleotides are linked together end–to–end to form DNA strands. A short single-stranded polynucleotide chain, usually less than 30 nucleotides long, is called an *oligonucleotide.* The DNA sequence has a *polarity*: a sequence of DNA is distinct from its reverse. The two distinct ends of a DNA sequence are known under the name of the $5'$ end and the $3'$ end, respectively. Taken as pairs, the nucleotides A and T and the nucleotides C and G are said to be *complementary.* Two complementary single–stranded DNA sequences with opposite polarity will join together to form a double helix in a process called *base-pairing* or *annealing.* The reverse process – a double helix coming apart to yield its two constituent single strands – is called *melting* (Fig.1).

A single strand of DNA can be likened to a string consisting of a combination of four different symbols, A, G, C, T. Mathematically, this means we have at our disposal a 4 letter alphabet $\Sigma = \{A, G, C, T\}$ to encode information, which is more than enough, considering that an electronic computer needs only two digits, 0 and 1, for the same purpose.

Some of the simple operations that can be performed on DNA sequences are accomplished by a number of commercially available enzymes that execute a few basic tasks. One class of enzymes, called *restriction endonucleases*, will recognize a specific short sequence of DNA, known as a *restriction site.* Any double-stranded DNA that contains the restriction site within its sequence is cut by the enzyme at that location (Fig.2).

Another enzyme, called *DNA ligase*, will bond together, or "ligate", the end of a DNA strand to another strand. The *DNA polymerases* perform several functions including replication of DNA. The replication reaction requires a guiding DNA single-strand called *template*, and a shorter oligonucleotide called *primer*, that is annealed to it. Under these conditions, DNA polymerase catalyzes DNA synthesis by successively adding nucleotides to one end of the primer. The primer is thus extended in one direction until the desired strand that starts with the primer and is complementary to the template is obtained (Fig.3).

Finally, using enzymes called *exonucleases*, either double-stranded or single-stranded DNA molecules may be selectively destroyed. The exonucleases chew up DNA molecules from the end in, and exist with specificity to either single-stranded or double-stranded form. There are other enzymes that could potentially be useful, but for our models of computation these are sufficient.

The practical possibilities of encoding information in a DNA sequence and of performing simple bio–operations were used in [1] to solve a 7 node instance of the Directed Hamiltonian Path Problem. A directed graph G with designated vertices v_{in} and v_{out} is said to have a Hamiltonian path if and only if there exists a sequence of compatible "one–way" edges $e_1, e_2, \ldots, e_z$ (that is, a path) that begins at v_{in}, ends at v_{out} and enters every other vertex exactly once.

The following (nondeterministic) algorithm solves the problem:

Step 1. Generate random paths through the graph.

Step 2. Keep only those paths that begin with v_{in} and end with v_{end}.

Step 3. If the graph has n vertices, then keep only those paths that enter exactly n vertices.

Step 4. Keep only those paths that enter all of the vertices of the graph at least once.

Step 5. If any paths remain, say "YES"; otherwise say "NO".

To implement Step 1, each vertex of the graph was encoded into a random 20–nucleotide strand (20–letter sequence) of DNA. Then, for each (oriented) edge of the graph, a DNA sequence was created consisting of the second half of the sequence encoding the source vertex and the first half of the sequence encoding the target vertex. By using complements of the vertices as splints, DNA sequences corresponding to compatible edges were ligated, that is, linked together. Hence, the ligation reaction resulted in the formation of DNA molecules encoding random paths through the graph.

To implement Step 2, the product of Step 1 was amplified by polymerase chain reaction (PCR). (See Section 2 and Fig.2.) Thus, only those molecules encoding paths that begin with v_{in} and end with v_{end} were amplified.

For implementing Step 3, a technique called gel electrophoresis was used, that makes possible the separation of DNA strands by length (Fig.4). The molecules are placed at the top of a wet gel, to which an electric field is applied, drawing them to the bottom. Larger molecules travel more slowly through the gel. After a period, the molecules spread out into distinct bands according to size.

Step 4 was accomplished by iteratively using a process called affinity purification. This process permits single strands containing a given subsequence v (encoding a vertex of the graph) to be filtered out from a heterogeneous pool of other strands (Fig.5). After synthesizing strands complementary to v and attaching them to magnetic beads, the heterogeneous solution is passed over the beads. Those strands containing v anneal to the complementary sequence and are retained. Strands not containing v pass through without being retained.

To implement Step 5, the presence of a molecule encoding a Hamiltonian path was checked. (This was done by amplifying the result of Step 4 by polymerase chain reaction and then determining the DNA sequence of the amplified molecules. See Section 2 for details).

A remarkable fact about Adleman's result is that not only does it give a solution to a mathematical problem, but that the problem solved is a hard computational problem in the sense explained below (see [24], [27]).

Problems can be ranked in difficulty according to how long the best algorithm to solve the problem will take to execute on a single computer. Algorithms whose running time is bounded by a polynomial (respectively exponential) function, in terms of the size of the input describing the problem, are in the "polynomial time" class P (respectively the "exponential time" class EXP). A problem is called *intractable* if it is so hard that no polynomial time algorithm can possibly solve it.

A special class of problems, apparently intractable, including P and included in EXP is the "non–deterministic polynomial time" class, or NP. The following inclusions between classes of problems hold:

$$P \subseteq NP \subseteq \text{ EXP} \subseteq \text{ Universal.}$$

NP contains the problems for which no polynomial time algorithm solving them is known, but that can be solved in polynomial time by using a non–deterministic computer (a computer that has the ability to pursue an unbounded number of independent computational searches in parallel). The directed Hamiltonian path problem is a special kind of problem in NP known as "NP–complete". An NP–complete problem has the property that every other problem in NP can be reduced to it in polynomial time. Thus, in a sense, NP–complete problems are the "hardest" problems in NP.

The question of whether or not the NP–complete problems are intractable, mathematically formulated as "Does P equal NP?", is now considered to be one of the foremost open problems of contemporary mathematics and computer science. Because the directed Hamiltonian path problem has been shown to be NP–complete, it seems likely that no efficient (that is, polynomial time) algorithm exists for solving it with an electronic computer.

Following [1], in [42] a potential DNA experiment was described for finding a solution to another NP–complete problem, the Satisfiability Problem. The Satisfiability Problem consists of a Boolean expression, the question being whether or not there is an assignment of truth values – true or false – to its variables, that makes the value of the whole expression true. Later on, the method from [42] was used in [43], [44] and [45], to show how other NP–complete problems can be solved. DNA algorithms have since been proposed for expansion of symbolic determinants [41], graph connectivity and knapsack problem using dynamic programming [8], road coloring problem [35], matrix multiplication [49], addition [28], exascale computer algebra problems [68], etc.

In [3], [12], "molecular programs" were given for breaking the U.S. government's Data Encryption Standard (DES). DES encrypts 64 bit messages and uses a 56–bit key. Breaking DES means that given one (plain–text, cipher–text) pair, we can find a key which maps the plain–text to the cipher–text. A conventional attack on DES would need to perform an exhaustive search through all of the 2^{56} DES keys, which, at a rate of 100,000 operations per second, would take 10,000 years. In contrast, it was estimated that DES could be broken by using molecular computation in about 4 months of laboratory work.

The problems mentioned above show that molecular computation has the potential to outperform existing computers. One of the reasons is that the operations molecular biology currently provides can be used to organize massively parallel searches. It is estimated that DNA computing could yield tremendous advantages from the point of view of *speed, energy efficiency* and *economic information storing.* For example, in Adleman's model, [2], the number of operations per second could be up to approximately 1.2×10^{18}. This is approximately 1,200,000 times faster than the fastest supercomputer. While existing supercomputers execute 10^9 operations per Joule, the energy efficiency of a DNA computer could be 2×10^{19} operations per Joule, that is, a DNA computer could be about 10^{10} times more energy efficient (see [1]). Finally, according to [1], storing information in molecules of DNA could allow for an information density of approximately 1 bit per cubic nanometer, while existing storage media store information at a density of approximately 1 bit per 10^{12} nm^3. As estimated in [6], a single DNA memory could hold more words than all the computer memories ever made.

2 Can DNA compute everything?

The potential advantages of DNA computing versus electronic computing are clear in the case of problems like the Directed Hamiltonian Path Problem, the Satisfiability Problem, and breaking DES. On the other hand, these are only particular problems solved by means of molecular biology. They are one–time experiments to derive a combinatorial solution to a particular sort of problem. This immediately leads to two fundamental questions, posed in Adleman's article and in [27] and [45]:

(1) What kind of problems can be solved by DNA computing?

(2) Is it possible, at least in principle, to design a programmable DNA computer?

More precisely, one can reformulate the problems above as:

(1) Is the DNA model of computation computationally complete in the sense that the action of any computable function (or, equivalently, the computation of any Turing machine) can be carried out by DNA manipulation?

(2) Does there exist a universal DNA system, i.e., a system that, given the encoding of a computable function as an input, can simulate the action of that function for any argument? (Here, the notion of function corresponds to the notion of a program in which an argument w is the input of the program and the value $f(w)$ is the output of the program. The existence of a universal DNA system amounts thus to the existence of a DNA computer capable of running programs.)

Opinions differ as to whether the answer to these questions has practical relevance. One can argue as in [13] that from a practical point of view it maybe not be that important to simulate a Turing machine by a DNA computing device. Indeed, one should not aim to fit the DNA model into the Procrustean bed of classical models of computation, but try to completely rethink the notion of computation. On the other hand, finding out whether the class of DNA algorithms is computationally complete has many important implications. If the answer to it were unknown, then the practical efforts for solving a particular problem might be proven futile at any time: a Gödel minded person could suddenly announce that it belongs to a class of problems that are impossible to solve by DNA manipulation. The same holds for the theoretical proof of the existence of a DNA computer. As long as it is not proved that such a thing theoretically exists, the danger that the practical efforts will be in vane is always lurking in the shadow.

One more indication of the relevance of the questions concerning computational completeness and universality of DNA–based devices is that they have been addressed for most models of DNA computation that have so far been proposed.

The existing models of DNA computation are based on various combinations of a few primitive *biological operations*:

– *Synthesizing* a desired polynomial–length strand, used in all models (Fig.6). In standard solid phase DNA synthesis, a desired DNA molecule is built up nucleotide by nucleotide on a support particle in sequential coupling steps. For example, the first nucleotide (monomer), say A, is bound to a glass support. A solution containing C is poured in, and the A reacts with the C to form a two-nucleotide (2-mer) chain AC. After washing the excess C solution away, one could have the C from the chain AC coupled with T to form a 3-mer chain (still attached to the surface) and so on.

– *Mixing*: pour the contents of two test tubes into a third one to achieve union, [1], [2], [5], [45], [59]. Mixing can be performed by rehydrating the tube contents (if not already in solution) and then combining the fluids together into a new tube, by pouring and pumping for example.

– *Annealing*: bond together two single-stranded complementary DNA sequences by cooling the solution, as illustrated in Fig.1. (See [9], [13], [59], [63], [70].) Annealing in vitro is also known as *hybridization*.

– *Melting*: break apart a double-stranded DNA into its single-stranded complementary components by heating the solution, as described in Fig.1. (See [9], [13], [59], [63], [70].) Melting in vitro is also known under the name of *denaturation*.

– *Amplifying (copying)*: make copies of DNA strands by using the Polymerase Chain Reaction (PCR) as illustrated in Fig.3. (See [1], [2], [5], [9], [10], [11], [13], [41], [45], [63].) PCR is an in vitro method that relies on DNA polymerase to quickly amplify specific DNA sequences in a solution. PCR involves a repetitive series of temperature cycles, with each cycle comprising three stages: denaturation of the guiding template DNA to separate its strands, then cooling to allow annealing to the template of the *primer* oligonucleotides, which are specifically designed to flank the region of DNA of interest, and, finally, extension of the primers by DNA polymerase. Each cycle of the reaction doubles the number of target DNA molecules, the reaction giving thus an exponential growth of their number.

– *Separating* the strands by length using gel electrophoresis as used in Step 3 of Adleman's experiment and depicted in Fig.4. (See [1], [2], [5], [9], [10], [11], [13].)

– *Extracting* those strands that contain a given pattern as a substring by using affinity purification, as used in Step 4 of Adleman's experiment and depicted in Fig.5. (See [1], [2], [9], [11], [13], [45].)
– *Cutting* DNA double-strands at specific sites by using restriction enzymes as explained in Section 1 and Fig.2. (See [9], [10], [11], [13], [30], [58], [63].)
– *Ligating*: paste DNA strands with compatible sticky ends by using DNA ligases as shown in Section 1. (See [9], [10], [11], [30], [58], [63], [70].)
– *Substituting*: substitute, insert or delete DNA sequences by using PCR site-specific oligonucleotide mutagenesis, as shown in Fig.7 (see [11], [38]). The process is a variation of PCR in which a change in the template can be induced by the process of primer modification. Namely, one can use a primer that is only partially complementary to a template fragment. (The modified primer should contain enough bases complementary to the template to make it anneal despite the mismatch.) After the primer is extended by the polymerase, the newly obtained strand will consist of the complement of the template in which a few oligonucleotides seem "substituted" by other, desired ones.
– *Marking* single strands by hybridization: complementary sequences are attached to the strands, making them double-stranded. The reverse operation is *unmarking* of the double-strands by denaturing, that is, by detaching the complementary strands. The marked sequences will be double-stranded while the unmarked ones will be single-stranded, [5], [46], [59].
– *Destroying* the marked strands by using exonucleases as explained in Section 1 (see [46]), or by cutting all the marked strands with a restriction enzyme and removing all the intact strands by gel electrophoresis, [5].
– *Detecting* and *Reading*: given the contents of a tube, say "yes" if it contains at least one DNA strand, and "no" otherwise, [1], [2], [5], [9], [13], [45]. PCR may be used to amplify the result and then a process called *sequencing* is used to actually read the solution. The basic idea of the most widely used sequencing method is to use PCR and gel electrophoresis. Assume we have a homogeneous solution, that is, a solution containing mainly copies of the strand we wish to sequence, and very few contaminants (other strands). For detection of the positions of A's in the target strand, a blocking agent is used that prevents the templates from being extended beyond A's during PCR. As a result of this modified PCR, a population of subsequences is obtained, each corresponding to a different occurrence of A in the original strand. Separating them by length using gel electrophoresis will give away the positions where A occurs in the strand. The process can then be repeated for each of C, G and T, to yield the sequence of the strand. Recent methods use four different fluorescent dyes, one for each base, which allows all four bases to be processed simultaneously. As the fluorescent molecules pass a detector near the bottom of the gel, data are output directly to an electronic computer.

The bio-operations listed above, and possibly others, will then be used to write "programs" which receive a tube containing DNA strands as input and return as output either "yes" or "no" or a set of tubes. A computation consists of a sequence of tubes containing DNA strands.

There are pro's and con's for each model (combination of operations). The ones using operations similar to Adleman's have the obvious advantage that they could already be successfully implemented in the lab. The obstacle preventing the large scale automatization of the process is that most bio–operations are error prone and rely on mainly manual handling of tubes. In contrast, the model introduced by Tom Head in [30] aims to be a "one–pot" computer with all the operations carried out in principle by enzymes. Moreover, it has the theoretical advantage of being a mathematical model with all the claims backed up by mathematical proofs. Its disadvantage is that the current state of art in biotechnology has not allowed yet practical implementation. Overall, the existence of different models with complementing features shows the versatility of DNA computing and increases the likelihood of practically constructing a DNA computing–based device.

In the sequel we will restrict our attention to the *splicing system* model of DNA recombination that has been introduced in the seminal article of Tom Head, [30], as early as 1987. A formal definition of the *splicing* operation (a combination of cut and paste), that can be used as the sole primitive for carrying out a computation, is given in Section 3. We will then prove in Section 4 that for the DNA model based on splicing we can affirmatively answer both questions posed at the beginning of this section.

3 A mathematical model: splicing systems

As described in Section 1, a DNA strand can be likened to a string over a four letter alphabet. Consequently, a natural way to model DNA computation is within the framework of formal language theory (a branch of discrete mathematics related to computer science), which deals with letters and strings of letters. Before presenting the technical background and formal definition of the *splicing system* model, we briefly describe the abstraction process that leads from the actual DNA recombination (a combination of cutting by restriction enzymes and ligation by DNA ligases) to the mathematical operation of splicing. (The reader can consult [30], [31], [54] as well as the introduction and the appendix of [32] for details.)

Consider the following two double-strands of DNA:

$$\begin{matrix} 5'CCCCCTCGACCCCC3' & 5'AAAAAGCGCAAAAA3' \\ 3'GGGGGAGCTGGGGG5' & 3'TTTTTCGCGTTTTT5' \end{matrix}$$

and two restriction enzymes (*Taq*I and *Sci*NI), whose recognition sites are

$$\begin{matrix} 5'TCGA3' & 5'GCGC3' \\ 3'AGCT5' & 3'CGCG5' \end{matrix}$$

respectively. The effect of the enzymes on the two given DNA strands is the cutting, by each enzyme, of the strand containing its restriction site as a subsequence. As a result, four new DNA strands are produced:

$$\begin{matrix} 5'CCCCCT & CGACCCCC3' & 5'AAAAAG & CGCAAAAA3' \\ 3'GGGGGAGC & TGGGGG5' & 3'TTTTTCGC & GTTTTT5' \end{matrix}$$

Note that the sticky end of the first strand is complementary to the sticky end of the last one, while the same thing happens with the second and third strand. The DNA strands with compatible ends can now be ligated by DNA ligase, the result being

$$\begin{array}{ll} 5'CCCCCTCGCAAAAA3' & 5'AAAAAGCGACCCCC3' \\ 3'GGGGGAGCGTTTTT5' & 3'TTTTTCGCTGGGGG5' \end{array}$$

The above combination of cut and paste performed by enzymes is an instance of *DNA recombination*. Its mathematical abstractization, called the *splicing operation*, is defined under the following assumptions:

(i) we consider strings of *symbols*, rather than double-stranded structures. (Due to the complementarity $A-T$ and $C-G$, no loss of information occurs.);

(ii) the size of the alphabet is not restricted to four letters. (This generalization seems reasonable, as any alphabet can be encoded into a binary one.);

(iii) The length of sites and the number of enzymes working together is not restricted. (This is a rather liberal supposition, but it is justified by the fact that the splicing system aims to be an abstract general model of DNA computation, not necessarily confined to today's biotechnology.)

With these assumptions in mind, we are almost ready for the definition of the splicing operation and splicing system as mathematical objects. Our exposition requires a few formal language notions and notations, introduced in the following. (For further formal language notions the reader is referred to [60]).

An alphabet is a finite nonempty set; its elements are called *letters* or *symbols*. Σ^* denotes the free monoid generated by the alphabet Σ under the operation of catenation (juxtaposition). The elements of Σ^* are called *words* or *strings*. The empty string (the null element of Σ^*) is denoted by λ. A *language* over the alphabet Σ is a subset of Σ^*. For instance, if $\Sigma = \{a, b\}$ then $aaba, aabbb = a^2b^3$ are words over Σ, and the following sets are languages over Σ: $L_1 = \{\lambda\}$, $L_2 = \{a, ba, aba, abbaa\}$, $L_3 = \{a^p \mid\ p \text{ prime }\}$.

Since languages are sets, we may define the set–theoretic operations of union, intersection, difference, and complement in the usual fashion. The catenation of languages L_1 and L_2, denoted L_1L_2, is defined by $L_1L_2 = \{uv \mid\ u \in L_1, v \in L_2\}$.

A finite language can always be defined by listing all of its words. Such a procedure is not possible for infinite languages and therefore other devices for the representation of infinite languages have been developed. One of them is to introduce a *generative device* and define the language as consisting of all the words generated by the device. The basic generative devices used for specifying languages are *grammars*.

A *generative grammar* is an ordered quadruple

$$G = (N, T, S, P),$$

where N and T are disjoint alphabets, $S \in N$ and P is a finite set of ordered pairs (u, v) such that u, v are words over $N \cup T$ and u contains at least one letter of N. The elements of N are called *nonterminals* and those of T *terminals*; S is called the axiom. Elements (u, v) of P are called rewriting rules and are written $u \longrightarrow v$. If $x = x_1 u x_2$, $y = x_1 v x_2$ and $u \longrightarrow v \in P$, then we write $x \Longrightarrow y$ and say that x derives y in the grammar G. The reflexive and transitive closure of the derivation relation $\Longrightarrow$ is denoted by $\Longrightarrow^*$. The language generated by G is

$$L(G) = \{w \in T^* | \ S \Longrightarrow^* w\}.$$

Intuitively, the language generated by the grammar G is the set of words over the terminal alphabet that are derived from the axiom by repeatedly applying the rewriting rules.

Grammars are classified by imposing restrictions on the forms of productions. A grammar is called of *type–0* if no restriction (zero restrictions) is applied to the rewriting rules and is called *regular* if each rule of P is of the form $A \longrightarrow aB$, $A \longrightarrow a$, $A, B \in N$, $a \in T$. The family of finite languages will be denoted by FIN, the family of languages generated by regular grammars by REG and the family of languages generated by type–0 grammars by $\mathcal{L}_0$.

Using these formal language theory prerequisites, we can proceed now to define the *splicing operation.*

As described in [30] and modified in [26], given an alphabet Σ and two strings x and y over Σ, the splicing of x and y according to the splicing rule r consists of two steps: *(i)* cut x and y at certain positions determined by the splicing rule r, and *(ii)* paste the resulting prefix of x with the suffix of y, respectively the prefix of y with the suffix of x. Using the formalism introduced in [50], a *splicing rule* r over Σ is a word of the form $\alpha_1 \# \beta_1 \$ \alpha_2 \# \beta_2$, where $\alpha_1, \beta_1, \alpha_2, \beta_2$ are strings over Σ and $\#, \$$ are markers not belonging to Σ.

We say that z and w are obtained by *splicing* x and y according to the splicing rule $r = \alpha_1 \# \beta_1 \$ \alpha_2 \# \beta_2$, and we write

$$(x, y) \longrightarrow_r (z, w)$$

if and only if

$$\begin{matrix} x = x_1 \alpha_1 \beta_1 x_1' \\ y = y_2 \alpha_2 \beta_2 y_2' \end{matrix} \quad \text{and} \quad \begin{matrix} z = x_1 \alpha_1 \beta_2 y_2' \\ w = y_2 \alpha_2 \beta_1 x_1', \end{matrix}$$

for some $x_1, x_1', y_2, y_2' \in \Sigma^*$.

The words $\alpha_1 \beta_1$ and $\alpha_2 \beta_2$ are called *sites* of the splicing, while x and y are called the *terms* of the splicing. The splicing rule r determines both the sites and the positions of the cutting: between α_1 and β_1 for the first term and between α_2 and β_2 for the second. Note that the site $\alpha_1 \beta_1$ can occur more than once in x while the site $\alpha_2 \beta_2$ can occur more than once in y. Whenever this happens, the sites are chosen nondeterministically. As a consequence, the result of splicing x and y can be a set containing more than one pair (z, w).

We illustrate the way splicing works by using it to simulate the addition of two positive numbers, n and m. If we consider the alphabet $\Sigma = \{a, b, c\}$ and the splicing rule $r = a \# b \$ c \# a$, then the splicing of $x = a^n b$ and $y = ca^m$ according to r yields the words a^{n+m} and cb. Indeed,

$$(a^n b, ca^m) = (x, y) =$$

$$(\underbrace{a^{n-1}}_{x_1}\underbrace{a}_{\alpha_1}\underbrace{b}_{\beta_1}\underbrace{\lambda}_{x_1'},\underbrace{\lambda}_{y_2}\underbrace{c}_{\alpha_2}\underbrace{a}_{\beta_2}\underbrace{a^{m-1}}_{y_2'})\longrightarrow_r(\underbrace{a^{n-1}}_{x_1}\underbrace{a}_{\alpha_1}\underbrace{a}_{\beta_2}\underbrace{a^{m-1}}_{y_2'},\underbrace{c}_{y_2\alpha_2}\underbrace{b}_{\beta_1 x_1'}) =$$
$$= (a^{n+m}, cb) = (z, w).$$

The splicing operation can be used as a basic tool for building a generative mechanism, called *splicing system*. Given a set of strings (axioms) and a set of splicing rules, the generated language will consist of the strings obtained as follows. Starting from the set of axioms, we iteratively use the splicing rules to splice strings from the set of axioms and/or strings obtained in preceding splicing steps.

If the classical notion of a set is used, we implicitly assume that, after splicing x and y and obtaining z and w, we may use again x or y as terms of the splicing, that is, the strings are not "consumed" by splicing. Similarly, there is no restriction on the number of copies of the newly obtained z and w. More realistic is the assumption that some of the strings are only available in a limited number of copies. In mathematical terms this translates in using, instead of sets, the notion of *multisets*, where one keeps track of the number of copies of a string at each moment.

In the style of [20], if $\mathbf{N}$ is the set of natural numbers, a multiset of Σ^* is a mapping $M : \Sigma^* \longrightarrow \mathbf{N} \cup \{\infty\}$ where for a word $w \in \Sigma^*$, $M(w)$ represents the number of occurrences of w. Here $M(w) = \infty$ is taken to mean that there are unboundedly many copies of the string w. The set $supp(M) = \{w \in \Sigma^* |\ M(w) \neq 0\}$ is called the *support* of M. With this modification of the notion of a set, we are now ready to introduce splicing systems.

Definition 3.1 *A splicing system is a quadruple $\gamma = (\Sigma, T, A, R)$, where Σ is an alphabet, $T \subseteq \Sigma$ is the terminal alphabet, A is a multiset over Σ^*, and $R \subseteq \Sigma^*\#\Sigma^*\$\Sigma^*\#\Sigma^*$ is the set of splicing rules.*

A splicing system γ defines a binary relation $\Longrightarrow_\gamma$ on the family of multisets of Σ^* as follows. For multisets M and M', $M \Longrightarrow_\gamma M'$ holds iff there exist $x, y \in supp(M)$ and $z, w \in supp(M')$ such that:

(i) $M(x) \geq 1$, $M(y) \geq 1$ if $x \neq y$ (resp. $M(x) \geq 2$ if $x = y$);

(ii) $(x, y) \longrightarrow_r (z, w)$ according to a splicing rule $r \in R$;

(iii) $M'(x) = M(x) - 1, M'(y) = M(y) - 1$ if $x \neq y$ (resp. $M'(x) = M(x) - 2$ if $x = y$);

(iv) $M'(z) = M(z) + 1, M'(w) = M(w) + 1$ if $z \neq w$ (resp. $M'(z) = M(z) + 2$ if $z = w$).

Informally, having a "set" of strings with a certain number (possibly infinite) of available copies of each string, the next "set" is produced by splicing two of the existing strings (by "existing" we mean that both strings have multiplicity at least 1). After performing a splicing, the terms of the splicing are consumed (their multiplicity decreases by 1), while the products of the splicing are added to the "set" (their multiplicity increases by 1).

The *language generated* by a splicing system γ is defined as

$$L(\gamma) = \{w \in T^* |\ A \Longrightarrow_\gamma^* M \text{ and } w \in \text{supp}(M)\},$$

where A is the set of axioms and $\Longrightarrow_\gamma^*$ is the reflexive and transitive closure of $\Longrightarrow_\gamma$.

For two families of type–0 languages, F_1, F_2, denote

$$H(F_1, F_2) = \{L(\gamma) \mid \gamma = (\Sigma, T, A, R), A \in F_1, R \in F_2\}.$$

(The notation $H(F_1, F_2)$ comes from the initial of Tom Head, who first introduced the notion of splicing.)

For example, $H(FIN, REG)$ denotes the family of languages generated by splicing systems where the set of axioms is finite and the set of rules is a regular language. A splicing system $\gamma = (\Sigma, T, A, R)$ with $A \in F_1, R \in F_2$, is said to be *of type* (F_1, F_2).

Splicing systems have been extensively studied in the literature. For example, the generative power of different types of splicing systems has been studied in [15], [17], [23], [25], [30], [50], [51], [52], [53]. Decidability problems have been tackled in [20]. Moreover, variations of the model have been considered: splicing systems with permitting/forbidding contexts in [15], linear and circular splicing systems in [31], [55], [67], splicing systems on graphs in [22], distributed splicing systems in [16], [18]. Earlier versions of Sections 1, 3, 4 can be found in [37], while for a survey of the main results on splicing the reader is referred to [32], [54].

4 The existence of DNA computers

Having defined a mathematical model of DNA computation, we now proceed to answer – for this model – the questions raised in Section 2. We start by showing that the splicing systems are computationally complete. By computational completeness of splicing we mean that every algorithm (effective procedure) can be carried out by a splicing system. It is obvious that this is not a mathematical definition of computational completeness. For an adequate definition, the intuitive notion of an algorithm (effective procedure) must be replaced by a formalized notion.

Since 1936, the standard accepted model of universal computation has been the Turing machine introduced in [64]. The Church–Turing thesis, the prevailing paradigm in computer science, states that no realizable computing device can be more powerful than a Turing machine. One of the main reasons that Church–Turing's thesis is widely accepted is that very diverse alternate formalizations of the class of effective procedures have all turned out to be equivalent to the Turing machine formalization. These alternate formalizations include Markov normal algorithms, Post normal systems, type–0 grammars, (which we have already considered in Section 3) as well as "computable" functions.

Showing that the splicing systems are computationally complete amounts thus, for example, to showing that the action of any computable function can be realized by a splicing system, where the term of computable function is detailed below (see [65]) .

Mappings of a subset of the Cartesian power set $\mathbf{N}^n$ into $\mathbf{N}$, where $n \geq 1$ and $\mathbf{N}$ is the set of natural numbers, are referred to as *partial functions*. If the domain of such a function equals $\mathbf{N}^n$, then the function is termed *total*. Examples of total functions (for different values of n) are:

The zero function: $Z(x_0) = 0$, for all $x_0 \in \mathbf{N}$.

The successor function: $S(x_0) = x_0 + 1$, for all $x_0 \in \mathbf{N}$.

The projection functions, for all i, n and $x_i \in \mathbf{N}$, $0 \leq i \leq n$:

$$U_i^{n+1}(x_0, x_1, \ldots, x_n) = x_i.$$

The class of partial recursive functions can be defined as the smallest class which contains certain basic functions and is closed under certain operations.

An $(n+1)$–ary function f is defined by the *recursion scheme* from the functions g and h if:

$$\begin{aligned} f(0, x_1, \ldots, x_n) &= g(x_1, \ldots, x_n) \\ f(n+1, x_1, \ldots, x_n) &= h(f(n, x_1, \ldots, x_n), n, x_1, \ldots, x_n). \end{aligned}$$

The operation of *composition* associates to the functions $h, g_0, \ldots, g_k$ the function f defined by:

$$f(x_0, x_1, \ldots, x_n) = h(g_0(x_0, \ldots, x_n), \ldots, g_k(x_0, \ldots x_n)),$$

which is defined exactly for those arguments $(x_0, \ldots, x_n)$ for which each of g_i, $0 \leq i \leq k$, as well the corresponding value of f is defined.

We say that f is defined by using the *minimization* operation on g, if

$$f(x_0, \ldots, x_n) = \begin{cases} (\mu y)[g(y, x_0, \ldots, x_n) = 0], & \text{if there is such a } y \\ \text{undefined}, & \text{otherwise}, \end{cases}$$

whose value, for a given $(x_0, \ldots, x_n)$, is the smallest value of y for which $g(y, x_0, \ldots, x_n) = 0$, and which is undefined if no such y exists.

A function f is defined *partial–recursively* if (i) it is the zero function, the successor function, or a projection function; (ii) it is defined by composing functions which are defined partial–recursively; (iii) it is defined by the recursion scheme from functions which are defined partial–recursively; or (iv) it is defined using the minimization operation on a function that is defined partial–recursively and is total.

It was proved that a function f is partial recursive if and only if there is a Turing machine which computes the values of f. On the other hand, according to the Church–Turing thesis, everything that can be effectively computed by any kind of device, can be computed by a Turing machine. As a consequence, partial recursive functions came to be known also under the name of *(effectively) computable functions*.

The formal language notion equivalent to the notion of a computable function is the notion of a *type–0 language*, i.e., a language generated by a grammar with no restriction imposed on its rewriting rules. One can prove that a language is of type–0 if and only if its characteristic function is computable. (By the characteristic function of a language $L \subseteq \Sigma^*$ we mean the function ϕ of Σ^* such that $\phi(w) = 1$ if $w \in L$, and ϕ is undefined otherwise.)

Recalling the notation $\mathcal{L}_0$ for the family of type–0 languages, the result proving the computational completeness of splicing systems can be formulated as follows.

Theorem 4.1 $\mathcal{L}_0 = H(FIN, FIN)$.

Informally, the theorem says that every type–0 language can be generated by a splicing system with finitely many axioms and finitely many rules. Given a type–0 grammar G generating the language $L(G)$, the proof of the inclusion $\mathcal{L}_0 \subseteq H(FIN, FIN)$ consists of two steps: (a) construct a splicing system $\gamma \in H(FIN, FIN)$, that simulates the rewriting rules of the grammar G, and (b) prove that the constructed splicing system generates the given type–0 language, i.e., show the equality $L(\gamma) = L(G)$. The reverse inclusion can be proved directly or by invoking the Church–Turing thesis. (The proof techniques used in Theorem 4.1 were suggested in [20] and first developed in [51]. The reader is referred to [23] for details.)

In terms of computable functions, Theorem 4.1 states that the work of any computable function can be carried out by a splicing system. Equivalently, Theorem 4.1 tells that everything that is Turing–computable can be computed also by this DNA model of computation. This answers the question as regards to what kinds of algorithms (effective procedures, computable functions) can be simulated by DNA computing devices based on splicing, and the answer is: all of them.

Theorem 4.1 shows that every program (computable function) can be simulated by a finite splicing system, but this does not say anything about the existence of a *programmable DNA computer* based on splicing. To this aim, it is necessary to find a *universal splicing system*, i.e., a system with all components but one fixed, able to behave as any given splicing system γ when a code of γ is introduced in the set of axioms of the universal system. Formally,

Definition 4.1 *Given an alphabet T and two families of type–0 languages, F_1, F_2, a construct*

$$\gamma_U = (\Sigma_U, T, A_U, R_U),$$

where Σ_U is an alphabet, $T \subseteq \Sigma_U$, $A_U \in F_1$, and $R_U \subseteq \Sigma_U^*\#\Sigma_U^*\$\Sigma_U^*\#\Sigma_U^*$, $R_U \in F_2$, is said to be a universal splicing system of type (F_1, F_2), if for every $\gamma = (\Sigma, T, A, R)$ of type (F_1', F_2'), $F_1', F_2' \subseteq \mathcal{L}_0$, there exists a language A_γ such that $A_U \cup A_\gamma \in F_1$ and $L(\gamma) = L(\gamma_U')$, where $\gamma_U' = (\Sigma_U, T, A_U \cup A_\gamma, R_U)$.

Note that the type of the universal system is fixed, but the universal system is able to simulate systems of any type (F_1', F_2'), when F_1' and F_2' are families of type–0 languages, that is, languages with a computable characteristic function. Based on Definition 4.1 we are now in position to state the main universality result.

Theorem 4.2 *For every given alphabet T there exists a splicing system, with finitely many axioms and finitely many rules, that is universal for the class of systems with the terminal alphabet T.*

The proof is based on Theorem 4.1 and on the existence of universal type–0 grammars (or, equivalently, universal Turing machines). For the details of the proof the reader is referred to [23]. Another proof, based on the fact that a language generated by a Post system can be generated by a splicing system, can be found in [21].

The interpretation of Theorem 4.2 from the point of view of DNA computing is that, theoretically, there exist *universal programmable DNA computers* based on the splicing operation. A program consists of a single string to be added to the axiom set of the universal computer. The program has multiplicity one, while an unbounded number of the other axioms is available. The "fixed" axioms of the computer can be interpreted as the "energy" that has to be constantly supplied to the DNA computer for running the programs. The only bio–operations used in these computers are splicing (cut/ligate) and extraction (which in mathematical terms amounts to the intersection of the result with T^*, where T is the terminal alphabet). In the case of splicing systems, we can conclude that Theorem 4.2 provides an affirmative answer to the second question posed in Section 2 with regards to the existence of programmable DNA computers.

Results analogous to Theorem 4.1 and Theorem 4.2 have been obtained for several variants of the splicing systems model presented in Section 3. For example, similar results hold if the condition of the axiom set to be a multiset is replaced by a control condition: a splicing rule is applicable only when certain strings, called *permitting contexts*, are present in the terms of splicing (see [23]).

Constructions showing how to simulate the work of a Turing machine by a DNA model of computation have also been proposed in [2], [5], [9], [11], [13], [57], [58], [59], [63], [70], [71]. In an optimistic way, one may think of an analogy between these results and the work on finding models of computation carried out in the 30's, which has laid the foundation for the design of the electronic computers. In a similar fashion, the results obtained about the models of DNA computation show that programmable DNA computers are not science fiction material, but the reality of the near future.

5 Down to earth: implementation techniques

As expected, the transition from the ideal world of mathematics to the empirical world of the molecular biology laboratory is full of challenges. Indeed, playing Pygmalion is not easy, as all scientists who attempted to blow life over their mathematical models of a DNA computer can attest. Despite the progress achieved, the main obstacles to creating a practical DNA computer still remain to be overcome. These obstacles are roughly of two types, [2]:

– *practical*, arising primarily from difficulties in dealing with large scale systems and in coping with errors;

– *theoretical*, concerning the versatility of DNA computers and their capacity to efficiently accomodate a wide variety of computational problems.

This section will touch upon both issues, focusing on practical implementation techniques of existing models of DNA computers. In the following we will point out possible pitfalls and complications that might arise in the process of implementing each of the bio-operations enumerated in Section 2. None of the encountered problems seems clearly insurmountable: being aware of their existence is the first step towards overcoming them.

To start from the beginning, *synthesis* of a DNA strand can sometimes result in the strand annealing to itself and creating a hairpin structure. Even the seemingly straightforward *mixing* operation can sometimes pose problems: if DNA is not handled gently, the sheer forces from pouring and mixing will fragment it. Also of concern for this operation is the amount of DNA which remains stuck to the walls of the tubes, pumps, pipette tips, etc., and is thus "lost" from the computation.

Hybridization has also to be carefully monitored because the thermodynamic parameters necessary for annealing to occur are sequence dependent. This is important because, depending on the conditions under which the DNA reactions occur, two oligonucleotides can hybridize without exact matching between their base pairs. *Hybridization stringency* refers to the number of complementary base pairs that have to match for DNA oligonucleotides to bond. It depends on reaction condiditions (salt concentration, temperature, relative percentage of A's and T's to G's and C's, duration of the reaction) and it increases with temperature. One solution for increasing hybridization stringency is the choice of good encodings for the input information of the problem, [19]. Another solution proposed in [7] to avoid self annealing and mismatches is encoding using specially chosen sequences as spacers that separate the information bits.

Amplification by PCR is used with the assumption that by maintaining a surplus of primer to template one can avoid undesired template-template interactions. As pointed out in [36], this assumption is not necessarily valid. Indeed, experimental evidence points to the possibility of the creation of complex structures like folded DNA, complexes between several strands and incorrect ligation products. This might further affect the accuracy of using the gel electrophoresis technique for *separation of strands by length.* Indeed, in the presence of complex structures, the mobility of the strands will not depend only on their length, as desired, but also on the DNA conformation (shape). As a possible solution, the use of denaturing or single-stranded gels for analysis is recommended in [36]. Moreover, by keeping concentrations low, heteroduplex (double-strands with mismatches) formation and template-template interactions can be minimized.

Separation of strands by length and *extraction* of strands containing a given pattern can also be inefficient, and this might pose problems with scale-up of the test-tube approach. An alternative methodology has been proposed in [46]: the set of oligonucleotides is initially attached to a surface (glass, silicon, gold, or beads). They are then subjected to bio-operations such as marking, unmarking and destruction (see Section 2), in order to obtain the desired solution. This method greatly reduces losses of DNA molecules that occur during extraction by affinity purification. Its drawbacks are that it relies on *marking* and *unmarking* which, in turn, assume specificity and discrimination of single-base mismatches. While these processes have proved reliable when using 15-mer sequences, they become more difficult for shorter or longer polynucleotide strands. Another problem is that the scale of the computation is restricted by the two-dimensional nature of the surface-based approach: one cannot reach too high an information storing density.

Extraction of those strands that contain some given pattern is not 100% efficient, and may at times inadvertently retain strands that *do not* contain the specified sequence. While the error rate is reasonable in case only a few extractions are needed, if the number of extractions is in the hundreds or thousands, problems arise even if 95% efficiency of extraction is assumed. Indeed, the probability of obtaining a strand encoding the solution, while at the same time obtaining no strands encoding illegal solutions is quite low. As another possible solution, in [5] the operation *remove* was proposed as a replacement for *extract*. The compound operation *remove* removes from a set of strands all strings that contain at least one occurrence of a given sequence. The operation is achieved by first marking all the strands that contain the given sequence as a substring and then destroying the marked strands (see Section 2). The advantage of the method is that the restriction enzymes used for the remove operation have a far lower error rate than extraction. One of the drawbacks is that, although the initial tube might contain multiple copies of each strand, after many *remove* operations the volume of material may be depleted below an acceptable empirical level. This difficulty can be avoided by periodic amplification by PCR.

Cutting (cleavage) of a DNA strand by a restriction endonuclease is also referred to as digestion of the DNA by that enzyme. The process may sometimes produce partial digestion products. One must test all protocols for the effectiveness of the restriction enzyme used, and it is often necessary to find means to remove undigested material. Similarly, the accuracy of *ligation* is high, but not perfect. A ligase may ligate the wrong molecule to a sticky end, if it bears a close resemblance to the target molecule.

Detection and *sequencing* conventionally require enzymatic reactions and gel electrophoresis that are expensive and laborious processes. A possible solution to these drawbacks is using a technique that achieves sequencing by hybridization, offering a one-step automated process for reading the output, [47]. In this method, target strands are hybridized to a complete set of oligonucleotides synthesized on a flat solid surface (for example an array containing all the possible 8-mers) and then the target is reconstructed from the hybridization data obtained. However, to avoid errors arising from self-annealing, a restricted genetic alphabet is recommended with this method, using only two of the four bases. In this way, the test tube contents would be resistant to intramolecular reactions but not to intermolecular reactions.

Besides the accuracy of bio-operations, another peril of the implementation of DNA computations is the fact that the size of the problem influences the concentration of reactants, and this, in turn, has an effect on the rate of production and quality of final reaction products. In [40], an analysis of Adleman's experiment showed that an exponential loss of concentration occurs even on sparse digraphs, and that this loss of concentration can become a significant consideration for problems not much larger than those solved by Adleman. For volume decreasing DNA algorithms, an error-resistant solution seems to be the repeated application of PCR to the intermediate products, [14]. However, this cannot always be the solution, as not all algorithms are volume decreasing. Indeed, as pointed out in [29], one barrier to scalable DNA computation is the weight of DNA. In some cases, [3], to achieve the desirable error rate, approximately 23 Earth masses of DNA were needed. Clearly, this is not an acceptable situation, and a combination of *algorithm transformations* might be required to reduce the amount of DNA.

Indeed, until now we have mainly been concerned with the perils and pitfalls of the biotechnology used to implement the bio-operations. This might have given the skewed impression that the only way to advance the DNA computing research is to wait for progresses in biotechnology. The paragraphs below point out that a complementary approach to improving the accuracy of DNA computing should involve development of novel programming techniques, the use of probabilistic algorithms and other modifications of the classical mathematical problem-solving tools.

The idea of algorithm transformation has already been suggested in the cases where the DNA algorithm amounts to sorting a huge library of initial candidate solution complexes into those which encode a solution to a problem and those which do not. In such cases, (including Adleman's experiment) the main occurring errors are of the following types: *false positives*, *false negatives* and *strand losses*. A possible solution to these types of problems is to change the algorithms by using intelligent space and time trade-offs. In [2], a basic transformation called *repeating* was introduced. The transformation makes use of a slowdown factor of M, proposing for a given algorithm A a new algorithm A' that has a smaller error rate than A, as follows:

- Repeat M times:
  - Run A on input I, producing tubes Y and N.
  - Discard tube N and rename tube Y as tube I.
- Return tube I as the "Yes" tube and an empty tube as "No".

This approach is of value when the original algorithm A is known to place very reliably good sequences into the "Yes" tube (i.e. low false negatives) but to often also place bad sequences into the "Yes" tube (i.e. high false positives). A corresponding variation of the above transformation can be used if the algorithm is known to have high false negatives and low false positives. This and similar methods are used in [3], [59]: the model employs *stickers* (short complementary sequences), to mark the "on" bits, while the unmarked bits are considered "'off". The advantage of the proposed sticker model is that it does not rely on short lived materials as enzymes and that it does not involve processes that are costly in terms of energy, such as PCR. To put things in perspective, an opposite approach that relies only on PCR to solve problems has proved to be less error prone, [41].

Section 4 showed that several major roadblocks have been overcome at the theoretical level of DNA computation research, and suggested that real applications of molecular computation may be feasible in the future. This section, which discusses implementation techniques and the associated error rates, indicates that many substantial engineering challenges remain at almost all stages. However, we want to point out that the issues of actively monitoring and adjusting the concentration of reactants, as well as fault tolerance, are all addressed by biological systems in nature: cells need to control the concentrations of various compounds, to arrange for rare molecules to react, and they need to deal with undesirable byproducts of their own activity. There is no reason why, when these problems have successfully been dealt with in vivo, they should not have solutions in vitro. As a step in this direction, in [40] a mechanism is suggested, based on membranes that separate volumes (vesicles) and on active systems that transport selected chemicals across membranes. Moreover, [4] and [68] suggest how familiar computer design principles for electronic computers can be exploited to build molecular computers.

Having addressed the practical obstacles, we briefly return to the theoretical questions that arise when contemplating the construction of a molecular computer. One of the most debated theoretical issues is whether a DNA computer will outperform and finally replace the electronic computer in *all* practical applications, or whether there will be only special classes of applications for which DNA computers will be preferred. DNA is an attractive medium for computation because of its potential parallelism resulting from the possibility of having up to 10^{18} bytes represented in a single test tube of DNA solution. Also attractive is its high information storage density of 10^{23} bytes per kilogram, which dwarfs the human genome, 10^9 bytes, [68]. These features make it tempting to conjecture that a DNA computer will be faster and more efficient than an electronic one, no matter what the application. While this might be the case, the following suggests that perhaps we should not expect it to be so, even in theory.

To draw a parallel, in many realms of life, where the processing of large amounts of data quickly, efficiently and in parallel is involved, humans outshine the fanciest computers. Example of such situations are our daily social encounters of persons, when the computers inside our skulls evaluate, in a split-second, data fed by our sense organs about their shape, size, colour, sound, smell, posture, movement and expression, [48]. The information is then processed at lightning speed and out comes the answer: friend or stranger, to smile or not to smile, to touch or not to touch. The most modern electromechanical robots pale in comparison, having difficulties even in processing the parallel sensory input necessary for moving in a less-than-clumsy fashion, let alone in socializing. On the other hand, when faced with the task of adding up 1,000,000 fifty-digit numbers, an electronic computer will outperform the human brain anytime.

The comparison suggests that perhaps we should not expect the DNA computer to completely replace an electronic computer. Those problems whose algorithms can be highly parallelized, could possibly become the domain of DNA computers, while the ones whose algorithms are inherently sequential might remain the speciality of electronic computers. Indeed, the search of a "killer application", that is, an application for which the DNA based solution has obvious advantages over the electronic one, is one of the main directions of research in the area. Along with it, substantial effort is being invested in testing the bio-operations for implementability, and in finally choosing a set of "primitives" that will form the basics of a *molecular high level programming language.* Last, but not least, it is expected that the experimental work of probing the limits of biomolecular computation will lead to insights into the information handling capacities of cellular organisms.

6 Meta–thoughts on biomathematics

We have seen in Sections 2 and 5 that the bio–operations are quite different from the usual arithmetical operations. Indeed, even more striking than the quantitative differences between a virtual DNA computer and an electronic computer are the qualitative differences between the two.

DNA computing is a new way of thinking about computation altogether. Maybe this is how nature does mathematics: not by adding and subtracting, but by cutting and pasting, by insertions and deletions. Perhaps the primitive functions we currently use for computation are just as dependent on the history of humankind, as the fact that we use base 10 for counting is dependent on our having ten fingers. In the same way humans moved on to counting in other bases, maybe it is time we realized that there are other ways to compute besides the ones we are familiar with.

The fact that phenomena happening inside living organisms (copying, cutting and pasting of DNA strands) could be computations in disguise suggests that life itself may consist of a series of complex computations. As life is one of the most complex natural phenomena, we could generalize by conjecturing the whole cosmos to consist of computations. The differences between the diverse forms of matter would then only reflect various degrees of computational complexity, with the qualitative differences pointing to huge computational speed–ups. From chaos to inorganic matter, from inorganic to organic, and from that to consciousness and mind, perhaps the entire evolution of the universe is a history of the ever–increasing complexity of computations.

Of course, the above is only a hypothesis, and the enigma whether modern man is "homo sapiens" or "homo computans" still awaits solving. But this is what makes DNA computing so captivating. Not only may it help compute faster and more efficiently, but it stirs the imagination and opens deeper philosophical issues. What can be more mesmerizing than something that makes you dream?

To a mathematician, DNA computing tells that perhaps mathematics is the foundation of all there is. Indeed, mathematics has already proven to be an intrinsic part of sciences like physics and chemistry, of music, visual arts (see [33]) and linguistics, to name just a few. The discovery of DNA computing, indicating that mathematics also lies at the root of biology, makes one wonder whether mathematics isn't in fact the core of all known and (with noneuclidean geometry in mind) possible reality.

Maybe indeed, Plato was right: Truth, Beauty and Good are one and the same. Maybe indeed, [56], the material things are mere instances of "ideas" that are everlasting, never being born nor perishing. By intimating that – besides everything else – mathematics lies at the very heart of life, DNA computing suggests we take Plato's philosophy one step further: the eternal "ideas" reflected in the ephemeral material world could be mathematical ones.

If this were the case, and the quintessence of reality is the objective world of mathematics, then we should feel privileged to be able to contemplate it.

Acknowledgements. This article has benefited from discussions with Leonard Adleman, Mark Gijzen, Greg Gloor, Tom Head, Jarkko Kari, Gheorghe Paun, Clive Reis, Arto Salomaa, Gabriel Thierrin and Gary Walsh, to all of whom I wish to express my gratitude. Special thanks are due to Luiz Ernesto Merkle for the graphical illustrations of molecular processes. It would be impossible to separately point out to all written sources that have influenced this paper. Adopting therefore Seneca's maxim *"Whatever is well said by anyone belongs to me.", [62]*, I want to thank all the authors in the references.

References

[1] L.Adleman. Molecular computation of solutions to combinatorial problems. *Science* v.266, Nov.1994, 1021–1024.

[2] L.Adleman. On constructing a molecular computer, *ftp://usc.edu/pub/csinfo/papers/adleman.*

[3] L.Adleman, P.Rothemund, S.Roweis, E.Winfree. On applying molecular computation to the Data Encryption Standard. *2nd DIMACS workshop on DNA based computers*, Princeton, 1996, 28–48.

[4] J.Amenyo. Mesoscopic computer engineering: automating DNA-based molecular computing via traditional practices of parallel computer architecture design. *2nd DIMACS workshop on DNA based computers*, Princeton, 1996, 217–235.

[5] M.Amos, A.Gibbons, D.Hodgson. Error-resistant implementation of DNA computation. *2nd DIMACS workshop on DNA based computers*, Princeton, 1996, 87–101.

[6] E.Baum. Building an associative memory vastly larger than the brain. *Science*, vol.268, April 1995, 583–585.

[7] E.Baum. DNA sequences useful for computation. *2nd DIMACS workshop on DNA based computers*, Princeton, 1996, 122–127.

[8] E.Baum. D.Boneh. Running dynamic programming algorithms on a DNA computer. *2nd DIMACS workshop on DNA based computers*, Princeton, 1996, 141–147.

[9] D.Beaver. The complexity of molecular computation. *http://www.transarc.com/˜beaver/research/alternative/molecute/molec.html.*

[10] D. Beaver. Computing with DNA. *Journal of Computational Biology*, (2:1), Spring 1995.

[11] D.Beaver. A universal molecular computer. *http://www.transarc.com/~beaver/research/alternative/molecute/molec.html.*

[12] D.Boneh, R.Lipton, C.Dunworth. Breaking DES using a molecular computer. *http://www.cs.princeton.edu/~dabo.*

[13] D.Boneh, R.Lipton, C.Dunworth, J.Sgall. On the computational power of DNA. *http://www.cs.princeton.edu/~dabo.*

[14] D.Boneh, C.Dunworth, J.Sgall, R.Lipton. Making DNA computers error resistant. *2nd DIMACS workshop on DNA based computers*, Princeton, 1996, 102–110.

[15] E.Csuhaj–Varju, R.Freund, L.Kari, G.Paun. DNA computing based on splicing: universality results. *First Annual Pacific Symposium on Biocomputing*, Hawaii, 1996, also *http://www.csd.uwo.ca/~lila.*

[16] E.Csuhaj–Varju, L.Kari, G.Paun. Test tube distributed systems based on splicing. *Computers and AI*, 15, 2-3(1996), 211–232.

[17] K.Culik, T.Harju. Splicing semigroups of dominoes and DNA. *Discrete Applied Mathematics*, 31(1991) 261–277.

[18] J.Dassow, V.Mitrana. Splicing grammar systems. *Computers and AI.* To appear.

[19] R.Deaton, R.Murphy, M.Garzon, D.Franceschetti, S.Stevens. Good encodings for DNA-based solutions to combinatorial problems. *2nd DIMACS workshop on DNA based computers*, Princeton, 1996, 131–140.

[20] K.L.Deninnghoff, R.W.Gatterdam. On the undecidability of splicing systems. *International Journal of Computer Mathematics*, 27(1989) 133–145.

[21] C.Ferretti, S.Kobayashi, T.Yokomori. DNA splicing systems and Post Systems. *First Annual Pacific Symposium on Biocomputing*, Hawaii, 1996.

[22] R.Freund. Splicing systems on graphs. *Proc.Intelligence in Neural and Biological Systems*, IEEE Press, May 1995, 189–194.

[23] R.Freund, L.Kari, G.Paun. DNA computing based on splicing: the existence of universal computers. T.Report 185–2/FR–2/95, TU Wien, Institute for Computer Languages, 1995, and *Journal of the ACM*, to appear. Also at *http://www.csd.uwo.ca/~lila.*

[24] M.Garey, D.Johnson. *Computers and Intractability. A Guide to the Theory of NP–completeness.* W.H.Freeman and Company, San Francisco, 1979.

[25] R.W.Gatterdam. Splicing systems and regularity. *International Journal of Computer Mathematics*, 31(1989) 63–67.

[26] R.W.Gatterdam. DNA and twist free splicing systems, in *Words, Languages and Combinatorics II*, Eds.:M.Ito and H.Jürgensen, World Scientific Publishers, Singapore, 1994, 170–178.

[27] D.K.Gifford. On the path to computation with DNA. *Science* 266(Nov.1994), 993–994.

[28] F.Guarnieri, C.Bancroft. Use of a horizontal chain reaction for DNA-based addition. *2nd DIMACS workshop on DNA based computers*, Princeton, 1996, 249–259.

[29] J.Hartmanis. On the weight of computations. *Bulletin of the European Association of Theoretical Computer Science*, 55(1995), 136–138.

[30] T.Head. Formal language theory and DNA: an analysis of the generative capacity of recombinant behaviors. *Bulletin of Mathematical Biology*, 49(1987) 737–759.

[31] T.Head. Splicing schemes and DNA, in: *Lindenmayer systems – Impacts on Theoretical Computer Science, Computer Graphics and Developmental Biology.* Springer Verlag, Berlin 1992, 371–383.

[32] T.Head, G.Paun, D.Pixton. Language theory and genetics. Generative mechanisms suggested by DNA recombination. In *Handbook of Formal Languages* (G.Rozenberg, A.Salomaa eds.), Springer Verlag, 1996.

[33] D.Hofstadter. *Gödel, Escher, Bach: an eternal golden braid*, New York, Basic Books, 1979.

[34] F.Hoppensteadt. Getting started in mathematical biology. *Notices of the AMS*, vol.42, no.9, Sept.1995.

[35] N.Jonoska, S.Karl. A molecular computation of the road coloring problem. *2nd DIMACS workshop on DNA based computers*, Princeton, 1996, 148-158.

[36] P.Kaplan, G.Cecchi, A.Libchaber. DNA-based molecular computation: template–template interactions in PCR. *2nd DIMACS workshop on DNA based computers*, Princeton, 1996, 159–171.

[37] L.Kari. DNA computers: tomorrow's reality. *Bulletin of The European Association of Theoretical Computer Science*, 59(1996), 256–266.

[38] L.Kari, G.Thierrin. Contextual insertions/deletions and computability. To appear in *Information and Computation.*

[39] J.Kendrew et al., eds. *The Encyclopedia of Molecular Biology*, Blackwell Science, Oxford, 1994.

[40] S.Kurtz, S.Mahaney, J.Royer, J.Simon. Active transport in biological computing. *2nd DIMACS workshop on DNA based computers*, Princeton, 1996, 111–121.

[41] T.Leete, M.Schwartz, R.Williams, D.Wood, J.Salem, H.Rubin. Massively parallel DNA computation: expansion of symbolic determinants. *2nd DIMACS workshop on DNA based computers*, Princeton, 1996, 49–66.

[42] R.Lipton. DNA solution of hard computational problems. *Science*, vol.268, April 1995, 542–545.

[43] R.Lipton. Using DNA to solve NP–complete problems. *http://www.cs.princeton.edu/˜rjl.*

[44] R.Lipton. Using DNA to solve SAT. *http://www.cs.princeton.edu/˜rjl.*

[45] R.Lipton. Speeding up computations via molecular biology. Manuscript.

[46] Q.Liu, Z.Guo, A.Condon, R.Corn, M.Lagally, L.Smith. A surface-based approach to DNA computation. *2nd DIMACS workshop on DNA based computers*, Princeton, 1996, 206–216.

[47] K.Mir. A restricted genetic alphabet for DNA computing. *2nd DIMACS workshop on DNA based computers*, Princeton, 1996, 128–130.

[48] D.Morris. *Intimate Behaviour*, Random House, New York, 1971.

[49] J.Oliver. Computation with DNA: matrix multiplication. *2nd DIMACS workshop on DNA based computers*, Princeton, 1996, 236–248.

[50] G.Paun. On the splicing operation. *Discrete Applied Mathematics*, 70(1996), 57–79.

[51] G.Paun. On the power of the splicing operation. *International Journal of Computer Mathematics*, 59(1995), 27–35.

[52] G.Paun. Regular extended H systems are computationally universal. *Journal of Automata, Languages and Combinatorics*, 1, 1(1996), 27–36.

[53] G.Paun, G.Rozenberg, A.Salomaa. Computing by splicing. To appear in *Theoretical Computer Science.*

[54] G.Paun, A.Salomaa. DNA computing based on the splicing operation. *Mathematica Japonica*, vol.43, no.3, 1996, 607–632.

[55] D.Pixton. Linear and circular splicing systems. *Proc.Intelligence in Neural and Biological Systems*, IEEE Press, May 1995, 181–188.

[56] Plato. *Great Dialogues of Plato.* The New American Library, New York, 1956.

[57] J.Reif. Parallel molecular computation: models and simulation. To appear in SPAA'95, also at *http://www.cs.duke.edu/˜reif/HomePage.html.*

[58] P.Rothemund. A DNA and restriction enzyme implementation of Turing machines. Abstract at *http://www.ugcs.caltech.edu/˜pwkr/oett.html.*

[59] S. Roweis, E. Winfree, R. Burgoyne, N. Chelyapov, M. Goodman, P. Rothemund, L. Adleman. A sticker based architecture for DNA computation. *2nd DIMACS workshop on DNA based computers*, Princeton, 1996, 1–27.

[60] A.Salomaa. *Formal Languages.* Academic Press, New York, 1973.

[61] N.Seeman et al. The perils of polynucleotides: the experimental gap between the design and assembly of unusual DNA structures. *2nd DIMACS workshop on DNA based computers*, Princeton, 1996, 191–205.

[62] L.Seneca. *Letters from a stoic*, Harmondsworth, Penguin, 1969.

[63] W.Smith, A.Schweitzer. DNA computers in vitro and in vivo, *NEC Technical Report 3/20/95.*

[64] A.M.Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc.London Math Soc.*, Ser.2, 42(1936), 230–265.

[65] A.Yasuhara. *Recursive Function Theory and Logic.* Academic Press, New York, 1971.

[66] T.Yokomori, S.Kobayashi. DNA evolutionary linguistics and RNA structure modeling: a computational approach. *Proc.Intelligence in Neural and Biological Systems*, IEEE Press, May 1995, 38–45.

[67] T.Yokomori, S.Kobayashi, C.Ferretti. On the power of circular splicing systems and DNA computability. Rep. CSIM–95–01996, University of Electro–Communications, Dept.of Comp.Sci. and Information Mathematics, Chofu, Tokyo 182, Japan.

[68] R.Williams, D.Wood. Exascale computer algebra problems interconnect with molecular reactions and complexity theory. *2nd DIMACS workshop on DNA based computers*, Princeton, 1996, 260–268.

[69] E.Winfree. Complexity of restricted and unrestricted models of molecular computation. *http://dope.caltech.edu/winfree/DNA.html.*

[70] E.Winfree. On the computational power of DNA annealing and ligation. *http://dope.caltech.edu/winfree/DNA.html.*

[71] E.Winfree, X.Yang, N.Seeman. Universal computation via self-assembly of DNA: some theory and experiments. *2nd DIMACS workshop on DNA based computers*, Princeton, 1996, 172–190.

[72] W.Wordsworth. The tables turned. In *Wordsworth's Poems*, vol.1996, Ed.P.Wayne, Dent, London, 1965.

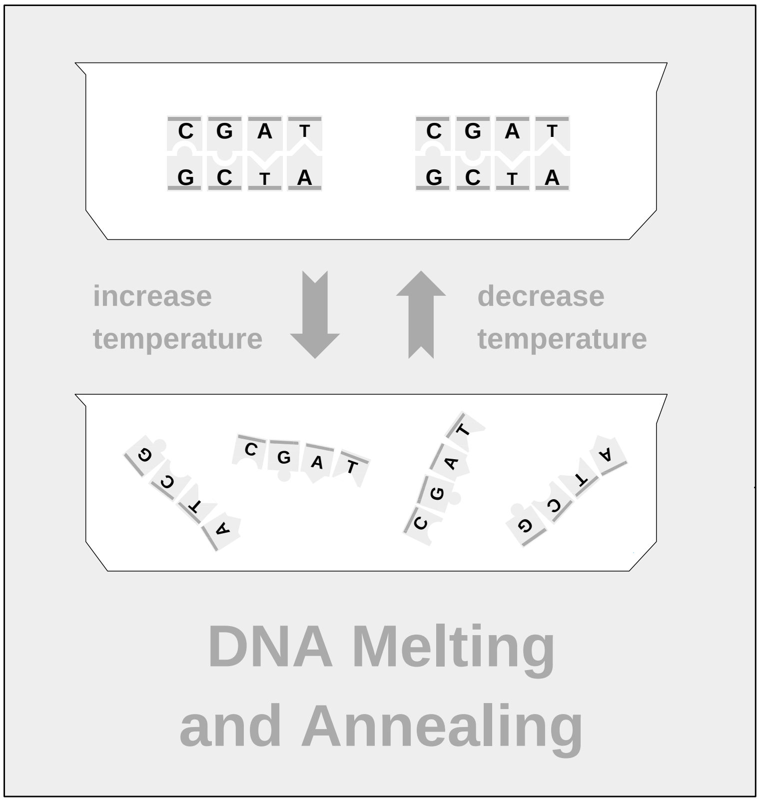

Figure 1: DNA melting and annealing

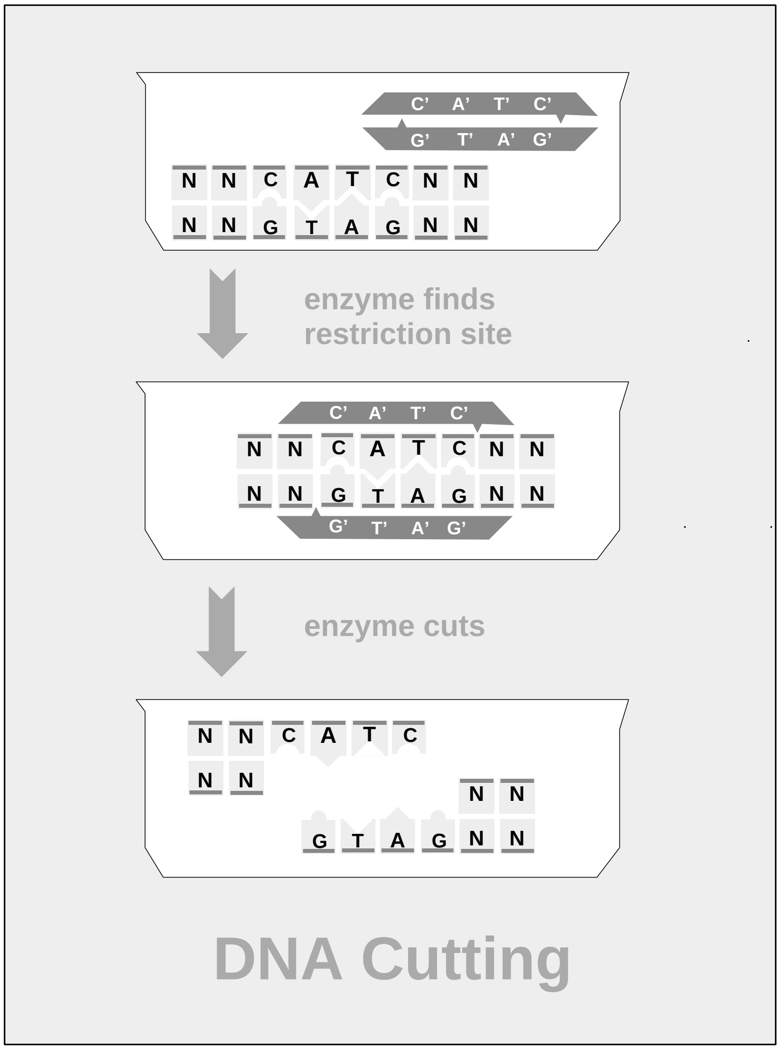

Figure 2: DNA cutting

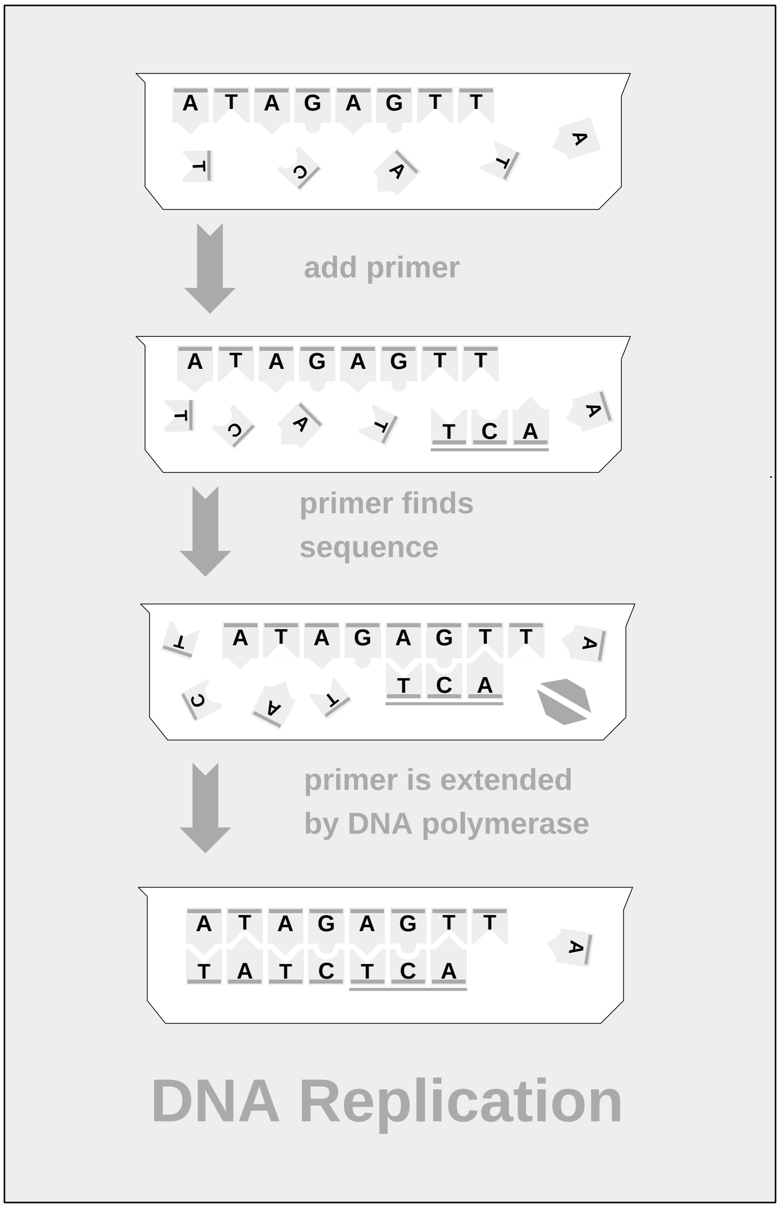

Figure 3: DNA replication

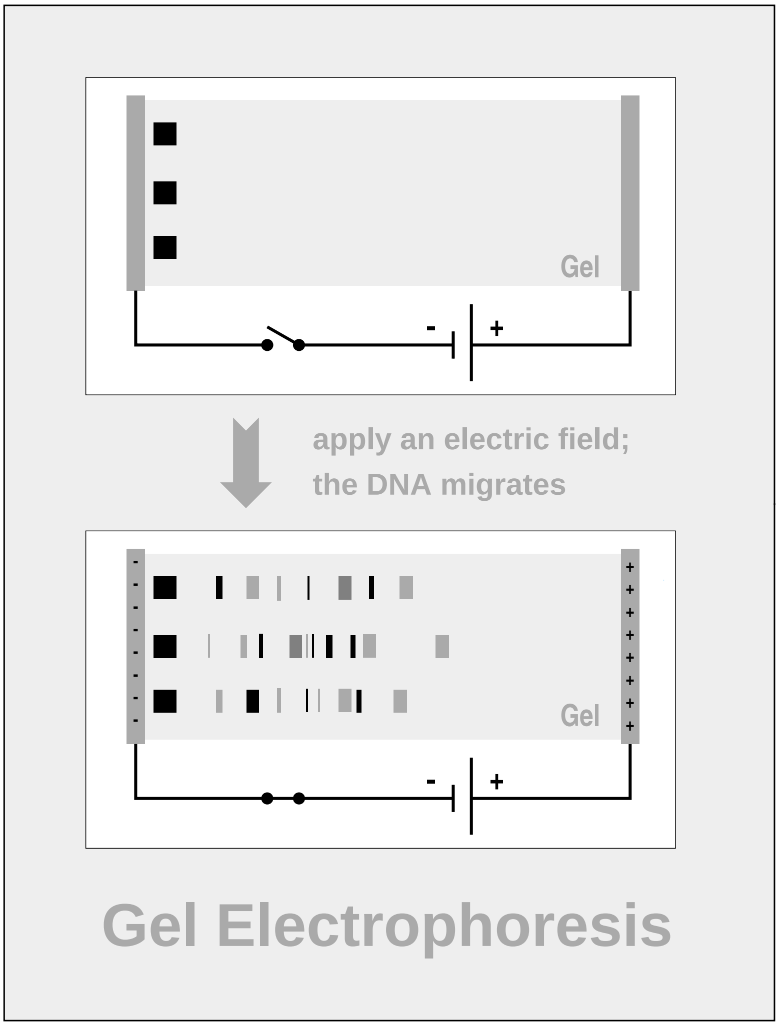

Figure 4: Gel electrophoresis

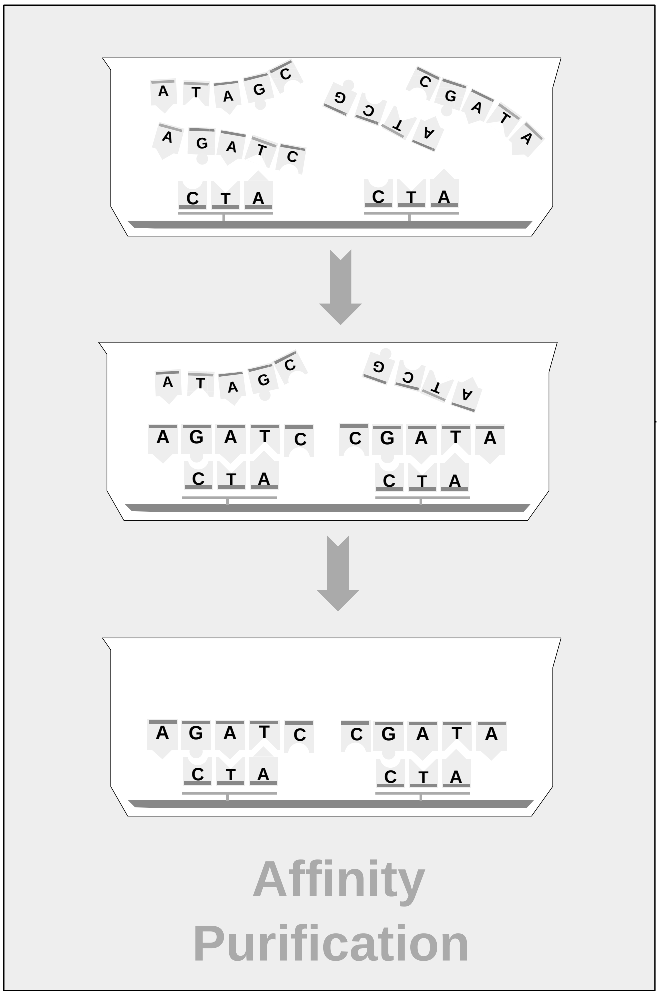

Figure 5: Affinity purification

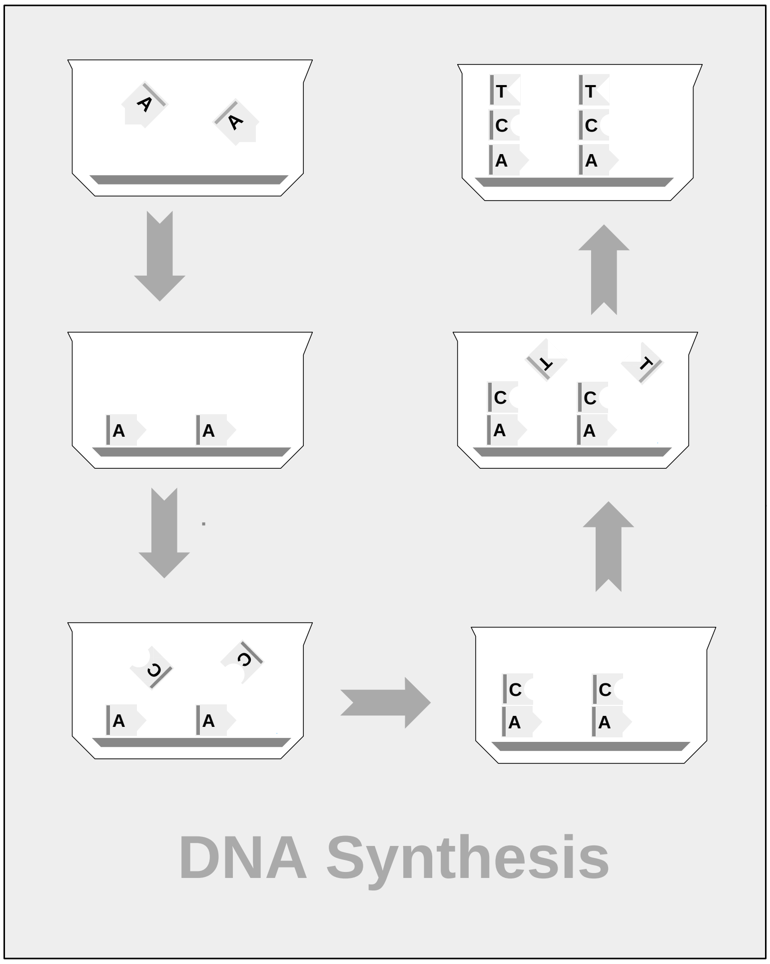

Figure 6: DNA synthesis

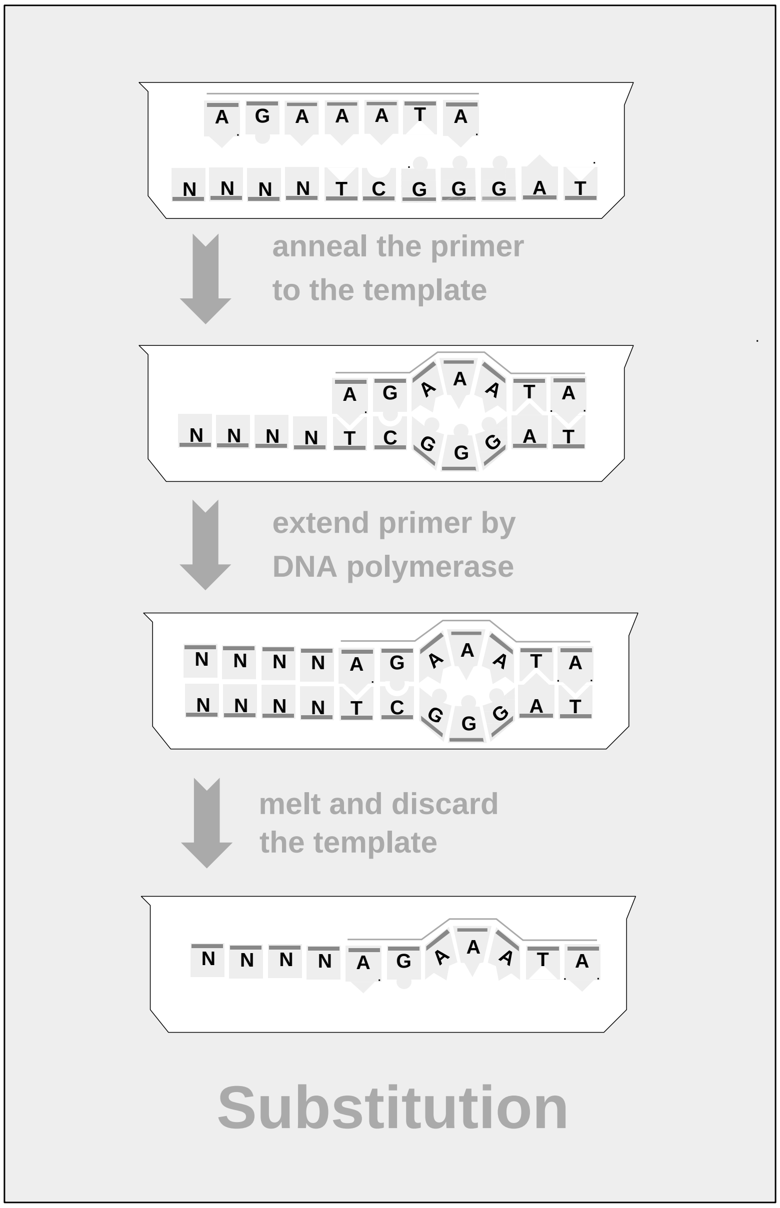

Figure 7: Substitution