Fast Parallel Molecular Algorithms for DNA-Based Computation: Factoring Integers

Weng-Long Chang, Minyi Guo*, *Member, IEEE*, and Michael Shan-Hui Ho

***Abstract*—The RSA public-key cryptosystem is an algorithm that converts input data to an unrecognizable encryption and converts the unrecognizable data back into its original decryption form. The security of the RSA public-key cryptosystem is based on the difficulty of factoring the product of two large prime numbers. This paper demonstrates to factor the product of two large prime numbers, and is a breakthrough in basic biological operations using a molecular computer. In order to achieve this, we propose three DNA-based algorithms for parallel subtractor, parallel comparator, and parallel modular arithmetic that formally verify our designed molecular solutions for factoring the product of two large prime numbers. Furthermore, this work indicates that the cryptosystems using public-key are perhaps insecure and also presents clear evidence of the ability of molecular computing to perform complicated mathematical operations.**

***Index Terms*—Biological parallel computing, DNA-based algorithms, DNA-based computing, factoring integers, RSA public-key cryptosystem.**

I. INTRODUCTION

THE RSA public-key cryptosystem [34] is an algorithm that converts input data to an unrecognizable encryption, and converts the unrecognizable data back into its original decryption form. The construction of the RSA public-key cryptosystem is based on the ease of finding large prime numbers. The security for the cryptosystem using public-key is based on the difficulty of factoring the product of two large prime numbers. The RSA public-key cryptosystem is the most popular cryptosystem. No method in a reasonable amount of time can be applied to break the RSA public-key cryptosystem.

Feynman proposed molecular computation in 1961, but his idea was not implemented by experiment for a few decades [37]. In 1994 Adleman [2] succeeded in solving an instance of the Hamiltonian path problem in a test tube, just by handling DNA strands. Lipton [3] demonstrated that the Adleman techniques could be used to solve the satisfiability problem (the first NP-complete problem). Adleman *et al.* [14] proposed *sticker* for enhancing the error rate of hybridization.

Through advances in molecular biology [1], it is now possible to produce roughly 10^{18} DNA strands that fit in a test tube. Those 10^{18} DNA strands can also be applied to represent 10^{18} bits of information. In the future (perhaps after many years) if biological operations can be applied to deal with a tube with 10^{18} DNA strands and they are run without errors, then 10^{18} bits of information can simultaneously be correctly processed. Hence, in the future, it is possible that biological computing can provide a huge amount of parallelism for dealing with many computationally intensive problems in the real world.

The fastest super computers currently available can execute approximately 10^{12} integer operations per second. This implies that (128×10^{12}) bits of information can be simultaneously processed in a second. The fastest super computers can process (128×10^{15}) bits of information in 1000 seconds. The *extract* operation is one of basic biological operations of the longest execution time. An *extract* operation could be approximately done in 1000 s [12]. In the future, if an *extract* operation can be used to deal with a tube with 10^{18} DNA strands and it is run without errors, then 10^{18} bits of information can simultaneously be correctly processed in 1000 s. If it becomes true in the future, then basic biological operations will perhaps be faster than the fastest super computer in the future. In [12], it was pointed out that storing information in molecules of DNA allows for an information density of approximately 1 bit/nm^3. Videotape is a kind of traditional storage media and its information density is approximately 1 bit/10^{12} nm^3. This implies that an information density in molecules of DNA is better than that of traditional storage media.

In this paper, we first construct solution spaces of DNA strands for encoding every integer of k bits. By using basic biological operations, we then develop DNA-based algorithms for a parallel subtractor, a parallel comparator, and a parallel divider, respectively, to factor the product of two large prime numbers of k bits. We also show that cryptosystems based on the dramatic difference between the ease of finding large prime numbers of k bits and the difficulty of factoring the product of two large prime numbers of k bits can be broken. Furthermore, this work presents clear evidence of molecular computing ability to finish parallel mathematical operations.

The rest of this paper is organized as follows. Section II first introduces DNA models of computation proposed by Adleman *et al.* and compares them with other models. Section III introduces the DNA program to factor the product of two large prime numbers of k bits for solution spaces of DNA strands. Discussion and conclusion are drawn in Section IV and Section V, respectively.

Manuscript received April 7, 2004; revised November 25, 2004. This work was supported in part by the Republic of China National Science Foundation under Grant NSC-92-2213-E-218-026 and Japan JSPS Grant-in-Aid for Scientific Research under Grant (C) 14580386. *Asterisk indicates corresponding author.*

W.-L. Chang and M. S.-H. Ho are with the Department of Information Management, Southern Taiwan University of Technology, Tainan, Taiwan, R.O.C. (E-mail: changwl@mail.stut.edu.tw; MHoInCerritos@yahoo.com).

*M. Guo is with the School of Computer Science and Engineering, University of Aizu, Aizu-Wakamatsu City 965-8580, Japan (E-mail: minyi@u-aizu.ac.jp).

Digital Object Identifier 10.1109/TNB.2005.850474

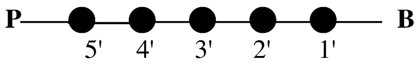

Fig. 1. A schematic representation of a nucleotide.

II. Background

In this section we review the basic structure of the DNA molecule and then discuss available techniques for dealing with DNA that will be used to solve the problem of factoring integers. Simultaneously, several well-known DNA models are compared.

A. The Structure of DNA

From [1], [16], DNA (*DeoxyriboNucleic Acid*) is the *molecule* that plays the main role in DNA-based computing. In the biochemical world of large and small *molecules, polymers,* and *monomers*, DNA is a polymer, which is strung together from monomers called *deoxyriboNucleotides*. The monomers used for the construction of DNA are deoxyribonucleotides. Each deoxyribonucleotide contains three components: a *sugar*, a *phosphate* group, and a *nitrogenous* base. The sugar has five carbon atoms—for the sake of reference there is a fixed numbering of them. Because the base also has carbons, to avoid confusion the carbons of the sugar are numbered from $1'$ to $5'$ (rather than from one to five). The phosphate group is attached to the $5'$ carbon, and the base is attached to the $1'$ carbon. Within the sugar structure there is a *hydroxyl* group attached to the $3'$ carbon.

Distinct nucleotides are detected only with their bases, which come in two sorts: *purines* and *pyrimidines*. Purines include *adenine* and *guanine*, abbreviated A and G. Pyrimidines contain *cytosine* and *thymine*, abbreviated C and T. Because nucleotides are distinguished solely from their bases, they are simply represented as A, G, C, or T nucleotides, depending upon the kinds of base that they have. The structure of a nucleotide, cited from [16], is illustrated (in a very simplified way) in Fig. 1. In Fig. 1, B is one of the four possible bases (A, G, C, or T), P is the phosphate group, and the rest (the "stick") is the sugar base (with its carbons enumerated $1'$ through $5'$).

Nucleotides can be linked together in two different ways [1], [16]. The first method is that the $5'$-phosphate group of one nucleotide is joined with $3'$-hydroxyl group of the other forming a *phosphodiester* bond. The resulting molecule has the $5'$-phosphate group of one nucleotide, denoted as $5'$ end, and the $3'$-OH group of the other nucleotide available, denoted as $3'$ end, for bonding. This gives the molecule the *directionality*, and we can talk about the direction of $5'$ end to $3'$ end or $3'$ end to $5'$ end. The second way is that the base of one nucleotide interacts with the base of the other to form a *hydrogen* bond. This bonding is the subject of the following restriction on the base pairing: A and T can pair together, and C and G can pair together—no other pairings are possible. This pairing principle is called the Watson–Crick complementarity (named after J. D. Watson and F. H. C. Crick, who deduced the famous double helix structure of DNA in 1953 and won the Nobel Prize for the discovery).

A DNA strand is essentially a sequence (polymer) of four types of nucleotides detected by one of four bases they contain. Two strands of DNA can form (under appropriate conditions) a double strand, if the respective bases are the Watson–Crick complements of each other—A matches T and C matches G; also $3'$ end matches $5'$ end. The length of a single-stranded DNA is the number of nucleotides composing the single strand. Thus, if a single stranded DNA includes 20 nucleotides, then we say that it is a 20 mer (i.e., it is a polymer containing 20 monomers). The length of a double-stranded DNA (where each nucleotide is base paired) is counted in the number of base pairs. Thus, if we make a double-stranded DNA from a single stranded 20 mer, then the length of the double stranded DNA is 20 base pairs, also written 20 bp. Hybridization is a special technology term for the pairing of two single DNA strands to make a double helix and also takes advantages of the specificity of DNA base pairing for the detection of specific DNA strands. (For more discussions of the relevant biological background, refer to [1] and [16]).

B. Adleman's Experiment for Solving the Hamiltonian Path Problem

Assume a directed graph $G = (V, E)$, where V and E are the set of vertices and the set of edges respectively. In general, the Hamiltonian path problem consists of deciding whether G has a Hamiltonian path or not. G with designed 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 $e_1, \ldots, e_z$ (that is, a "path"), which begins at v_{in}, ends at v_{out}, and enters every other vertex exactly once [2].

Adleman's experiment is used to solve the Hamiltonian path problem for a directed $G = (V, E)$, where $V = \{v_0, v_1, v_2, v_3, v_4, v_5, v_6\}$ and $E = \{(v_0, v_3), (v_0, v_1), (v_0, v_6), (v_2, v_3), (v_2, v_1), (v_3, v_2), (v_3, v_4), (v_4, v_1), (v_4, v_5), (v_5, v_2), (v_5, v_6)\}$ [2]. The first step of Adleman's experiment is to generate random paths through the directed graph G. To generate random paths, each vertex v_i in V for $0 \leq i \leq 6$ was associated with a random 20-mer sequence of DNA denoted O_i. For each edge (v_i, v_j) in E, an oligonucleotide $O_{(vi,vj)}$ was created which was the $3'$ 10 mer of O_i (unless $i = 0$, in which case it was all of O_i) followed by the $5'$ mer of O_j (unless $j = 6$, in which case it was all of O_j). The 20-mer sequence Watson–Crick complementary to O_i was denoted O_i^1. For each vertex i in V (except $i = 0$ and $i = 6$) and for each edge (v_i, v_j) in E, large quantities of oligonucleotides O_i and $O_{(vi,vj)}$ were mixed together in a single ligation reaction. Here the oligonucleotides O_i^1 served as *splints* to bring oligonucleotides associated with compatible edges together for ligation. Consequently, the ligation reaction resulted in the formation of DNA molecules that can be viewed as encoding random paths through the directed graph G. From the random paths generated, basic biological operations are applied to remove illegal paths and select a Hamiltonian path [2].

C. The Sticker-Based Model

The sticker-based model employs two basic groups of single-stranded DNA molecules in its representation of a bit string [14]. Consider a *memory strand* N bases in length subdivided into K nonoverlapping regions each M bases long (thus, $N \geq M * K$). Each region is identified with exactly one bit position (or equivalently one Boolean variable) during the course of the computation. Adleman *et al.* [14] also designed K different *sticker strands* or *simply stickers*. Each sticker is M bases long and is complementary to one and only one of

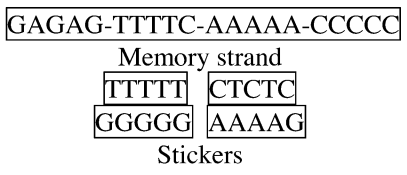

Fig. 2. An example of a sticker memory.

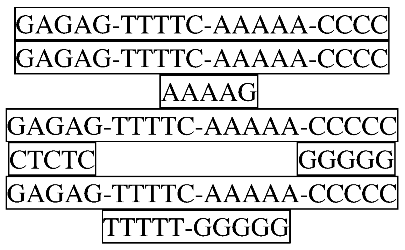

Fig. 3. Examples of memory complexes.

the K memory regions. If a sticker is annealed to its matching region on a given memory strand, then the bit corresponding that particular region is on for that strand. If no sticker is annealed to a region, then that region's bit is off. Each memory strand along with its annealed stickers (if any) represents one bit string. Such partial duplexes are called *memory complexes*. A large set of bit strings is represented by a large number of identical memory strands each of which has stickers annealed only at the required bit positions. Such a collection of memory complexes is called as a tube.

In this model, a unique association of memory strands and stickers represents each possible bit string. In the illustration given in Fig. 2, we consider a memory strand of length $n = 20$, divided into $k = 4$ regions, each of length $m = 5$. Thus, in this case the necessary complexes are interpreted as containing four bits of information. In particular, consider the memory complexes depicted in Fig. 3. In the first memory complex, all regions are off, whereas in the last complex the last two regions are on. The binary numbers represented by these four memory complexes are 0000, 0100, 1001, and 0011, respectively.

D. Adleman's Experiment for Solution of a Satisfability Problem

Adleman *et al.* [22], [46] performed experiments that were applied to, respectively, solve a six-variable 11-clause formula and a 20-variable 24-clause three-conjunctive normal form (3-CNF) formula. A Lipton encoding [3] was used to represent all possible variable assignments for the chosen six-variable or 20-variable SAT problem. For each of the six variables $x_1, \ldots, x_6$, two distinct 15 base value sequences were designed. One represents *true* (T), x_k^T, and another represents *false* (F), x_k^F for $1 \le k \le 6$. Each of the 2^6 truth assignments was represented by a *library sequence* of 90 bases consisting of the concatenation of one value sequence for each variable. DNA molecules with library sequences are termed *library strands* and a combinatorial pool containing library strands is termed a *library*. The six-variable library strands were synthesized by employing a mix-and-split combinatorial synthesis technique [24]. The library strands were assigned library sequences with x_1 at the $5'$-end and x^6 at the $3'$-end $(5' - x_1 - x_2 - x_3 - x_4 - x_5 - x_6 - 3')$. Thus synthesis began by assembling the two 15 base oligonucleotides with sequences x_6^T and x_6^F. This process was repeated until all 6 variables had been treated.

The probes used for separating the library strands have sequences complementary to the value sequences. Errors in the separation of the library strands are errors in the computation. Sequences must be designed to ensure that library strands have little secondary structure that might inhibit intended probe-library hybridization. The design must also exclude sequences that might encourage unintended probe-library hybridization. To help achieve these goals, sequences were computer-generated to satisfy the proposed seven constraints [22]. The similar method also is applied to solve a 20-variable of 3-SAT [46].

E. DNA Manipulations

In the past decade, there have been revolutionary advances in the field of biomedical engineering, particularly in recombinant DNA and RNA manipulating. Due to the industrialization of the biotechnology field, laboratory techniques for recombinant DNA and RNA manipulation are becoming highly standardized. Basic principles about recombinant DNA can be found in [47]–[50]. In this subsection we describe eight biological operations that are useful for solving the problem of factoring integers. The method of constructing DNA solution space for the problem of factoring integers is based on the proposed method in [22], [46].

A (test) tube is a set of molecules of DNA (a multiset of finite strings over the alphabet $\{A, C, G, T\}$). Given a tube, one can perform the following operations.

1. *Extract*. Given a tube P and a short single strand of DNA, S, the operation produces two tubes $+(P, S)$ and $-(P, S)$, where $+(P, S)$ is all of the molecules of DNA in P which contain S as a substrand and $-(P, S)$ is all of the molecules of DNA in P which do not contain S.
2. *Merge*. Given tubes P_1 and P_2, yield $\cup(P_1, P_2)$, where $\cup(P_1, P_2) = P_1 \cup P_2$. This operation is to pour two tubes into one, without any change in the individual strands.
3. *Detect*. Given a tube P, if P includes at least one DNA molecule, we have "yes," and if P contains no DNA molecule, we have "no."
4. *Discard*. Given a tube P, the operation will discard P.
5. *Amplify*. Given a tube P, the operation Amplify(P, P_1, P_2), will produce two new tubes P_1 and P_2 so that P_1 and P_2 are totally a copy of P (P_1 and P_2 are now identical) and P becomes an empty tube.
6. *Append*. Given a tube P containing a short strand of DNA Z, the operation will append Z onto the end of every strand in P.
7. *Append-head*. Given a tube P containing a short strand of DNA, Z, the operation will append Z onto the head of every strand in P.
8. *Read*. Given a tube P, the operation is used to describe a single molecule, which is contained in tube P. Even if P contains many different molecules each encoding a different set of bases, the operation can give an explicit description of exactly one of them.

F. Comparisons of Various Famous DNA Models

Based on solution space of *splint* in the Adleman–Lipton model, their methods [7], [17]–[20], [35] could be applied toward solving the traveling salesman problem, the dominating-set problem, the vertex cover problem, the clique problem, the independent-set problem, the three-dimensional matching problem, the set-packing problem, the set-cover problem, and the problem of exact cover by three-sets. Lipton *et al.* [51] indicated that DNA-based computing had been shown to easily be capable of breaking the data encryption standard from solution space of *splint*. The methods used for solving problems have exponentially increased volumes of DNA and linearly increased the time.

Bach *et al.* [33] proposed a $n1.89^n$ volume, $O(n^2 + m^2)$ time molecular algorithm for the three-coloring problem and a 1.51^n volume, $O(n^2m^2)$ time molecular algorithm for the independent set problem, where n and m are, subsequently, the number of vertices and the number of edges in the problems resolved. Fu [21] presented a polynomial-time algorithm with a 1.497^n volume for the three-SAT problem, a polynomial-time algorithm with a 1.345^n volume for the three-coloring problem, and a polynomial-time algorithm with a 1.229^n volume for the independent set. Though the size of those volumes [21], [33] is lower, constructing those volumes is more difficult and the time complexity is higher.

Quyang *et al.* [4] showed that enzymes could be used to solve the NP-complete clique problem. Because the maximum number of vertices that they can process is limited to 27, the maximum number of DNA strands for solving this problem is 2^{27} [4]. Shin *et al.* [8] presented an encoding scheme for decreasing the error rate of hybridization. This method [8] can be employed toward ascertaining the traveling salesman problem for representing integers and real values with fixed-length codes. Arita *et al.* [5] and Morimoto *et al.* [6] proposed a new molecular experimental technique and a solid-phase method to find a Hamiltonian path. Amos [13] proposed a parallel filtering model for resolving the Hamiltonian path problem, the subgraph isomorphism problem, the three-vertex-colorability problem, the clique problem, and the independent-set problem. The methods in [5], [6], and [13] have lowered the error rate in real molecular experiments. In [26], [27], and [30], the methods for DNA-based computing by self-assembly require the use of DNA nanostructures, called tiles, to own expressive computational power and convenient input and output (I/O) mechanisms. That is, DNA tiles have lower error rate in self-assembly.

One of the earliest attempts to perform arithmetic operations (addition of two positive binary numbers) using DNA is by Guarneiri *et al.* [38], utilizing the idea of encoding differently bit values zero and one as single-stranded DNAs, based upon their positions and the operands in which they appear. Gupta *et al.* [39] performed logic and arithmetic operations using the fixed bit encoding of the full corresponding truth tables. Qiu and Lu [40] applied substitution operation to insert results (by encoding all possible outputs of bit by bit operation along with second operand) in the operand strands. Ogihara and Ray [41], as well as Amos and Dunne [42] proposed methods to realize any Boolean circuit (with bounded fan in) using DNA strands in a constructive fashion. Other new suggestions to perform all basic arithmetic operations are by Atanasiu [43] using P systems and by Frisco [44] using splicing operation under general H systems, and by Hubert and Schuler [45]. Barua *et al.* [31] proposed a recursive DNA algorithm for adding two binary numbers, which require $O(\log n)$ biosteps using only $O(n)$ different type of DNA strands, where n is the size of the binary string representing the larger of the two numbers.

Adleman *et al.* [14] proposed a sticker-based model to enhance the error rate of hybridization in the Adleman–Lipton model. Their model can be used for determining solutions of an instance in the set cover problem. Simultaneously, Adleman *et al.* [52] also pointed out that the data encryption standard could be easily broken from solution space of *stickers* in the sticker-based model. Perez-Jimenez *et al.* [15] employed the sticker-based model [14] to resolve knapsack problems. In our previous work, Chang *et al.* [25], [32], [36], [53] also employed the sticker-based model and the Adleman–Lipton model for dealing with Cook's theorem [9], [10], the set-splitting problem, the subset-sum problem, and the dominating-set problem for decreasing the error rate of hybridization.

III. Factoring the Product of Two Large Prime Numbers of K Bits

A. RSA Public-Key Cryptosystem

In the RSA cryptosystem [34], a participant creates his public and secret keys with the following steps. The first step is to select at random two large prime numbers p and q, assuming that the length of p and q are both k bits. The second step is to compute n by the equation $n = p * q$. The third step is to select a small odd integer e that is relatively prime to $\varnothing(n)$, which is equal to $(p-1)*(q-1)$. The fourth step is to compute d as the multiplicative inverse of e, module $\varnothing(n)$. The fifth step is to publish the pair $P = (e, n)$ as his RSA public key. The sixth step is to keep secret the pair $S = (d, n)$ as his secret key. A method to factor n as $p * q$ in a reasonable amount of time has not been found.

B. Solution Space of DNA Strands for Every Unsigned Integer of k Bits

Suppose that an unsigned integer of k bits M is represented as a k-bit binary number, $m_k \ldots m_1$, where the value of each bit m_j is either one or zero for $1 \leq j \leq k$. The bits m_k and m_1 represent, respectively, the most significant bit and the least significant bit for M. The range of the value to an unsigned integer of k bits is from 0 to $2^k - 1$. From [22], [46], for every bit m_j, two *distinct* 15 base value sequences are designed. One represents the value zero for m_j and the other represents the value one for m_j. For convenience, we assume that m_j^1 denotes the value of m_j to be one and m_j^0 defines the value of m_j to be zero. The following algorithm is used to construct the solution space of DNA strands for 2^k different unsigned integer values.

```
Procedure InitialSolution($T_0$)
(1) For $j = k$ down to 1
    (1a) Amplify($T_0$, $T_1$, $T_2$).
    (1b) Append($T_1$, $m_j^1$).
    (1c) Append($T_2$, $m_j^0$).
    (1d) $T_0 = \cup(T_1, T_2)$.
  EndFor
EndProcedure
```

TABLE I
RESULT FOR TUBE T_0 IS GENERATED BY THE ALGORITHM INITIALSOLUTION(T_0)

Tub e	The result is generated by InitialSolution(T_0)
T_0	$\{{m_3}^1{m_2}^1{m_1}^1, {m_3}^1{m_2}^1{m_1}^0, {m_3}^1{m_2}^0{m_1}^1, {m_3}^1{m_2}^0{m_1}^0, {m_3}^0{m_2}^1{m_1}^1, {m_3}^0{m_2}^1{m_1}^0, {m_3}^0{m_2}^0{m_1}^1, {m_3}^0{m_2}^0{m_1}^0\}$

Consider that the number of bits for M is 3 bits. Eight values for M are, respectively, 000, 001, 010, 011,100, 101 110, and 111. Tube T_0 is an empty tube and is regarded as an input tube for the algorithm InitialSolution(T_0). Because the value for k is three, Steps (1a) through (1d) will be run three times. After the first execution of Step (1a) is finished, tube $T_0 = \phi$, tube $T_1 = \phi$, and tube $T_2 = \phi$. Next, after the first execution for Step (1b) and Step (1c) is performed, tube $T_1 = \{m_3^1\}$ and tube $T_2 = \{m_3^0\}$. After the first execution of Step (1d) is run, tube $T_0 = \{m_3^1, m_3^0\}$, tube $T_1 = \phi$, and tube $T_2 = \phi$. Then, after the second execution of Step (1a) is finished, tube $T_0 = \phi$, tube $T_1 = \{m_3^1, m_3^0\}$, and tube $T_2 = \{m_3^1, m_3^0\}$. After the rest of operations are performed, tube $T_1 = \phi$, tube $T_2 = \phi$, and the result for tube T_0 is shown in Table I. Lemma 1 is applied to demonstrate correction of the algorithm InitialSolution(T_0).

Lemma 1: The algorithm InitialSolution(T_0) is used to construct the solution space of DNA strands for 2^k different unsigned integer values.

Proof: The algorithm InitialSolution(T_0) is implemented by means of the *amplify, append*, and *merge* operations. Each execution of Step (1a) is used to amplify tube T_0 and to generate two new tubes, T_1 and T_2, which are copies of T_0. Tube T_0 then becomes empty. Then, Step (1b) is applied to append a DNA sequence, representing the value one for m_j, onto the end of every strand in tube T_1. This is to say that those integers containing the value one to the jth bit appear in tube T_1. Step (1c) is also employed to append a DNA sequence, representing the value zero for m_j, onto the end of every strand in tube T_2. That implies that these integers containing the value zero to the jth bit appear in tube T_2. Next, Step (1d) is used to pour tubes T_1 and T_2 into tube T_0. This indicates that DNA strands in tube T_0 include DNA sequences of $m_j = 1$ and $m_j = 0$. At the end of Step (1), tube T_0 consists of 2^k DNA sequences representing 2^k different unsigned integer values.

From InitialSolution(T_0), it takes k *amplify* operations, $2 * k$ *append* operations, k *merge* operations, and three test tubes to construct the solution space of DNA strands. A value sequence for every bit contains 15 bases. Therefore, the length of a DNA strand, encoding an unsigned integer value of k bits, is $15 * k$ bases consisting of the concatenation of one value sequence for each bit.

C. The Construction to the Product of Two Large Prime Numbers of k Bits

Assume that the length for n, the product of two large prime numbers of k bits, denoted in Section III-A, is $(2 * k)$ bits. Also suppose that the product n is used to represent the minuend (dividend) and the difference for successive compare, shift, and subtract operations in a divider. When n is divided by M, an unsigned integer of k bits denoted in Section III-B, M is one of two large prime numbers if the remainder is equal to zero. Assume that in a divider the length of a dividend is $(2 * k)$ bits and the length of a divisor is d bits, where $1 \leq d \leq k$. It is very obvious that the division instruction is finished through successive compare, shift, and subtract operations of at most $(2 * k)$ times. Therefore, suppose that n is represented as a $(2 * k)$-bit binary number, $n_{o,(2*k)} \ldots n_{o,1}$, where the value of each bit $n_{o,q}$ is either one or zero for $1 \leq o \leq (2 * k + 1)$ and $1 \leq q \leq (2 * k)$. The bits $n_{o,(2*k)}$ and $n_{o,1}$, respectively, represent the most significant bit and the least significant bit for n. One binary number $n_{o,(2*k)} \ldots n_{o,1}$ and another binary number $n_{o+1,(2*k)} \ldots n_{o+1,1}$ are, respectively, applied to represent the minuend and the difference for the successive compare, shift, and subtract operations of the oth time. This is to say that the binary number $n_{o+1,(2*k)} \ldots n_{o+1,1}$ is the minuend for the successive compare, shift, and subtract operations of the $(o + 1)$th time.

TABLE II
RESULT FOR TUBE T_0 IS GENERATED BY THE ALGORITHM INITIALPRODUCT(T_0)

Tub e	The result is generated by InitialProduct(T_0)
T_0	$\{{n_{1,6}}^0{n_{1,5}}^0{n_{1,4}}^1{n_{1,3}}^1{n_{1,2}}^1{n_{1,1}}^1{m_3}^1{m_2}^1{m_1}^1, {n_{1,6}}^0{n_{1,5}}^0{n_{1,4}}^1{n_{1,3}}^1{n_{1,2}}^1{n_{1,1}}^1{m_3}^1{m_2}^1{m_1}^0, {n_{1,6}}^0{n_{1,5}}^0{n_{1,4}}^1{n_{1,3}}^1{n_{1,2}}^1{n_{1,1}}^1{m_3}^1{m_2}^0{m_1}^1, {n_{1,6}}^0{n_{1,5}}^0{n_{1,4}}^1{n_{1,3}}^1{n_{1,2}}^1{n_{1,1}}^1{m_3}^1{m_2}^0{m_1}^0, {n_{1,6}}^0{n_{1,5}}^0{n_{1,4}}^1{n_{1,3}}^1{n_{1,2}}^1{n_{1,1}}^1{m_3}^0{m_2}^1{m_1}^1, {n_{1,6}}^0{n_{1,5}}^0{n_{1,4}}^1{n_{1,3}}^1{n_{1,2}}^1{n_{1,1}}^1{m_3}^0{m_2}^1{m_1}^0, {n_{1,6}}^0{n_{1,5}}^0{n_{1,4}}^1{n_{1,3}}^1{n_{1,2}}^1{n_{1,1}}^1{m_3}^0{m_2}^0{m_1}^1, {n_{1,6}}^0{n_{1,5}}^0{n_{1,4}}^1{n_{1,3}}^1{n_{1,2}}^1{n_{1,1}}^1{m_3}^0{m_2}^0{m_1}^0\}$

For every bit $n_{o,q}$, two *distinct* 15 base value sequences were designed. One represents the value zero for $n_{o,q}$ and the other represents the value one for $n_{o,q}$. For convenience, we assume that $n_{o,q}^1$ denotes the value of $n_{o,q}$ to be one and $n_{o,q}^0$ defines the value of $n_{o,q}$ to be zero. The following algorithm is used to construct a DNA strand for the value of n.

```
Procedure InitialProduct($T_0$)
(1) For $q = 1$ to $2 * k$
    (1a) Append-head($T_0, n_1, q$).
  EndFor
EndProcedure
```

Consider that the number of bits for n is 6 bits and the value for n is 001 111. Tube T_0 with the result shown in Table I is regarded as an input tube for the algorithm, InitialProduct(T_0). Because the value for $2 * k$ is six, Step (1a) will be executed six times. After each operation for Step (1a) is performed, the result is shown in Table II. Lemma 2 is used to prove correction of the algorithm InitialProduct(T_0).

Lemma 2: A DNA strand for the value of n can be constructed from InitialProduct(T_0).

Proof: Refer to Lemma 1.

From InitialProduct(T_0), it takes $(2 * k)$ *append-head* operations and one test tube to construct a DNA strand. The length of the DNA strand, encoding the value of n, is $30 * k$ bases consisting of the concatenation of one value sequence for each bit.

TABLE III
RESULT IS GENERATED BY ONEBITCOMPARATOR$(T_0^{>}, T_0^{=}, T_0^{<}, d, o, j)$

Tube	The result is generated by OneBitComparator($T_0^>$, $T_0^=$, $T_0^<$, d, o, j)
$T_0^>$ and $T_0^=$	ϕ
$T_0^<$	$\{b_{1,0}{}^0 n_{1,6}{}^0 n_{1,5}{}^0 n_{1,4}{}^1 n_{1,3}{}^1 n_{1,2}{}^1 n_{1,1}{}^1 m_3{}^1 m_2{}^1 m_1{}^1, b_{1,0}{}^0 n_{1,6}{}^0 n_{1,5}{}^0 n_{1,4}{}^1 n_{1,3}{}^1 n_{1,2}{}^1 n_{1,1}{}^1 m_3{}^1 m_2{}^1 m_1{}^0, b_{1,0}{}^0 n_{1,6}{}^0 n_{1,5}{}^0 n_{1,4}{}^1 n_{1,3}{}^1 n_{1,2}{}^1 n_{1,1}{}^1 m_3{}^1 m_2{}^0 m_1{}^1, b_{1,0}{}^0 n_{1,6}{}^0 n_{1,5}{}^0 n_{1,4}{}^1 n_{1,3}{}^1 n_{1,2}{}^1 n_{1,1}{}^1 m_3{}^1 m_2{}^0 m_1{}^0\}$

D. The Construction of a Parallel Comparator

A division operation for a dividend of $(2*k)$ bits and a divisor of d bits for $1 \le d \le k$ are carried out by successive compare, shift, and subtract operations of at most $(2*k+1)$ times. This indicates that compare and shift operations must be finished before the corresponding subtraction operation is done. Therefore, the algorithm OneBitComparator$(T_0^{>}, T_0^{=}, T_0^{<}, d, o, j)$ is presented to perform the function of a 1-bit parallel comparator and the algorithm ParallelComparator$(T_0, T_0^{>}, T_0^{=}, T_0^{<}, d, o)$ also is proposed to perform the function of a k-bit parallel comparator.

```
Procedure OneBitComparator($T_0^{>}, T_0^{=}, T_0^{<}, d, o, j$)
(1) $T_1 = +(T_0^{=}, n^1_{o,(2*k)-(o-1)-(j-o)})$ and $T_2 = -(T_0^{=}, n^1_{o,(2*k)-(o-1)-(j-o)})$.
(2) $T_3 = +(T_1, m^1_{(k-d+1)+o-j})$ and $T_4 = -(T_1, m^1_{(k-d+1)+o-j})$.
(3) $T_5 = +(T_2, m^1_{(k-d+1)+o-j})$ and $T_6 = -(T_2, m^1_{(k-d+1)+o-j})$.
(4) $T_0^{=} = \cup(T_0^{=}, T_3, T_6)$.
(5) $T_0^{>} = \cup(T_0^{>}, T_4)$.
(6) $T_0^{<} = \cup(T_0^{<}, T_5)$.
EndProcedure
```

Consider that the first execution for the algorithm OneBitComparator$(T_0^{>}, T_0^{=}, T_0^{<}, d, o, j)$ is invoked. The values for d, o and j are, respectively, one, one, and one. Tube $T_0^{>} = \phi$, tube

$$T_0^{=} = \{b^0_{1,0}n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3m^1_2m^1_1, b^0_{1,0}n^0_{1,6}n^0_{1,5}n^1_{1,4} n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3m^1_2m^0_1, b^0_{1,0}n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3 m^0_2m^1_1, b^0_{1,0}n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3m^0_2m^0_1\},$$

tube $T_0^{<} = \phi$, and three tubes are regarded as input tubes. After each operation in the algorithm is performed, the result is shown in Table III. Lemma 3 is used to show correction of the algorithm OneBitComparator$(T_0^{>}, T_0^{=}, T_0^{<}, d, o, j)$.

Lemma 3: The algorithm OneBitComparator$(T_0^{>}, T_0^{=}, T_0^{<}, d, o, j)$ can be applied to perform the function of a 1-bit parallel comparator.

Proof: The algorithm OneBitComparator$(T_0^{>}, T_0^{=}, T_0^{<}, d, o, j)$ is implemented by the *extract* and *merge* operations. The execution of Step (1) employs the *extract* operation to form two test tubes: T_1 and T_2. The first tube T_1 includes all of the strands that have $n_{o,(2*k)-(o-1)-(j-o)} = 1$. The second tube T_2 consists of all of the strands that have $n_{o,(2*k)-(o-1)-(j-o)} = 0$. Next, on the execution of Step (2), it also uses the *extract* operation to form two test tubes: T_3 and T_4. The first tube T_3 includes all of the strands that have $n_{o,(2*k)-(o-1)-(j-o)} = 1$ and $m_{(k-d+1)+o-j} = 1$. The second tube T_4 consists of all of the strands that have $n_{o,(2*k)-(o-1)-(j-o)} = 1$ and $m_{(k-d+1)+o-j} = 0$. The execution of Step (3) uses the extract operation to form two test tubes: T_5 and T_6. The first tube T_5 includes all of the strands that have $n_{o,(2*k)-(o-1)-(j-o)} = 0$ and $m_{(k-d+1)+o-j} = 1$. The second tube T_6 consists of all of the strands that have $n_{o,(2*k)-(o-1)-(j-o)} = 0$ and $m_{(k-d+1)+o-j} = 0$. Because the corresponding bits of the dividend and the divisor in T_3 are both one and the corresponding bits of the dividend and the divisor in T_6 are both zero, next, the execution of Step (4) uses the *merge* operations to pour T_3 and T_6 into $T_0^{=}$. In T_4, the corresponding bit of the dividend is one and the corresponding bit of the divisor is zero, so the execution of Step (5) also applies the *merge* operations to pour T_4 into $T_0^{>}$. Next, in T_5, since the corresponding bit of the dividend is zero and the corresponding bit of the divisor is one, the execution of Step (6) employs the *merge* operations to pour T_5 into $T_0^{<}$.

From OneBitComparator$(T_0^{>}, T_0^{=}, T_0^{<}, d, o, j)$, it takes three *extract* operations, three *merge* operations, and nine test tubes to finish the function of a 1-bit parallel comparator.

```
Procedure ParallelComparator($T_0, T_0^{>}, T_0^{=}, T_0^{<}, d, o$)
(1) For $j = 1$ to $o - 1$
    (1a) $T_7 = +(T_0, n^1_{o,(2*k)-(j-1)})$ and $T_8 = -(T_0, n^1_{o,(2*k)-(j-1)})$.
    (1b) $T_0^{>} = \cup(T_0^{>}, T_7)$.
    (1c) If (Detect($T_8$)= "yes") then
        (1d) $T_0 = \cup(T_0, T_8)$.
      Else
        (1e) Terminate the algorithm.
      EndIf
    EndFor
(2) $T_0^{=} = \cup(T_0^{=}, T_0)$.
(3) For $j = o$ to $k + o - d$
(3a) OneBitComparator($T_0^{>}, T_0^{=}, T_0^{<}, d, o, j$).
(3b) If (Detect($T_0^{=}$) = "no") then
      (3c) Terminate the algorithm.
    EndIf
  EndFor
EndProcedure
```

Consider that the first execution for the algorithm ParallelComparator$(T_0, T_0^{>}, T_0^{=}, T_0^{<}, d, o)$ is invoked. The values for d and o are, respectively, one and one. Tube

$$T_0 = \{b^0_{1,0}n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3m^1_2m^1_1, b^0_{1,0}n^0_{1,6}n^0_{1,5}n^1_{1,4} n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3m^1_2m^0_1, b^0_{1,0}n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3 m^0_2m^1_1, b^0_{1,0}n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3m^0_2m^0_1\},$$

tube $T_0^{>} = \phi$, tube $T_0^{=} = \phi$, tube $T_0 < = \phi$, and four tubes are regarded as input tubes. Because the value for the upper bound in Step (1) is zero, Steps (1a) through (1e) are not run. After the first execution for Step (2), Step (3a) and Step (3b) is finished, a "no" is returned from Step (3b). Therefore, the algorithm is terminated from Step (3c). The result is shown in Table IV. Lemma 4 is used to show correction of the algorithm ParallelComparator$(T_0, T_0^{>}, T_0^{=}, T_0^{<}, d, o)$.

Lemma 4: The algorithm ParallelComparator$(T_0, T_0^{>}, T_0^{=}, T_0^{<}, d, o)$ can be used to finish the function of a k-bit parallel comparator.

TABLE IV
RESULT IS GENERATED BY PARALLELCOMPARATOR$(T_0, T_0^{>}, T_0^{=}, T_0^{<}, d, o)$

Tube	The result is generated by ParallelComparator(T_0, $T_0^{>}$, $T_0^{=}$, $T_0^{<}$, d, o)
T_0	ϕ
$T_0^{>}$ and $T_0^{=}$	ϕ
$T_0^{<}$	$\{b_{1,0}^{0}n_{1,6}^{0}n_{1,5}^{0}n_{1,4}^{1}n_{1,3}^{1}n_{1,2}^{1}n_{1,1}^{1}m_3^{1}m_2^{1}m_1^{1}, b_{1,0}^{0}n_{1,6}^{0}n_{1,5}^{0}n_{1,4}^{1}n_{1,3}^{1}n_{1,2}^{1}n_{1,1}^{1}m_3^{1}m_2^{1}m_1^{0}, b_{1,0}^{0}n_{1,6}^{0}n_{1,5}^{0}n_{1,4}^{1}n_{1,3}^{1}n_{1,2}^{1}n_{1,1}^{1}m_3^{1}m_2^{0}m_1^{1}, b_{1,0}^{0}n_{1,6}^{0}n_{1,5}^{0}n_{1,4}^{1}n_{1,3}^{1}n_{1,2}^{1}n_{1,1}^{1}m_3^{1}m_2^{0}m_1^{0}\}$

Proof: Step (1) is the first loop and is used to compare the most significant $(o - 1)$ bits of the dividend with $(o - 1)$ zeros for the oth compare and shift operations. The first execution of Step (1a) employs the *extract* operation to form two test tubes: T_7 and T_8. The first tube T_7 includes all of the strands that have $n_{o,(2*k)-(j-1)} = 1$. The second tube T_8 consists of all of the strands that have $n_{o,(2*k)-(j-1)} = 0$. In T_7, the corresponding bit of the dividend is one and the shift bit of the divisor is zero, so the first execution of Step (1b) uses the *merge* operations to pour T_7 into $T_0^{>}$. The first execution of Step (1c) employs the *detect* operations to check whether tube T_8 contains any DNA strand or not. If a "yes" is returned, then the first execution of Step (1d) applies the *merge* operations to pour T_8 into T_0. Otherwise, the algorithm is terminated in Step (1e). Repeat the execution of each step in the loop until the number of the execution for the loop is performed.

After each operation in the first loop is finished, tube T_0 contains the strands that have the comparative result ("=") for the most significant $(o-1)$ bits of the dividend with $(o-1)$ zeros for the oth compare and shift operations. Step (2) uses the *merge* operation to pour T_0 into $T_0^{=}$. When the first execution of Step (3a) calls the algorithm OneBitComparator$(T_0^{>}, T_0^{=}, T_0^{<}, d, o, j)$ to finish the comparative result of the corresponding bit for the $(2 * k)$-bit dividend and the d-bit divisor for $1 \le d \le k$ in a divider. After Step (3a) is performed, the comparative results are, respectively, represented in $T_0^{>}$, $T_0^{=}$, and $T_0^{<}$. On the first execution of Step (3b), it uses the detect operations to check whether there is any DNA sequence in $T_0^{=}$. If a "no" is returned, then the execution of Step (3c) is used to terminate the algorithm. Otherwise, Steps (3a) through (3b) are repeated to execute until the corresponding bits of the $(2 * k)$-bit dividend and the d-bit divisor for $1 \le d \le k$ in a divider are all processed. Finally, tube $T_0^{>}$ contains the strands with the comparative result of greater than (">"), tube $T_0^{=}$ includes the strands with the comparative result of equal ("=") and tube $T_0^{<}$ consists of the strands with the comparative result of less than ("<").

From ParallelComparator$(T_0, T_0^{>}, T_0^{=}, T_0^{<}, d, o)$, it takes $(3 * k - 3 * d + o + 2)$ *extract* operations, $(3 * k - 3 * d + 2 * o + 2)$ *merge* operations, $(k - d + o)$ *detect* operations, and 11 tubes to finish the function of a k-bit parallel comparator.

E. The Construction of a Parallel 1-Bit Subtractor

A 1-bit subtractor is a function that forms the arithmetic subtraction of three input bits. It consists of three inputs and two outputs. Two of the input bits, respectively, represent minuend and subtrahend bits to be subtracted. The third input represents the borrow bit from the previous higher significant position. The first output gives the value of the difference for minuend and subtrahend bits to be subtracted. The second output gives the value of the borrow bit to minuend and subtrahend bits to be subtracted. The truth table of the 1-bit subtractor is as follows.

Suppose that a 1-bit binary number $n_{o,q}$ denoted in Section III-C is used to represent the first input of a 1-bit subtractor for $1 \le o \le (2*k+1)$ and $1 \le q \le (2*k)$. Also assume that a 1-bit binary number $n_{o+1,q}$ denoted in Section III-C is applied to represent the first output of a 1-bit subtractor. Suppose that a 1-bit binary number m_j denoted in Section III-B is also employed to represent the second input of a 1-bit subtractor for $1 \le j \le k$. Also assume that a 1-bit binary number $b_{o,q}$ is employed to represent the second output of a 1-bit subtractor. Also suppose that a 1-bit binary number $b_{o,q-1}$ is employed to represent the third input of a 1-bit subtractor.

For every bit $b_{o,q-1}$ and $b_{o,q}$ to $1 \le o \le (2 * k + 1)$ and $1 \le q \le (2 * k)$, two *distinct* DNA sequences are designed to represent the value zero or one of every corresponding bit. For convenience, we assume that $b_{o,q}^{1}$ contains the value of $b_{o,q}$ to be one and $b_{o,q}^{0}$ contains the value of $b_{o,q}$ to be zero. Also suppose that $n_{o+1,q}^{1}$ denotes the value of $n_{o+1,q}$ to be one and $n_{o+1,q}^{0}$ defines the value of $n_{o+1,q}$ to be zero. Similarly, assume that $b_{o,q-1}^{1}$ contains the value of $b_{o,q-1}$ to be one and $b_{o,q-1}^{0}$ contains the value of $b_{o,q-1}$ to be zero. The following algorithm is proposed to finish the function of a parallel 1-bit subtractor.

```
Procedure ParallelOneBitSubtractor($T_0^{>=}, o, q, j$)
 (1) $T_1 = +(T_0^{>=}, n_{o,q}^{1})$ and $T_2 = -(T_0^{>=}, n_{o,q}^{1})$.
 (2) $T_3 = +(T_1, m_j^{1})$ and $T_4 = -(T_1, m_j^{1})$.
 (3) $T_5 = +(T_2, m_j^{1})$ and $T_6 = -(T_2, m_j^{1})$.
 (4) $T_7 = +(T_3, b_{o,q-1}^{1})$ and $T_8 = -(T_3, b_{o,q-1}^{1})$.
 (5) $T_9 = +(T_4, b_{o,q-1}^{1})$ and $T_{10} = -(T_4, b_{o,q-1}^{1})$.
 (6) $T_{11} = +(T_5, b_{o,q-1}^{1})$ and $T_{12} = -(T_5, b_{o,q-1}^{1})$.
 (7) $T_{13} = +(T_6, b_{o,q-1}^{1})$ and $T_{14} = -(T_6, b_{o,q-1}^{1})$.
 (8a) If (Detect($T_7$) = "yes") then
 (8) Append-head($T_7, n_{o+1,q}^{1}$) and
Append-head($T_7, b_{o,q}^{1}$).
   EndIf
 (9a) If (Detect($T_8$) = "yes") then
   (9) Append-head($T_8, n_{o+1,q}^{0}$) and
Append-head($T_8, b_{o,q}^{o}$).
   EndIf
 (10a) If (Detect($T_9$) = "yes") then
     (10) Append-head($T_9, n_{o+1,q}^{0}$) and
Append-head($T_9, b_{o,q}^{0}$).
   EndIf
 (11a) If (Detect($T_{10}$) = "yes") then
   (11) Append-head($T_{10}, n_{o+1,q}^{1}$) and
Append-head($T_{10}, b_{o,q}^{0}$).
   EndIf
 (12a) If (Detect($T_{11}$) = "yes") then
   (12) Append-head($T_{11}, n_{o+1,q}^{0}$) and
Append-head($T_{11}, b_{o,q}^{1}$).
   EndIf
 (13a) If (Detect($T_{12}$) = "yes") then
   (13) Append-head($T_{12}, n_{o+1,q}^{1}$) and
Append-head($T_{12}, b_{o,q}^{1}$).
   EndIf
```

```
  (14a) If (Detect($T_{13}$) = "yes") then
    (14) Append-head($T_{13}, n^1_{o+1,q}$) and
Append-head($T_{13}, b^1_{o,q}$).
    EndIf
  (15a) If (Detect($T_{14}$) = "yes") then
    (15) Append-head($T_{14}, n^0_{o+1,q}$) and
Append-head($T_{14}, b^0_{o,q}$).
    EndIf
  (16) $T_0^{>=} = \cup(T_7, T_8, T_9, T_{10}, T_{11}, T_{12}, T_{13}, T_{14})$.
  EndProcedure
```

Consider that the first execution for the algorithm ParallelOneBitSubtractor($T_0^{>=}, o, q, j$) invokes tube

$$\begin{aligned}T_0^{>=}=\{&b^0_{3,1}n^1_{4,1}b^0_{3,0}b^0_{2,6}n^0_{3,6}b^0_{2,5}n^0_{3,5}b^0_{2,4}n^1_{3,4}b^0_{2,3}n^1_{3,3}b^0_{2,2}n^1_{3,2}\\&b^0_{2,1}n^1_{3,1}b^0_{2,0}b^0_{1,6}n^0_{2,6}b^0_{1,5}n^0_{2,5}b^0_{1,4}n^1_{2,4}b^0_{1,3}n^1_{2,3}b^0_{1,2}n^1_{2,2}b^0_{1,1}\\&n^1_{2,1}b^0_{1,0}n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3m^1_2m^1_1, b^0_{3,1}n^1_{4,1}b^0_{3,0}\\&b^0_{2,6}n^0_{3,6}b^0_{2,5}n^0_{3,5}b^0_{2,4}n^1_{3,4}b^0_{2,3}n^1_{3,3}b^0_{2,2}n^1_{3,2}b^0_{2,1}n^1_{3,1}b^0_{2,0}b^0_{1,6}\\&n^0_{2,6}b^0_{1,5}n^0_{2,5}b^0_{1,4}n^1_{2,4}b^0_{1,3}n^1_{2,3}b^0_{1,2}n^1_{2,2}b^0_{1,1}n^1_{2,1}b^0_{1,0}n^0_{1,6}n^0_{1,5}\\&n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3m^1_2m^0_1, b^0_{3,1}n^1_{4,1}b^0_{3,0}b^0_{2,6}n^0_{3,6}b^0_{2,5}n^0_{3,5}\\&b^0_{2,4}n^1_{3,4}b^0_{2,3}n^1_{3,3}b^0_{2,2}n^1_{3,2}b^0_{2,1}n^1_{3,1}b^0_{2,0}b^0_{1,6}n^0_{2,6}b^0_{1,5}n^0_{2,5}b^0_{1,4}\\&n^1_{2,4}b^0_{1,3}n^1_{2,3}b^0_{1,2}n^1_{2,2}b^0_{1,1}n^1_{2,1}b^0_{1,0}n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}\\&n^1_{1,1}m^1_3m^0_2m^1_1, b^0_{3,1}n^1_{4,1}b^0_{3,0}b^0_{2,6}n^0_{3,6}b^0_{2,5}n^0_{3,5}b^0_{2,4}n^1_{3,4}b^0_{2,3}\\&n^1_{3,3}b^0_{2,2}n^1_{3,2}b^0_{2,1}n^1_{3,1}b^0_{2,0}b^0_{1,6}n^0_{2,6}b^0_{1,5}n^0_{2,5}b^0_{1,4}n^1_{2,4}b^0_{1,3}\\&n^1_{2,3}b^0_{1,2}n^1_{2,2}b^0_{1,1}n^1_{2,1}b^0_{1,0}n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3\\&m^0_2m^0_1\}\end{aligned}$$

and it is regarded as an input tube. The values for o, q, and j are, respectively, three, two, and one. After each operation in the algorithm is performed, the result is shown in Table VI. Lemma 5 is applied to show correction of the algorithm ParallelOneBitSubtractor($T_0^{>=}, o, q, j$).

Lemma 5: The algorithm ParallelOneBitSubtractor($T_0^{>=}, o, q, j$) can be applied to finish the function of a parallel 1-bit subtractor.

Proof: The algorithm ParallelOneBitSubtractor($T_0^{>=}, o, q, j$) is implemented by means of the *extract*, *append-head*, and *merge* operations. The execution of Step (1) employs the *extract* operation to form two test tubes: T_1 and T_2. The first tube T_1 includes all of the strands that have $n_{o,q} = 1$. The second tube T_2 consists of all of the strands that have $n_{o,q} = 0$. In Step (2), the *extract* operation is used to form two test tubes: T_3 and T_4. The first tube T_3 includes all of the strands that have $n_{o,q} = 1$ and $m_j = 1$. The second tube T_4 consists of all of the strands that have $n_{o,q} = 1$ and $m_j = 0$. Next, the execution of Step (3) uses the *extract* operation to form two test tubes: T_5 and T_6. The first tube T_5 includes all of the strands that have $n_{o,q} = 0$ and $m_j = 1$. The second tube T_6 consists of all of the strands that have $n_{o,q} = 0$ and $m_j = 0$. The execution of Step (4) uses the *extract* operation to form two test tubes: T_7 and T_8. The first tube T_7 includes all of the strands that have $n_{o,q} = 1$, $m_j = 1$ and $b_{o,q-1} = 1$. The second tube T_8 consists of all of the strands that have $n_{o,q} = 1$, $m_j = 1$ and $b_{o,q-1} = 0$. Then, on the execution of Step (5), it applies the *extract* operation to form two test tubes: T_9 and T_{10}. The first tube T_9 includes all of the strands that have $n_{o,q} = 1$, $m_j = 0$ and $b_{o,q-1} = 1$. The second tube T_{10} consists of all of the strands that have $n_{o,q} = 1$, $m_j = 0$ and $b_{o,q-1} = 0$. On the execution of Step (6), it employs the *extract* operation to form two test tubes: T_{11} and T_{12}. The first tube T_{11} includes all of the strands that have $n_{o,q} = 0$, $m_j = 1$ and $b_{o,q-1} = 1$. The second tube T_{12} consists of all of the strands that have $n_{o,q} = 0$, $m_j = 1$ and $b_{o,q-1} = 0$. Next, the execution of Step (7) uses the *extract* operation to form two test tubes: T_{13} and T_{14}. The first tube T_{13} includes all of the strands that have $n_{o,q} = 0$, $m_j = 0$ and $b_{o,q-1} = 1$. The second tube T_{14} consists of all of the strands that have $n_{o,q} = 0$, $m_j = 0$ and $b_{o,q-1} = 0$. After finishing Steps (1) to (7), eight different inputs of a 1-bit subtractor in Table V, respectively, have been poured into tubes T_7 through T_{14}.

TABLE V
TRUTH TABLE OF A 1-BIT SUBTRACTOR

Minuend bit	Subtrahend bit	Previous borrow bit	Difference bit	Borrow bit
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Steps (8a), (9a), (10a), (11a), (12a), (13a), (14a), and (15a) are, respectively, used to check whether contains any DNA strand for tubes T_7, T_8, T_9, T_{10}, T_{11}, T_{12}, T_{13}, and T_{14} or not. If any "yes" is returned for those steps, then the corresponding *append-head* operations will be run. Next, the execution of Step (8) uses the *append-head* operations to append $n^1_{o+1,q}$ and $b^1_{o,q}$ onto the head of every strand in T_7. On the execution of Step (9), it applies the *append-head* operations to append $n^0_{o+1,q}$ and $b^0_{o,q}$ onto the head of every strand in T_8. Then, the execution of Step (10) employs the *append-head* operations to append $n^0_{o+1,q}$ and $b^0_{o,q}$ onto the head of every strand in T_9. On the execution of Step (11), it uses the *append-head* operations to append $n^1_{o+1,q}$ and $b^0_{o,q}$ onto the head of every strand in T_{10}. Next, the execution of Step (12) uses the *append-head* operations to append $n^0_{o+1,q}$ and $b^1_{o,q}$ onto the head of every strand in T_{11}. On the execution of Step (13), it uses the *append-head* operations to append $n^1_{o+1,q}$ and $b^1_{o,q}$ onto the head of every strand in T_{12}. Then, the execution of Step (14) applies the *append-head* operations to append $n^1_{o+1,q}$ and $b^1_{o,q}$ onto the head of every strand in T_{13}. On the execution of Step (15), it employs the *append-head* operations to append $n^0_{o+1,q}$ and $b^0_{o,q}$ onto the head of every strand in T_{14}. After finishing Steps (8) to (15), eight different outputs of a 1-bit subtractor in Table V, respectively, are appended into tubes T_7 through T_{14}. Finally, the execution of Step (16) applies the *merge* operation to pour tubes T_7 through T_{14} into $T_0^{>=}$. Tube $T_0^{>=}$ contains the strands finishing the subtraction of a bit.

From ParallelOneBitSubtractor($T_0^{>=}, o, q, j$), it takes seven *extract* operations, 16 *append-head* operations, 16 *detect* operations, one *merge* operation, and 15 test tubes to compute the subtraction of a bit. Two output bits of a 1-bit subtractor encode the difference bit and the borrow bit to the subtraction of a bit. A value sequence for every output bit contains 15 bases. Therefore, the length of a DNA strand, encoding two output bits, is 30 base pairs.

F. The Construction of a Binary Parallel Subtractor

The 1-bit subtractor introduced in Section III-E figures out the difference bit and the borrow bit for two input bits and a previous borrow bit. A minuend of k bits and a subtrahend of d bits for $1 \leq d \leq k$ can finish subtractions of at most k times by means of this 1-bit subtractor. A binary parallel subtractor is a function that performs the arithmetic subtraction for a minuend of k bits and a subtrahend of d bits for $1 \leq d \leq k$. The following algorithm is proposed to finish the function of a binary parallel subtractor.

```
Procedure BinaryParallelSubtractor($T_0^{>=}, d, o, q$)
(1) For $j = 1$ to $k - d + 1$
    (1a) ParallelOneBitSubtractor($T_0^{>=}, o, 2 * k - (o - 1) - (k - d + 1 - j), j$).
  EndFor
EndProcedure
```

Consider that the first execution for the algorithm BinaryParallelSubtractor$(T_0^{>=}, d, o, q)$ invokes tube

$$\begin{aligned}T_0^{>=}=\{&b_{3,1}^0 n_{4,1}^1 b_{3,0}^0 b_{2,6}^0 n_{3,6}^0 b_{2,5}^0 n_{3,5}^0 b_{2,4}^0 n_{3,4}^1 b_{2,3}^0 n_{3,3}^1 b_{2,2}^0 n_{3,2}^1\\ &b_{2,1}^0 n_{3,1}^1 b_{2,0}^0 b_{1,6}^0 n_{2,6}^0 b_{1,5}^0 n_{2,5}^0 b_{1,4}^0 n_{2,4}^1 b_{1,3}^0 n_{2,3}^1 b_{1,2}^0 n_{2,2}^1\\ &b_{1,1}^0 n_{2,1}^1 b_{1,0}^0 n_{1,6}^0 n_{1,5}^0 n_{1,4}^1 n_{1,3}^1 n_{1,2}^1 n_{1,1}^1 m_3^1 m_2^1 m_1^1, b_{3,1}^0\\ &n_{4,1}^1 b_{3,0}^0 b_{2,6}^0 n_{3,6}^0 b_{2,5}^0 n_{3,5}^0 b_{2,4}^0 n_{3,4}^1 b_{2,3}^0 n_{3,3}^1 b_{2,2}^0 n_{3,2}^1 b_{2,1}^0\\ &n_{3,1}^1 b_{2,0}^0 b_{1,6}^0 n_{2,6}^0 b_{1,5}^0 n_{2,5}^0 b_{1,4}^0 n_{2,4}^1 b_{1,3}^0 n_{2,3}^1 b_{1,2}^0 n_{2,2}^1 b_{1,1}^0\\ &n_{2,1}^1 b_{1,0}^0 n_{1,6}^0 n_{1,5}^0 n_{1,4}^1 n_{1,3}^1 n_{1,2}^1 n_{1,1}^1 m_3^1 m_2^1 m_1^0, b_{3,1}^0 n_{4,1}^1\\ &b_{3,0}^0 b_{2,6}^0 n_{3,6}^0 b_{2,5}^0 n_{3,5}^0 b_{2,4}^0 n_{3,4}^1 b_{2,3}^0 n_{3,3}^1 b_{2,2}^0 n_{3,2}^1 b_{2,1}^0 n_{3,1}^1 b_{2,0}^0\\ &b_{1,6}^0 n_{2,6}^0 b_{1,5}^0 n_{2,5}^0 b_{1,4}^0 n_{2,4}^1 b_{1,3}^0 n_{2,3}^1 b_{1,2}^0 n_{2,2}^1 b_{1,1}^0 n_{2,1}^1 b_{1,0}^0 n_{1,6}^0\\ &n_{1,5}^0 n_{1,4}^1 n_{1,3}^1 n_{1,2}^1 n_{1,1}^1 m_3^1 m_2^0 m_1^1, b_{3,1}^0 n_{4,1}^1 b_{3,0}^0 b_{2,6}^0 n_{3,6}^0 b_{2,5}^0\\ &n_{3,5}^0 b_{2,4}^0 n_{3,4}^1 b_{2,3}^0 n_{3,3}^1 b_{2,2}^0 n_{3,2}^1 b_{2,1}^0 n_{3,1}^1 b_{2,0}^0 b_{1,6}^0 n_{2,6}^0 b_{1,5}^0 n_{2,5}^0\\ &b_{1,4}^0 n_{2,4}^1 b_{1,3}^0 n_{2,3}^1 b_{1,2}^0 n_{2,2}^1 b_{1,1}^0 n_{2,1}^1 b_{1,0}^0 n_{1,6}^0 n_{1,5}^0 n_{1,4}^1 n_{1,3}^1\\ &n_{1,2}^1 n_{1,1}^1 m_3^1 m_2^0 m_1^0\}\end{aligned}$$

and it is regarded as an input tube. The values for d, o, and q are, respectively, one, three, and two. Because the value of the upper bound in Step (1) is three, the algorithm, ParallelOneBitSubtractor $(T_0^{>=}, o, 2 * k - (o - 1) - (k - d + 1 - j), j)$, in Step (1a) will be invoked three times. After the first execution of Step (1a) is run, the result for tube $T_0^{>=}$ is shown in Table VI. Finally, after the third execution for Step (1a) is performed, the result is shown in Table VII. Lemma 6 is applied to prove correction of the algorithm BinaryParallelSubtractor$(T_0^{>=}, d, o, q)$.

Lemma 6: The algorithm BinaryParallelSubtractor$(T_0^{>=}, d, o, q)$ can be applied to finish the function of a binary parallel subtractor.

Proof: Step (1) is the only loop and is mainly used to finish the function of a binary parallel subtractor. On the first execution of Step (1a), it calls the procedure ParallelOneBitSubtractor $(T_0^{>=}, o, 2*k-(o-1)-(k-d+1-j), j)$ to compute the arithmetic subtraction of the least significant bit to the minuend and the subtrahend with the result left in $T_0^{>=}$. Step (1a) is repeated until the most significant bit in the minuend and the subtrahend is processed. Tube $T_0^{>=}$ contains the strands finishing the subtraction operations of at most k bits.

TABLE VI
RESULT IS GENERATED BY PARALLELONEBITSUBTRACTOR$(T_0^{>=}, o, q, j)$

Tube	The result is generated by ParallelOneBitSubtractor($T_0^{>=}$, o, q, j)
$T_0^{>=}$	$\{b_{3,2}^0 n_{4,2}^0 b_{3,1}^0 n_{4,1}^1 b_{3,0}^0 b_{2,6}^0 n_{3,6}^0 b_{2,5}^0 n_{3,5}^0 b_{2,4}^0 n_{3,4}^1 b_{2,3}^0 n_{3,3}^1 b_{2,2}^0 n_{3,2}^1 b_{2,1}^0 n_{3,1}^1 b_{2,0}^0 b_{1,6}^0 n_{2,6}^0 b_{1,5}^0 n_{2,5}^0 b_{1,4}^0 n_{2,4}^1 b_{1,3}^0 n_{2,3}^1 b_{1,2}^0 n_{2,2}^1 b_{1,1}^0 n_{2,1}^1 b_{1,0}^0 n_{1,6}^0 n_{1,5}^0 n_{1,4}^1 n_{1,3}^1 n_{1,2}^1 n_{1,1}^1 m_3^1 m_2^1 m_1^1$, $b_{3,2}^0 n_{4,2}^1 b_{3,1}^0 n_{4,1}^1 b_{3,0}^0 b_{2,6}^0 n_{3,6}^0 b_{2,5}^0 n_{3,5}^0 b_{2,4}^0 n_{3,4}^1 b_{2,3}^0 n_{3,3}^1 b_{2,2}^0 n_{3,2}^1 b_{2,1}^0 n_{3,1}^1 b_{2,0}^0 b_{1,6}^0 n_{2,6}^0 b_{1,5}^0 n_{2,5}^0 b_{1,4}^0 n_{2,4}^1 b_{1,3}^0 n_{2,3}^1 b_{1,2}^0 n_{2,2}^1 b_{1,1}^0 n_{2,1}^1 b_{1,0}^0 n_{1,6}^0 n_{1,5}^0 n_{1,4}^1 n_{1,3}^1 n_{1,2}^1 n_{1,1}^1 m_3^1 m_2^1 m_1^0$, $b_{3,2}^0 n_{4,2}^0 b_{3,1}^0 n_{4,1}^1 b_{3,0}^0 b_{2,6}^0 n_{3,6}^0 b_{2,5}^0 n_{3,5}^0 b_{2,4}^0 n_{3,4}^1 b_{2,3}^0 n_{3,3}^1 b_{2,2}^0 n_{3,2}^1 b_{2,1}^0 n_{3,1}^1 b_{2,0}^0 b_{1,6}^0 n_{2,6}^0 b_{1,5}^0 n_{2,5}^0 b_{1,4}^0 n_{2,4}^1 b_{1,3}^0 n_{2,3}^1 b_{1,2}^0 n_{2,2}^1 b_{1,1}^0 n_{2,1}^1 b_{1,0}^0 n_{1,6}^0 n_{1,5}^0 n_{1,4}^1 n_{1,3}^1 n_{1,2}^1 n_{1,1}^1 m_3^1 m_2^0 m_1^1$, $b_{3,2}^0 n_{4,2}^1 b_{3,1}^0 n_{4,1}^1 b_{3,0}^0 b_{2,6}^0 n_{3,6}^0 b_{2,5}^0 n_{3,5}^0 b_{2,4}^0 n_{3,4}^1 b_{2,3}^0 n_{3,3}^1 b_{2,2}^0 n_{3,2}^1 b_{2,1}^0 n_{3,1}^1 b_{2,0}^0 b_{1,6}^0 n_{2,6}^0 b_{1,5}^0 n_{2,5}^0 b_{1,4}^0 n_{2,4}^1 b_{1,3}^0 n_{2,3}^1 b_{1,2}^0 n_{2,2}^1 b_{1,1}^0 n_{2,1}^1 b_{1,0}^0 n_{1,6}^0 n_{1,5}^0 n_{1,4}^1 n_{1,3}^1 n_{1,2}^1 n_{1,1}^1 m_3^1 m_2^0 m_1^0\}$

TABLE VII
RESULT IS GENERATED BY BINARYPARALLELSUBTRACTOR$(T_0^{>=}, d, o, q)$

Tube	The result is generated by BinaryParallelSubtractor($T_0^{>=}$, d, o, q)
$T_0^{>=}$	$\{b_{3,4}^0 n_{4,4}^0 b_{3,3}^0 n_{4,3}^0 b_{3,2}^0 n_{4,2}^0 b_{3,1}^0 n_{4,1}^1 b_{3,0}^0 b_{2,6}^0 n_{3,6}^0 b_{2,5}^0 n_{3,5}^0 b_{2,4}^0 n_{3,4}^1 b_{2,3}^0 n_{3,3}^1 b_{2,2}^0 n_{3,2}^1 b_{2,1}^0 n_{3,1}^1 b_{2,0}^0 b_{1,6}^0 n_{2,6}^0 b_{1,5}^0 n_{2,5}^0 b_{1,4}^0 n_{2,4}^1 b_{1,3}^0 n_{2,3}^1 b_{1,2}^0 n_{2,2}^1 b_{1,1}^0 n_{2,1}^1 b_{1,0}^0 n_{1,6}^0 n_{1,5}^0 n_{1,4}^1 n_{1,3}^1 n_{1,2}^1 n_{1,1}^1 m_3^1 m_2^1 m_1^1$, $b_{3,4}^0 n_{4,4}^0 b_{3,3}^0 n_{4,3}^0 b_{3,2}^0 n_{4,2}^1 b_{3,1}^0 n_{4,1}^1 b_{3,0}^0 b_{2,6}^0 n_{3,6}^0 b_{2,5}^0 n_{3,5}^0 b_{2,4}^0 n_{3,4}^1 b_{2,3}^0 n_{3,3}^1 b_{2,2}^0 n_{3,2}^1 b_{2,1}^0 n_{3,1}^1 b_{2,0}^0 b_{1,6}^0 n_{2,6}^0 b_{1,5}^0 n_{2,5}^0 b_{1,4}^0 n_{2,4}^1 b_{1,3}^0 n_{2,3}^1 b_{1,2}^0 n_{2,2}^1 b_{1,1}^0 n_{2,1}^1 b_{1,0}^0 n_{1,6}^0 n_{1,5}^0 n_{1,4}^1 n_{1,3}^1 n_{1,2}^1 n_{1,1}^1 m_3^1 m_2^1 m_1^0$, $b_{3,4}^0 n_{4,4}^0 b_{3,3}^0 n_{4,3}^1 b_{3,2}^0 n_{4,2}^0 b_{3,1}^0 n_{4,1}^1 b_{3,0}^0 b_{2,6}^0 n_{3,6}^0 b_{2,5}^0 n_{3,5}^0 b_{2,4}^0 n_{3,4}^1 b_{2,3}^0 n_{3,3}^1 b_{2,2}^0 n_{3,2}^1 b_{2,1}^0 n_{3,1}^1 b_{2,0}^0 b_{1,6}^0 n_{2,6}^0 b_{1,5}^0 n_{2,5}^0 b_{1,4}^0 n_{2,4}^1 b_{1,3}^0 n_{2,3}^1 b_{1,2}^0 n_{2,2}^1 b_{1,1}^0 n_{2,1}^1 b_{1,0}^0 n_{1,6}^0 n_{1,5}^0 n_{1,4}^1 n_{1,3}^1 n_{1,2}^1 n_{1,1}^1 m_3^1 m_2^0 m_1^1$, $b_{3,4}^0 n_{4,4}^0 b_{3,3}^0 n_{4,3}^1 b_{3,2}^0 n_{4,2}^1 b_{3,1}^0 n_{4,1}^1 b_{3,0}^0 b_{2,6}^0 n_{3,6}^0 b_{2,5}^0 n_{3,5}^0 b_{2,4}^0 n_{3,4}^1 b_{2,3}^0 n_{3,3}^1 b_{2,2}^0 n_{3,2}^1 b_{2,1}^0 n_{3,1}^1 b_{2,0}^0 b_{1,6}^0 n_{2,6}^0 b_{1,5}^0 n_{2,5}^0 b_{1,4}^0 n_{2,4}^1 b_{1,3}^0 n_{2,3}^1 b_{1,2}^0 n_{2,2}^1 b_{1,1}^0 n_{2,1}^1 b_{1,0}^0 n_{1,6}^0 n_{1,5}^0 n_{1,4}^1 n_{1,3}^1 n_{1,2}^1 n_{1,1}^1 m_3^1 m_2^0 m_1^0\}$

From BinaryParallelSubtractor$(T_0^{>=}, o, q)$, it takes $(7 * (k - d + 1))$ *extract* operations, $(16 * (k - d + 1))$ *append-head* operations, $(16 * (k - d + 1))$ *detect* operations, $(k - d + 1)$ *merge* operations, and 15 test tubes to compute the arithmetic subtraction of at most k bits. The length of a DNA strand, encoding the difference bit and the borrow bit for the minuend and the subtrahend, is $(30 * (k - d + 1))$ bases.

G. The Construction of a Binary Parallel Divider

A binary parallel divider is a function that performs the arithmetic division for a dividend of $(2 * k)$ bits and a divisor of d bits for $1 \leq d \leq k$. The quotient obtained from the dividend and the divisor can be at most up to $(2 * k)$ bits long. The remainder obtained from the dividend and the divisor can also be at most up to k bits long. Because we only check whether the remainder is equal to zero, therefore, the quotient can be ignored. The following algorithm is proposed to finish the function of a binary parallel divider. The second parameter, d, in the procedure is used to represent the dth division operation.

```
Procedure BinaryParallelDivider($T_0, d$)
 (1) For $o = 1$ to $k + d$
     (1a0) Append-head($T_0, b^0_{o,0}$).
     (1a) ParallelComparator($T_0, T_0^{>}, T_0^{=}, T_0^{<}, d, o$).
     (1b) $T_0^{>=} = \cup(T_0^{>}, T_0^{=})$.
     (1c) If (Detect($T_0^{>=}$) = "yes") then
     (2) For $q = 1$ to $(2 * k) - (o - 1) - (k - d) - 1$
        (2a) $T_1 = +(T_0^{>=}, n^1_{o,q})$ and
$T_2 = -(T_0^{>=}, n^1_{o,q})$.
          (2a1) If (Detect($T_1$) = "yes") then
          (2b) Append-head($T_1, n^1_{o+1,q}$) and
Append-head($T_1, b^0_{o,q}$).
          EndIf
          (2b1) If (Detect($T_2$) = "yes") then
          (2c) Append-head($T_2, n^0_{o+1,q}$) and
Append-head($T_2, b^0_{o,q}$).
          EndIf
          (2d) $T_0^{>=} = -(T_1, T_2)$.
     EndFor
     (3) BinaryParallelSubtractor($T_0^{>=}, d, o, q$).
     (4) For $q = (2 * k) - (o - 1) + 1$ to $2 * k$
          (4a) Append-head($T_0^{>=}, n^0_{o+1,q}$) and
Append-head($T_0^{>=}, b^0_{o,q}$).
        EndFor
     EndIf
     (4b) If (Detect($T_0^{<}$) = "yes") then
     (5) For $q = 1$ to $2 * k$
          (5a) $T_1 = +(T_0^{<}, n^1_{o,q})$ and
$T_2 = -(T_0^{<}, n^1_{o,q})$.
          (5a1) If (Detect($T_1$) = "yes") then
          (5b) Append-head($T_1, n^1_{o+1,q}$) and
Append-head($T_1, b^0_{o,q}$).
          EndIf
          (5b1) If (Detect($T_2$) = "yes") then
          (5c) Append-head($T_2, n^0_{o+1,q}$) and
Append-head($T_2, b^0_{o,q}$).
          EndIf
          (5d) $T_0^{<} = \cup(T_1, T_2)$.
     EndFor
   EndIf
   (6) $T_0 = \cup(T_0^{>=}, T_0^{<})$.
 EndFor
 EndProcedure
```

Consider that the first execution for the algorithm BinaryParallelDivider(T_0, d) invokes tube

$$\begin{aligned}T_0=\{&n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3m^1_2m^1_1, n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}\\&n^1_{1,2}n^1_{1,1}m^1_3m^1_2m^0_1, n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3m^0_2m^1_1,\\&n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3m^0_2m^0_1\}\end{aligned}$$

and it is regarded as an input tube. The value for d is one. Because the value of the upper bound in Step (1) is four, each operation embedded in Step (1) will be run four times. After the first execution for Step (1a0), Step (1a) and Step (1b) is performed, tube $T_0 = \phi$, tube $T_0^{>} = \phi$, tube $T_0^{=} = \phi$, tube $T_0^{>=} = \phi$ and the result for tube $T_0^{<}$ is shown in Table IV. A "no" is returned from the first execution of Step (1c), so Steps (2a) through (2d) are not run. A "yes" is returned from the first execution of Step (4b), so Steps (5a) through (5d) will be run six times. After each operation embedded in Step (5) is finished and the first execution for Step (6) is also performed, tube $T_1 = \phi$, tube $T_2 = \phi$, tube $T_0^{<} = \phi$, tube $T_0^{>=} = \phi$, and tube

$$\begin{aligned}T_0=\{&b^0_{1,6}n^0_{2,6}b^0_{1,5}n^0_{2,5}b^0_{1,4}n^1_{2,4}b^0_{1,3}n^1_{2,3}b^0_{1,2}n^1_{2,2}b^0_{1,1}n^1_{2,1}b^0_{1,0}\\&n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3m^1_2m^1_1, b^0_{1,6}n^0_{2,6}b^0_{1,5}n^0_{2,5}b^0_{1,4}\\&n^1_{2,4}b^0_{1,3}n^1_{2,3}b^0_{1,2}n^1_{2,2}b^0_{1,1}n^1_{2,1}b^0_{1,0}n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}\\&m^1_3m^1_2m^0_1, b^0_{1,6}n^0_{2,6}b^0_{1,5}n^0_{2,5}b^0_{1,4}n^1_{2,4}b^0_{1,3}n^1_{2,3}b^0_{1,2}n^1_{2,2}b^0_{1,1}\\&n^1_{2,1}b^0_{1,0}n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3m^0_2m^1_1, b^0_{1,6}n^0_{2,6}b^0_{1,5}\\&n^0_{2,5}b^0_{1,4}n^1_{2,4}b^0_{1,3}n^1_{2,3}b^0_{1,2}n^1_{2,2}b^0_{1,1}n^1_{2,1}b^0_{1,0}n^0_{1,6}n^0_{1,5}n^1_{1,4}.\end{aligned}$$

Next, the second execution of each operation in the algorithm is performed, tube $T_1 = \phi$, tube $T_2 = \phi$, $T_0^{>} = \phi$, tube $T_0^{=} = \phi$, tube $T_0^{<} = \phi$, tube $T_0^{>=} = \phi$, and tube

$$\begin{aligned}T_0=\{&b^0_{2,6}n^0_{3,6}b^0_{2,5}n^0_{3,5}b^0_{2,4}n^1_{3,4}b^0_{2,3}n^1_{3,3}b^0_{2,2}n^1_{3,2}b^0_{2,1}n^1_{3,1}b^0_{2,0}b^0_{1,6}\\&n^0_{2,6}b^0_{1,5}n^0_{2,5}b^0_{1,4}n^1_{2,4}b^0_{1,3}n^1_{2,3}b^0_{1,2}n^1_{2,2}b^0_{1,1}n^1_{2,1}b^0_{1,0}n^0_{1,6}n^0_{1,5}\\&n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3m^1_2m^1_1, b^0_{2,6}n^0_{3,6}b^0_{2,5}n^0_{3,5}b^0_{2,4}n^1_{3,4}b^0_{2,3}\\&n^1_{3,3}b^0_{2,2}n^1_{3,2}b^0_{2,1}n^1_{3,1}b^0_{2,0}b^0_{1,6}n^0_{2,6}b^0_{1,5}n^0_{2,5}b^0_{1,4}n^1_{2,4}b^0_{1,3}n^1_{2,3}\\&b^0_{1,2}n^1_{2,2}b^0_{1,1}n^1_{2,1}b^0_{1,0}n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3m^1_2m^0_1,\\&b^0_{2,6}n^0_{3,6}b^0_{2,5}n^0_{3,5}b^0_{2,4}n^1_{3,4}b^0_{2,3}n^1_{3,3}b^0_{2,2}n^1_{3,2}b^0_{2,1}n^1_{3,1}b^0_{2,0}b^0_{1,6}\\&n^0_{2,6}b^0_{1,5}n^0_{2,5}b^0_{1,4}n^1_{2,4}b^0_{1,3}n^1_{2,3}b^0_{1,2}n^1_{2,2}b^0_{1,1}n^1_{2,1}b^0_{1,0}n^0_{1,6}n^0_{1,5}\\&n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3m^0_2m^1_1, b^0_{2,6}n^0_{3,6}b^0_{2,5}n^0_{3,5}b^0_{2,4}n^1_{3,4}b^0_{2,3}\\&n^1_{3,3}b^0_{2,2}n^1_{3,2}b^0_{2,1}n^1_{3,1}b^0_{2,0}b^0_{1,6}n^0_{2,6}b^0_{1,5}n^0_{2,5}b^0_{1,4}n^1_{2,4}b^0_{1,3}n^1_{2,3}\\&b^0_{1,2}n^1_{2,2}b^0_{1,1}n^1_{2,1}b^0_{1,0}n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3m^0_2m^0_1\}.\end{aligned}$$

After the third execution for Steps (1a0) through Step (1c) is finished, a "yes" is returned from the third execution of Step (1c). Because the value of the upper bound in Step (2) is one, each operation embedded in Step (2) will be run one time. After those operations embedded in Step (2) are run, tube $T_1 = \phi$, tube $T_2 = \phi$, and tube

$$\begin{aligned}T_0^{>=}=\{&b^0_{3,1}n^1_{4,1}b^0_{3,0}b^0_{2,6}n^0_{3,6}b^0_{2,5}n^0_{3,5}b^0_{2,4}n^1_{3,4}b^0_{2,3}n^1_{3,3}b^0_{2,2}n^1_{3,2}\\&b^0_{2,1}n^1_{3,1}b^0_{2,0}b^0_{1,6}n^0_{2,6}b^0_{1,5}n^0_{2,5}b^0_{1,4}n^1_{2,4}b^0_{1,3}n^1_{2,3}b^0_{1,2}n^1_{2,2}b^0_{1,1}\\&n^1_{2,1}b^0_{1,0}n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3m^1_2m^1_1, b^0_{3,1}n^1_{4,1}\\&b^0_{3,0}b^0_{2,6}n^0_{3,6}b^0_{2,5}n^0_{3,5}b^0_{2,4}n^1_{3,4}b^0_{2,3}n^1_{3,3}b^0_{2,2}n^1_{3,2}b^0_{2,1}n^1_{3,1}b^0_{2,0}\\&b^0_{1,6}n^0_{2,6}b^0_{1,5}n^0_{2,5}b^0_{1,4}n^1_{2,4}b^0_{1,3}n^1_{2,3}b^0_{1,2}n^1_{2,2}b^0_{1,1}n^1_{2,1}b^0_{1,0}n^0_{1,6}\\&n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3m^1_2m^0_1, b^0_{3,1}n^1_{4,1}b^0_{3,0}b^0_{2,6}n^0_{3,6}b^0_{2,5}\\&n^0_{3,5}b^0_{2,4}n^1_{3,4}b^0_{2,3}n^1_{3,3}b^0_{2,2}n^1_{3,2}b^0_{2,1}n^1_{3,1}b^0_{2,0}b^0_{1,6}n^0_{2,6}b^0_{1,5}n^0_{2,5}\\&b^0_{1,4}n^1_{2,4}b^0_{1,3}n^1_{2,3}b^0_{1,2}n^1_{2,2}b^0_{1,1}n^1_{2,1}b^0_{1,0}n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}\\&n^1_{1,2}n^1_{1,1}m^1_3m^0_2m^1_1, b^0_{3,1}n^1_{4,1}b^0_{3,0}b^0_{2,6}n^0_{3,6}b^0_{2,5}n^0_{3,5}b^0_{2,4}n^1_{3,4}\\&b^0_{2,3}n^1_{3,3}b^0_{2,2}n^1_{3,2}b^0_{2,1}n^1_{3,1}b^0_{2,0}b^0_{1,6}n^0_{2,6}b^0_{1,5}n^0_{2,5}b^0_{1,4}n^1_{2,4}b^0_{1,3}\\&n^1_{2,3}b^0_{1,2}n^1_{2,2}b^0_{1,1}n^1_{2,1}b^0_{1,0}n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3m^0_2\\&m^0_1\}.\end{aligned}$$

After the first execution for Step (3) invokes the algorithm, BinaryParallelSubtractor$(T_0^{>=}, d, o, q)$, the result is shown in Table VII. Next, after the rest of operations in

TABLE VIII
RESULT IS GENERATED BY BINARYPARALLELDIVIDER(T_0, d)

Tube	The result is generated by BinaryParallelDivider(T_0, d)
T_0	$\{b_{4,6}^{0}n_{5,6}^{0}b_{4,5}^{0}n_{5,5}^{0}b_{4,4}^{0}n_{5,4}^{0}b_{4,3}^{0}n_{5,3}^{0}b_{4,2}^{0}n_{5,2}^{0}b_{4,1}^{0}n_{5,1}^{1}b_{4,0}^{0}b_{3,6}^{0}n_{4,6}^{0}b_{3,5}^{0}n_{4,5}^{0}b_{3,4}^{0}n_{4,4}^{0}b_{3,3}^{0}n_{4,3}^{0}b_{3,2}^{0}n_{4,2}^{0}b_{3,1}^{0}n_{4,1}^{1}b_{3,0}^{0}b_{2,6}^{0}n_{3,6}^{0}b_{2,5}^{0}n_{3,5}^{0}b_{2,4}^{0}n_{3,4}^{1}b_{2,3}^{0}n_{3,3}^{1}b_{2,2}^{0}n_{3,2}^{1}b_{2,1}^{0}n_{3,1}^{1}b_{2,0}^{0}b_{1,6}^{0}n_{2,6}^{0}b_{1,5}^{0}n_{2,5}^{0}b_{1,4}^{0}n_{2,4}^{1}b_{1,3}^{0}n_{2,3}^{1}b_{1,2}^{0}n_{2,2}^{1}b_{1,1}^{0}n_{2,1}^{1}b_{1,0}^{0}n_{1,6}^{0}\,n_{1,5}^{0}\,n_{1,4}^{1}\,n_{1,3}^{1}\,n_{1,2}^{1}\,n_{1,1}^{1}m_3^{1}m_2^{1}m_1^{1},$
	$b_{4,6}^{0}n_{5,6}^{0}b_{4,5}^{0}n_{5,5}^{0}b_{4,4}^{0}n_{5,4}^{0}b_{4,3}^{0}n_{5,3}^{0}b_{4,2}^{0}n_{5,2}^{1}b_{4,1}^{0}n_{5,1}^{1}b_{4,0}^{0}b_{3,6}^{0}n_{4,6}^{0}b_{3,5}^{0}n_{4,5}^{0}b_{3,4}^{0}n_{4,4}^{0}b_{3,3}^{0}n_{4,3}^{0}b_{3,2}^{0}n_{4,2}^{1}b_{3,1}^{0}n_{4,1}^{1}b_{3,0}^{0}b_{2,6}^{0}n_{3,6}^{0}b_{2,5}^{0}n_{3,5}^{0}b_{2,4}^{0}n_{3,4}^{1}b_{2,3}^{0}n_{3,3}^{1}b_{2,2}^{0}n_{3,2}^{1}b_{2,1}^{0}n_{3,1}^{1}b_{2,0}^{0}b_{1,6}^{0}n_{2,6}^{0}b_{1,5}^{0}n_{2,5}^{0}b_{1,4}^{0}n_{2,4}^{1}b_{1,3}^{0}n_{2,3}^{1}b_{1,2}^{0}n_{2,2}^{1}b_{1,1}^{0}n_{2,1}^{1}b_{1,0}^{0}n_{1,6}^{0}\,n_{1,5}^{0}\,n_{1,4}^{1}\,n_{1,3}^{1}\,n_{1,2}^{1}\,n_{1,1}^{1}m_3^{1}m_2^{1}m_1^{0},$
	$b_{4,6}^{0}n_{5,6}^{0}b_{4,5}^{0}n_{5,5}^{0}b_{4,4}^{0}n_{5,4}^{0}b_{4,3}^{0}n_{5,3}^{0}b_{4,2}^{0}n_{5,2}^{0}b_{4,1}^{0}n_{5,1}^{0}b_{4,0}^{0}b_{3,6}^{0}n_{4,6}^{0}b_{3,5}^{0}n_{4,5}^{0}b_{3,4}^{0}n_{4,4}^{0}b_{3,3}^{0}n_{4,3}^{1}b_{3,2}^{0}n_{4,2}^{0}b_{3,1}^{0}n_{4,1}^{1}b_{3,0}^{0}b_{2,6}^{0}n_{3,6}^{0}b_{2,5}^{0}n_{3,5}^{0}b_{2,4}^{0}n_{3,4}^{1}b_{2,3}^{0}n_{3,3}^{1}b_{2,2}^{0}n_{3,2}^{1}b_{2,1}^{0}n_{3,1}^{1}b_{2,0}^{0}b_{1,6}^{0}n_{2,6}^{0}b_{1,5}^{0}n_{2,5}^{0}b_{1,4}^{0}n_{2,4}^{1}b_{1,3}^{0}n_{2,3}^{1}b_{1,2}^{0}n_{2,2}^{1}b_{1,1}^{0}n_{2,1}^{1}b_{1,0}^{0}n_{1,6}^{0}\,n_{1,5}^{0}\,n_{1,4}^{1}\,n_{1,3}^{1}\,n_{1,2}^{1}\,n_{1,1}^{1}m_3^{1}m_2^{0}m_1^{1},$
	$b_{4,6}^{0}n_{5,6}^{0}b_{4,5}^{0}n_{5,5}^{0}b_{4,4}^{0}n_{5,4}^{0}b_{4,3}^{0}n_{5,3}^{0}b_{4,2}^{0}n_{5,2}^{1}b_{4,1}^{0}n_{5,1}^{1}b_{4,0}^{0}b_{3,6}^{0}n_{4,6}^{0}b_{3,5}^{0}n_{4,5}^{0}b_{3,4}^{0}n_{4,4}^{0}b_{3,3}^{0}n_{4,3}^{1}b_{3,2}^{0}n_{4,2}^{1}b_{3,1}^{0}n_{4,1}^{1}b_{3,0}^{0}b_{2,6}^{0}n_{3,6}^{0}b_{2,5}^{0}n_{3,5}^{0}b_{2,4}^{0}n_{3,4}^{1}b_{2,3}^{0}n_{3,3}^{1}b_{2,2}^{0}n_{3,2}^{1}b_{2,1}^{0}n_{3,1}^{1}b_{2,0}^{0}b_{1,6}^{0}n_{2,6}^{0}b_{1,5}^{0}n_{2,5}^{0}b_{1,4}^{0}n_{2,4}^{1}b_{1,3}^{0}n_{2,3}^{1}b_{1,2}^{0}n_{2,2}^{1}b_{1,1}^{0}n_{2,1}^{1}b_{1,0}^{0}n_{1,6}^{0}\,n_{1,5}^{0}\,n_{1,4}^{1}\,n_{1,3}^{1}\,n_{1,2}^{1}\,n_{1,1}^{1}m_3^{1}m_2^{0}m_1^{0}\}$

BinaryParallelDivider(T_0, d) are performed, tube $T_1 = \phi$, tube $T_2 = \phi$, $T_0^{>} = \phi$, tube $T_0^{=} = \phi$, tube $T_0^{<} = \phi$, tube $T_0^{>=} = \phi$, and the result for tube T_0 is shown in Table VIII. Lemma 7 is used to show correction of the algorithm BinaryParallelDivider(T_0, d).

Lemma 7: The algorithm BinaryParallelDivider(T_0, d) can be applied to finish the function of a binary parallel divider.

Proof: The division to a dividend of $(2 * k)$ bits and a divisor of d bits for $1 \le d \le k$ is finished through of successive compare, shift, and subtract operations of at most $(2*k)$ times. When the first compare, shift, and subtract operations, the least significant position for the dividend and the divisor is subtracted, the input borrow bit must be zero. Step (1) is the main loop and is applied to finish the function of a binary parallel divider. So each execution of Step (1a0) uses the *append-head* operation to append 15-based DNA sequences for representing $b_{o,0}^{0}$ onto the head of every strand in T_0. On each execution of Step (1a), it calls ParallelComparator$(T_0, T_0^{>}, T_0^{=}, T_0^{<}, d, o)$ to compare the divisor with the corresponding bits of the dividend. After it is finished, three tubes are generated and are, respectively, $T_0^{>}$, $T_0^{=}$, and $T_0^{<}$. The first tube $T_0^{>}$ includes the strands with the comparative result of greater than (">"). The second tube $T_0^{=}$ includes the strands with the comparative result of equal ("="). The third tube $T_0^{<}$ consists of the strands with the comparative result of less than ("<"). Next, each execution of Step (1b) employs the *merge* operation to pour tubes $T_0^{>}$ and $T_0^{=}$ into $T_0^{>=}$. On each execution Step (1c) applies the *detect* operation to check whether tube $T_0^{>=}$ contains any DNA strand or not. If a "yes" is returned, then Step (2) through Step (4a) will be run. Otherwise, those steps will not be executed. Step (2) is a loop and is used mainly to reserve the least significant $((2 * k) - (o - 1) - (k - d) - 1)$ bits of the dividend. This implies that the least significant $((2 * k) - (o - 1) - (k - d) - 1)$ bits of the minuend (dividend) for the oth compare, shift, and subtract operations are reserved. And they are equal to the least significant $((2*k)-(o-1)-(k-d)-1)$ bits of the difference for the same operations. Therefore, on each execution of Step (2a), it uses the *extract* operation to form two test tubes: T_1 and T_2. The first tube T_1 includes all of the strands that have $n_{o,q} = 1$. The second tube T_2 consists of all of the strands that have $n_{o,q} = 0$. On each execution Step (2a1) uses the *detect* operation to test if tube T_1 contains any DNA strand. If a "yes" is returned, then Step (2b) will be run. Otherwise, that step will not be executed. Next, each execution of Step (2b) uses the *append-head* operations to append $n_{o+1,q}^{1}$ and $b_{o,q}^{0}$ onto the head of every strand in T_1. Each execution of Step (2b1) applies the *detect* operation to examine if tube T_2 contains any DNA strand. If a "yes" is returned, then Step (2c) will be run. Otherwise, that step will not be executed. On each execution of Step (2c), it applies the *append-head* operations to append $n_{o+1,q}^{0}$ and $b_{o,q}^{0}$ onto the head of every strand in T_2. Then, each execution of Step (2d) employs the *merge* operation to pour tubes T_1 and T_2 into $T_0^{>=}$. Tube $T_0^{>=}$ contains the strands finishing compare, shift, and subtract operations of a bit. Repeat execution of Steps (2a) through (2d) until the least significant $((2 * k) - (o - 1) - (k - d) - 1)$ bits of the minuend (dividend) are processed. Tube $T_0^{>=}$ contains the strands finishing compare, shift, and subtract operations of the least significant $((2 * k) - (o - 1) - (k - d) - 1)$ bits of the minuend (dividend).

Next, when each execution of Step (3) calls the algorithm BinaryParallelSubtractor$(T_0^{>=}, d, o, q)$ to finish compare, shift, and subtract operations of $(k - d + 1)$ bits. Step (4) is a loop and it is used to finish compare, shift, and subtract operations of the most significant $(o - 1)$ bits in the minuend (dividend). Because the most significant $(o-1)$ bits in the minuend (dividend) for the oth compare, shift, and subtract operations are all zero, the most significant $(o-1)$ bits of the difference to the oth compare, shift, and subtract operations are equal to the most significant $(o - 1)$ bits of the minuend to the same operations. On each execution of Step (4a), it applies the *append-head* operations to append $n_{o+1,q}^{0}$ and $b_{o,q}^{0}$ onto the head of every strand in $T_0^{>=}$. Repeat execution of Step (4a) until the most significant $(o - 1)$ bits of the minuend are processed. Tube $T_0^{>=}$ contains the strands finishing the oth compare, shift, and subtract operations for the comparative result of greater than or equal to (">=").

Next, each execution of Step (4b) applies the *detect* operation to check whether tube $T_0^{<}$ contains any DNA strand or not. If a "yes" is returned, then Step (5) through Step (5d) will be run. Otherwise, those steps will not be executed. Since $T_0^{<}$ consists of all of the strands with the comparative result of less than ("<"). This implies that the $(2 * k)$ bits of the difference to the oth compare, shift, and subtract operations are equal to the $(2*k)$ bits of the minuend to the same operations. Step (5) is a loop and is employed to finish the oth compare, shift, and subtract operations for tube $T_0^{<}$. On each execution of Step (5a), it employs the *extract* operation to form two test tubes: T_1 and T_2. The first tube T_1 includes all of the strands that have $n_{o,q} = 1$. The second tube T_2 consists of all of the strands that have $n_{o,q} = 0$. On each execution Step (5a1) uses the *detect* operation to test if tube T_1 contains any DNA strand. If a "yes" is returned, then Step (5b) will be run. Otherwise, that step will not be executed.

Next, each execution of Step (5b) uses the *append-head* operations to append $n^1_{o+1,q}$ and $b^0_{o,q}$ onto the head of every strand in T_1. Each execution of Step (5b1) applies the *detect* operation to examine whether tube T_2 contains any DNA strand or not. If a "yes" is returned, then Step (5c) will be run. Otherwise, that step will not be executed. On each execution of Step (5c), it applies the *append-head* operations to append $n^0_{o+1,q}$ and $b^0_{o,q}$ onto the head of every strand in T_2. Then, each execution of Step (5d) applies the *merge* operation to pour tubes T_1 and T_2 into $T_0^{<}$. Tube $T_0^{<}$ contains the strands finishing compare, shift, and subtract operations of a bit. Repeat execution of Steps (5a) through (5d) until the $(2*k)$ bits are processed. Tube $T_0^{<}$ contains the strands finishing compare, shift, and subtract operations of the $(2*k)$ bits for the oth compare, shift, and subtract operations to the comparative result of less than ("<").

Next, each execution of Step (6) applies the *merge* operation to pour tubes $T_0^{>=}$ and $T_0^{<}$ into T_0. Tube T_0 contains the strands finishing the oth compare, shift, and subtract operations of $(2*k)$ bits for the comparative results of greater than or equal to or less than. Repeat execution of the steps above until successive compare, shift, and subtract operations of at most $(2*k)$ times are processed. Tube T_0 contains the strands finishing a division for a dividend of $(2*k)$ bits and a divisor of d bits for $1 \le d \le k$.

From BinaryParallelDivider(T_0), it takes $(13*k^2 + 4*k*d + 9*k - 9*d^2 + 9*d)$ *extract* operations, $(27*k^2 + 14*k*d + 13*k - 13*d^2 + 13*d + 1)$ *append-head* operations, $(7*k^2 + 4*k*d + 6*k - 3*d^2 + 6*d)$ *merge* operations, $((29*k^2 + 14*k*d + 19*k - 15*d^2 + 19*d) \div 2)$ *detect* operations, and 22 tubes to compute the division operation. The length of a DNA strand, encoding the difference bits and the borrow bits, is $(60*k^2 + 60*k*d + 15)$ bases.

H. Finding Two Large Prime Numbers of k Bits

The following DNA algorithm is applied to find two large prime numbers of k bits.

```
Algorithm 1: Finding two large prime numbers of
$k$ bits
(1) InitialSolution($T_0$).
(2) InitialProduct($T_0$).
(3) For $d = 1$ to $k$
  (3a) $T_0 = +(T_0, m^1_{k-d+1})$ and
$T_{\text{off}} = -(T_0, m^1_{k-d+1})$.
  (3b) BinaryParallelDivider($T_0, d$).
  (3c) For $q = 1$ to $k - d + 1$
  (3d) $T_0 = +(T_0 n^0_{k+d+1,q})$ and
$T_{\text{bad}} = -(T_0, n^0_{k+d+1,q})$.
  (3e) Discard($T_{\text{bad}}$).
  (3f) If (Detect($T_0$) = "no") then
    (3g) Terminate the execution of the
second (inner) loop.
    EndIf
  EndFor
  (3h) If (Detect($T_0$) = "yes") then
    (3i) Read($T_0$) and then terminate
the algorithm.
    EndIf
  (3j) $T_0 = \cup(T_0, T_{\text{off}})$.
EndFor
EndAlgorithm
```

Consider that the value for n is 001 111. Algorithm 1 is used to factor n into three and five. Tube T_0 is an empty tube and is regarded as an input tube for Algorithm 1. After the execution for Step (1) is performed, the result for tube T_0 is shown in Table I. Next, after the execution for Step (2) is finished, the result for tube T_0 is shown in Table II. Because the value for k is three, each operation embedded in Step (3) will be at most run three times. After the first execution for Step (3a) is performed, tube

$$T_0 = \{n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3m^1_2m^1_1, n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3} n^1_{1,2}n^1_{1,1}m^1_3m^1_2m^0_1, n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3m^0_2m^1_1, n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3m^0_2m^0_1\}$$

and tube

$$T_{\text{OFF}} = \{N^0_{1,6}N^0_{1,5}N^1_{1,4}N^1_{1,3}N^1_{1,2}N^1_{1,1}M^0_3M^1_2M^1_1, N^0_{1,6}N^0_{1,5} n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^0_3m^1_2m^0_1, n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1} m^0_3m^0_2m^1_1, n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^0_3m^0_2m^0_1\}.$$

Next, after the first execution of Step (3b) is finished, the result for tube T_0 is shown in Table VIII.

Since the value of the upper bound in Step (3c) is three, each operation embedded in Step (3c) will be at most run three times. Next, after the first execution of Step (3d) is run, tube

$$T_0 = \{b^0_{4,6}n^0_{5,6}b^0_{4,5}n^0_{5,5}b^0_{4,4}n^0_{5,4}b^0_{4,3}n^0_{5,3}b^0_{4,2}n^0_{5,2}b^0_{4,1}n^0_{5,1}b^0_{4,0}b^0_{3,6} n^0_{4,6}b^0_{3,5}n^0_{4,5}b^0_{3,4}n^0_{4,4}b^0_{3,3}n^1_{4,3}b^0_{3,2}n^0_{4,2}b^0_{3,1}n^1_{4,1}b^0_{3,0}b^0_{2,6}n^0_{3,6} b^0_{2,5}n^0_{3,5}b^0_{2,4}n^1_{3,4}b^0_{2,3}n^1_{3,3}b^0_{2,2}n^1_{3,2}b^0_{2,1}n^1_{3,1}b^0_{2,0}b^0_{1,6}n^0_{2,6}b^0_{1,5} n^0_{2,5}b^0_{1,4}n^1_{2,4}b^0_{1,3}n^1_{2,3}b^0_{1,2}n^1_{2,2}b^0_{1,1}n^1_{2,1}b^0_{1,0}n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3} n^1_{1,2}n^1_{1,1}m^1_3m^0_2m^1_1\}$$

and tube

$$T_{\text{bad}} = \{b^0_{4,6}n^0_{5,6}b^0_{4,5}n^0_{5,5}b^0_{4,4}n^0_{5,4}b^0_{4,3}n^0_{5,3}b^0_{4,2}n^0_{5,2}b^0_{4,1}n^1_{5,1}b^0_{4,0} b^0_{3,6}n^0_{4,6}b^0_{3,5}n^0_{4,5}b^0_{3,4}n^0_{4,4}b^0_{3,3}n^0_{4,3}b^0_{3,2}n^0_{4,2}b^0_{3,1}n^1_{4,1}b^0_{3,0}b^0_{2,6} n^0_{3,6}b^0_{2,5}n^0_{3,5}b^0_{2,4}n^1_{3,4}b^0_{2,3}n^1_{3,3}b^0_{2,2}n^1_{3,2}b^0_{2,1}n^1_{3,1}b^0_{2,0}b^0_{1,6}n^0_{2,6} b^0_{1,5}n^0_{2,5}b^0_{1,4}n^1_{2,4}b^0_{1,3}n^1_{2,3}b^0_{1,2}n^1_{2,2}b^0_{1,1}n^1_{2,1}b^0_{1,0}n^0_{1,6}n^0_{1,5} n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3m^1_2m^1_1, b^0_{4,6}n^0_{5,6}b^0_{4,5}n^0_{5,5}b^0_{4,4}n^0_{5,4}b^0_{4,3} n^0_{5,3}b^0_{4,2}n^1_{5,2}b^0_{4,1}n^1_{5,1}b^0_{4,0}b^0_{3,6}n^0_{4,6}b^0_{3,5}n^0_{4,5}b^0_{3,4}n^0_{4,4}b^0_{3,3}n^0_{4,3} b^0_{3,2}n^1_{4,2}b^0_{3,1}n^1_{4,1}b^0_{3,0}b^0_{2,6}n^0_{3,6}b^0_{2,5}n^0_{3,5}b^0_{2,4}n^1_{3,4}b^0_{2,3}n^1_{3,3}b^0_{2,2} n^1_{3,2}b^0_{2,1}n^1_{3,1}b^0_{2,0}b^0_{1,6}n^0_{2,6}b^0_{1,5}n^0_{2,5}b^0_{1,4}n^1_{2,4}b^0_{1,3}n^1_{2,3}b^0_{1,2}n^1_{2,2} b^0_{1,1}n^1_{2,1}b^0_{1,0}n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2}n^1_{1,1}m^1_3m^1_2m^0_1, b^0_{4,6}n^0_{5,6} b^0_{4,5}n^0_{5,5}b^0_{4,4}n^0_{5,4}b^0_{4,3}n^0_{5,3}b^0_{4,2}n^1_{5,2}b^0_{4,1}n^1_{5,1}b^0_{4,0}b^0_{3,6}n^0_{4,6}b^0_{3,5} n^0_{4,5}b^0_{3,4}n^0_{4,4}b^0_{3,3}n^1_{4,3}b^0_{3,2}n^1_{4,2}b^0_{3,1}n^1_{4,1}b^0_{3,0}b^0_{2,6}n^0_{3,6}b^0_{2,5}n^0_{3,5} b^0_{2,4}n^1_{3,4}b^0_{2,3}n^1_{3,3}b^0_{2,2}n^1_{3,2}b^0_{2,1}n^1_{3,1}b^0_{2,0}b^0_{1,6}n^0_{2,6}b^0_{1,5}n^0_{2,5}b^0_{1,4} n^1_{2,4}b^0_{1,3}n^1_{2,3}b^0_{1,2}n^1_{2,2}b^0_{1,1}n^1_{2,1}b^0_{1,0}n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}n^1_{1,2} n^1_{1,1}m^1_3m^0_2m^0_1\}.$$

Next, after the first execution for Step (3e) is performed, tube $T_{\text{bad}} = \phi$. A "yes" is returned from the first execution of Step (3f), so the first execution for Step (3g) is not run.

Next, after the rest of operations for Steps (3d) through (3f) are performed, tube

$$\begin{aligned}T_0=\{&b^0_{4,6}n^0_{5,6}b^0_{4,5}n^0_{5,5}b^0_{4,4}n^0_{5,4}b^0_{4,3}n^0_{5,3}b^0_{4,2}n^0_{5,2}b^0_{4,1}n^0_{5,1}b^0_{4,0}b^0_{3,6}\\ &n^0_{4,6}b^0_{3,5}n^0_{4,5}b^0_{3,4}n^0_{4,4}b^0_{3,3}n^1_{4,3}b^0_{3,2}n^0_{4,2}b^0_{3,1}n^1_{4,1}b^0_{3,0}b^0_{2,6}n_{3,6}\\ &{}^0b^0_{2,5}n^0_{3,5}b^0_{2,4}n^1_{3,4}b^0_{2,3}n^1_{3,3}b^0_{2,2}n^1_{3,2}b^0_{2,1}n^1_{3,1}b^0_{2,0}b^0_{1,6}n^0_{2,6}b^0_{1,5}\\ &n^0_{2,5}b^0_{1,4}n^1_{2,4}b^0_{1,3}n^1_{2,3}b^0_{1,2}n^1_{2,2}b^0_{1,1}n^1_{2,1}b^0_{1,0}n^0_{1,6}n^0_{1,5}n^1_{1,4}n^1_{1,3}\\ &n^1_{1,2}n^1_{1,1}m^1_3m^0_2m^1_1\}\end{aligned}$$

and tube $T_{\text{bad}} = \phi$. A "yes" is returned from the first execution of Step (3h). Next, the answer is five from the first execution of Step (3i) and Algorithm 1 is terminated from the first execution of Step (3i). Since one of two primers is five, another primer is equal to three. Theorem 1 is used to show correction of Algorithm 1.

Theorem 1: From those steps in Algorithm 1, the difficulty of factoring the product of two large prime numbers of k bits is solved.

Proof: On the execution of Step (1), it calls InitialSolution(T_0) to construct solution space of DNA strands for every unsigned integer of k bits. This means that tube T_0 includes strands encoding 2^k different integer values. Next, the execution of Step (2) calls InitialProduct(T_0) to append DNA sequences of encoding n, the product of two large prime numbers of k bits, onto the head of every strand in tube T_0. This implies that the front $(2 * k)$ bits and the last k bits of every strand in T_0, respectively, represent the dividend and the divisor of a division instruction after Step (2) is performed.

Step (3) is two level loops and is mainly used to factor the product of two large prime numbers of k bits. On each execution of Step (3a), it uses the *extract* operation to form two tubes: T_0 and T_{off}. The first tube T_0 includes all of the strands that have $m_{k-d+1} = 1$. This is to say that the $(k - d + 1)$th bit of every divisor in T_0 is equal to one. The second tube T_{off} consists of all of the strands that have $m_{k-d+1} = 0$. This indicates that the $(k-d+1)$th bit of every divisor in T_{off} is equal to zero. Because the front d bits of every divisor in T_{off} are all zeros, therefore, the dth division instruction is not applied to compute the remainder of every strand in T_{off}. Next, each execution of Step (3b) calls BinaryParallelDivider(T_0, d). The procedure is used to finish a division instruction. After Step (3b) is performed, the remainder of every strand in T_0 is computed. Step (3c) is the inner loop and is mainly employed to judge whether the remainder of a division operation is equal to zero. On each execution of Step (3d), it uses the *extract* operation to form two tubes: T_0 and T_{bad}. The first tube T_0 includes all of the strands that have $n_{k+d+1,q} = 0$. This means that the qth bit of every remainder in T_0 is equal to zero. The second tube T_{bad} consists of all of the strands that have $n_{k+d+1,q} = 1$. This implies that the qth bit of every remainder in T_{bad} is equal to one. Since the strands in T_{bad} encode every remainder that is not equal to zero, Step (3e) is used to discard T_{bad}. Then, each execution of Step (3f) applies the *detect* operation to check whether tube T_0 contains any DNA strand or not. If a "no" is returned, then this indicates that all of the remainders in T_0 for the dth division operation are not equal to zero. Therefore, Step (3g) is employed to terminate the execution of the inner loop. If a "yes" is returned, then repeat the steps until the number of the execution of the inner loop is performed.

After the inner loop is performed, Step (3h) is applied to detect whether T_0 contains any DNA strands or not. If it returns a "yes," then DNA sequences in T_0 represent the remainders that are equal to zero. Hence, Step (3i) is used to find the answer (one of two large prime numbers) from T_0. Simultaneously, the algorithm is terminated. If it returns a "no," then Step (3j) is employed to pour tube T_{off} into tube T_0. This is to say that T_0 reserves the strands that have $m_{k-d+1} = 0$. Repeat the steps until the number of the execution of the outer loop is performed. Finally, the strands in T_0 encode every strand that is zero. This indicates that the only two large prime numbers of k bits are in T_0. Therefore, it is inferred that the difficulty of factoring the product of two large prime numbers of k bits is solved from those steps in Algorithm 1.

I. Breaking the RSA Public-Key Cryptosystem

The RSA public-key cryptosystem can be used to encrypt messages sent between two communicating parties so that an eavesdropper who overhears the encrypted message will not be able to decode them. Assume that the encrypted message overheard is represented as C (the corresponding cipher-text). An eavesdropper only needs to use the following algorithm to decode them.

```
Algorithm 2: Breaking the RSA Public-key
Cryptosystem
(1) Call Algorithm 1.
(2) Compute the secret key $d$, from the multi-
plicative inverse of
$e$, module $(p-1)*(q-1)$ on a classical computer.
(3) Decode the messages overheard through the
decryption
function, $C^d$ (module $n$) on a classical computer.
EndAlgorithm
```

Theorem 2: From the steps in Algorithm 2, an eavesdropper can decode the encrypted message overheard.

Proof: Refer to Algorithm 1.

J. The Complexity of Algorithm 1

Lemma 8: Suppose that the length of n, the product of two large prime numbers of k bits is $(2 * k)$ bits. The difficulty of factoring n can be solved with $O(k^3)$ biological operations solution space of DNA strands.

Proof: Refer to Algorithm 1.

Lemma 9: Suppose that the length of n, the product of two large prime numbers of k bits, is $(2 * k)$ bits. The difficulty of factoring n can be solved with $O(2^k)$ library strands from solution space of DNA strands.

Proof: Refer to Algorithm 1.

Lemma 10: Suppose that the length of n, the product of two large prime numbers of k bits, is $(2 * k)$ bits. The difficulty of factoring n can be solved with $O(1)$ tubes from solution space of DNA strands.

Proof: Refer to Algorithm 1.

Lemma 11: Suppose that the length of n, the product of two large prime numbers of k bits, is $(2 * k)$ bits. The difficulty of factoring n can be solved with the longest library strand, $O(k^2)$, from solution space of DNA strands.

Proof: Refer to Algorithm 1.

IV. Discussion

The proposed algorithm (Algorithm 1) for factoring the product of two large prime numbers of k bits is based on biological operations from solution space of DNA strands. This algorithm has several advantages from biological operations and solution space of DNA strands. First, the Adleman program [22], [46] was used to generate good DNA sequences to construct the solution space of DNA strands. Good DNA sequences were applied to decrease a rate of errors for hybridization. This indicates that the proposed algorithm actually has a lower rate of errors for hybridization.

Second, basic biological operations were employed to finish the function of a k-bit parallel comparator, the function of a parallel subtractor, and the function of a parallel divider. This means that the proposed algorithm has the computational capability of mathematics to finish subtraction ("$-$") and division ("$\div$"). Basic biological operations had been performed in a fully automated manner in their lab. The full automation manner is essential not only for the speedup of computation but also for error-free computation.

Third, in Algorithm 1 for factoring the product of two large prime numbers of k bits, the number of tubes, the longest length of DNA strands, the number of DNA strands, and the number of biological operations, respectively, are $O(1)$, $O(k^2)$, $O(2^k)$, and $O(k^3)$. This implies that the proposed algorithm can be easily performed in a fully automated manner in a lab. Fourthly, after n is factored as $p * q$ from Algorithm 1, decoding an encrypted message overheard is performed on a classical computer. This is to say that decoding an overheard encrypted message can be easily implemented on a classical computer after n is factored as $p * q$.

V. Conclusion

A general *digital* computer mainly contains the CPU and memory. The main function for the CPU is to perform mathematical computational tasks and the main function to memory is to store each data needed for mathematical computational tasks. However, on a general molecular computer, each data needed for mathematical computational tasks is encoded by means of a DNA strand and performing mathematical computational tasks is by means of a DNA algorithm (including a series of basic biological operations) on those DNA strands. The execution time for any basic biological operation is very longer than that of a *digital* mathematical instruction. Hence, in order to significantly improve the execution time for any basic biological operation, Adleman [2] indicated that exponential DNA strands are necessary. This implies that by means of a basic biological operation on exponential DNA strands can be used to perform exponential *digital* mathematical instructions.

The paper is the first paper that demonstrates that the difficult problem for factoring the product of two large prime numbers of k bits can be solved on a DNA-based computer. The proposed algorithm takes a number of steps that is polynomial in the input size, e.g., the number of binary digits of the product (integer) to be factored. Simultaneously, the paper also shows that humans' mathematical operations can directly be performed with basic biological operations. The property for the difficulty of factoring the product of two large prime numbers is the basis of cryptosystems using public key. However, the property seems to be incorrect on a molecular computer. This indicates that the cryptosystems using public key are perhaps insecure. Furthermore, the first example of *molecular cryptanalysis* for cryptosystems based on public key is proposed in the paper.

Currently the future of molecular computers is unclear. It is possible that in the future molecular computers will be the clear choice for performing massively parallel computations. However, there are still many technical difficulties to overcome before this becomes a reality. We hope that this paper helps to demonstrate that molecular computing is a technology worth pursuing.

Acknowledgment

The authors would to thank the anonymous reviewers for extensive comments and suggestions, which improved the presentation of this paper considerably.

References

[1] R. R. Sinden, *DNA Structure and Function*. New York: Academic, 1994.
[2] L. Adleman, "Molecular computation of solutions to combinatorial problems," *Science*, vol. 266, pp. 1021–1024, Nov. 11, 1994.
[3] R. J. Lipton, "DNA solution of hard computational problems," *Science*, vol. 268, pp. 542–545, 1995.
[4] Q. Quyang, P. D. Kaplan, S. Liu, and A. Libchaber, "DNA solution of the maximal clique problem," *Science*, vol. 278, pp. 446–449, 1997.
[5] M. Arita, A. Suyama, and M. Hagiya, "A heuristic approach for the Hamiltonian path problem with molecules," in *Proc. 2nd Genetic Programming Conf. (GP '97)*, pp. 457–462.
[6] N. Morimoto, M. Arita, and A. Suyama, "Solid phase DNA solution to the Hamiltonian path problem," in *Proc. 3rd DIMACS Workshop DNA Based Computers*, vol. 48, 1999, pp. 93–101.
[7] A. Narayanan and S. Zorbala *et al.*, "DNA algorithms for computing shortest paths," in *Proc. 3rd Annu. Conf. Genetic Programming 1998*, J. R. Koza *et al.*, Eds., pp. 718–724.
[8] S.-Y. Shin, B.-T. Zhang, and S.-S. Jun, "Solving traveling salesman problems using molecular programming," in *Proc. 1999 Congr. Evolutionary Computation (CEC '99)*, vol. 2, pp. 994–1000.
[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990.
[10] M. R. Garey and D. S. Johnson, *Computer and Intractability*. San Fransico, CA: Freeman, 1979.
[11] D. Boneh, C. Dunworth, R. J. Lipton, and J. Sgall, "On the computational power of DNA," *In Discrete Appl. Math. (Special Issue on Computational Molecular Biology)*, vol. 71, pp. 79–94, 1996.
[12] L. M. Adleman, "On constructing a molecular computer, DNA based computers," in *DNA Based Computers (Selected Papers from Proc. DIMACS Workshop DNA Based Computers'95)*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, R. Lipton and E. Baum, Eds, 1996, pp. 1–21.
[13] M. Amos, "DNA computation," Ph.D. dissertation, Dept. Comput. Sci., Univ. Warwick, Coventry, U.K., 1997.
[14] S. Roweis, E. Winfree, R. Burgoyne, N. V. Chelyapov, M. F. Goodman, P. W. K. Rothemund, and L. M. Adleman, "A sticker based model for DNA computation," in *Proc. 2nd Annu. Workshop DNA Computing*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, L. Landweber and E. Baum, Eds., 1999, pp. 1–29.
[15] M. J. Perez-Jimenez and F. Sancho-Caparrini, "Solving knapsack problems in a sticker based model," in *Proc. 7nd Annu. Workshop DNA Computing*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 2001, pp. 161–171.
[16] G. Paun, G. Rozenberg, and A. Salomaa, *DNA Computing: New Computing Paradigms*. New York: Springer-Verlag, 1998.
[17] W.-L. Chang and M. Guo, "Solving the dominating-set problem in the Adleman–Lipton model," in *Proc. 3rd Int. Conf. Parallel and Distributed Computing, Applications and Technologies*, 2002, pp. 167–172.

[18] ——, “Solving the clique problem and the vertex cover problem in the Adleman–Lipton model,” in *Proc. IASTED Int. Conf. Networks, Parallel and Distributed Processing, and Applications*, 2002, pp. 431–436.

[19] ——, “Solving NP-complete problem in the Adleman–Lipton model,” in *Proc. 2002 Int. Conf. Computer and Information Technology*, pp. 157–162.

[20] ——, “Resolving the 3-dimensional matching problem and the set-packing problem in the Adleman–Lipton model,” in *Proc. IASTED Int. Conf. Networks, Parallel and Distributed Processing, and Applications*, 2002, pp. 455–460.

[21] B. Fu, “Volume bounded molecular computation,” Ph.D. Thesis, Dept. Comput. Sci., Yale Univ., New Haven, CT, 1997.

[22] R. S. Braich, C. Johnson, P. W. K. Rothemund, D. Hwang, N. Chelyapov, and L. M. Adleman, “Solution of a satisfiability problem on a gel-based DNA computer,” in *Proc. 6th Int. Conf. DNA Computation*, 2000, pp. 27–42.

[23] K. Mir, “A restricted genetic alphabet for DNA computing,” in *DNA Based Computers II: Proc. DIMACS Workshop 1996*, vol. 44, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, E. B. Baum and L. F. Landweber, Eds., 1998, pp. 243–246.

[24] A. R. Cukras, D. Faulhammer, R. J. Lipton, and L. F. Landweber, “Chess games: a model for RNA-based computation,” in *In Proc. 4th DIMACS Meeting DNA Based Computers*, Jun. 1998, pp. 27–37.

[25] M. Ho, W.-L. Chang, and M. Guo, “Is cook’s theorem correct for DNA-based computing—toward solving the NP-complete problems on a DNA-based supercomputer model,” *J. Parallel Distrib. Sci. Eng. Comput.*, to be published.

[26] J. H. Reif, T. LaBean, and H. Seeman, “Challenges and applications for self-assembled DNA-nanostructures,” in *Proc. 6th DIMACS Workshop DNA Based Computers*, 2002, pp. 145–172.

[27] T. H. LaBean, E. Winfree, and J. H. Reif, “Experimental progress in computation by self-assembly of DNA tilings,” *Theor. Comput. Sci.*, vol. 54, pp. 123–140, 2000.

[28] S. A. Cook, “The complexity of theorem-proving procedures,” in *Proc. 3rd Annu. ACM Symp. Theory of Computing*, 1971, pp. 151–158.

[29] R. M. Karp, “Reducibility among combinatorial problems,” in *Complexity of Computer Computation*. New York: Plenum, 1972, pp. 85–103.

[30] M. C. LaBean, T. H. Reif, and J. H. Seeman, “Logical computation using algorithmic self-assembly of DNA triple-crossover molecules,” *Nature*, vol. 407, pp. 493–496, 2000.

[31] R. Barua and J. Misra, “Binary arithmetic for DNA computers,” in *Proc. 8th Int. Workshop DNA Based Computers*, 2002, pp. 124–132.

[32] W.-L. Chang, M. Guo, and M. Ho, “Solving the set-splitting problem in sticker-based model and the Adleman–Lipton model,” *Future Gener. Comput. Syst.*, vol. 20, no. 5, pp. 875–885, Jun. 2004.

[33] E. Bach, A. Condon, E. Glaser, and C. Tanguay, “DNA models and algorithms for NP-complete problems,” in *Proc. 11th Annu. Conf. Structure in Complexity Theory*, 1996, pp. 290–299.

[34] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key crytosystem,” *Commun. ACM*, vol. 21, pp. 120–126, 1978.

[35] W.-L. Chang and M. Guo, “Solving the set-cover problem and the problem of exact cover by 3-Sets in the Adleman–Lipton model,” *BioSystems*, vol. 72, no. 3, pp. 263–275, 2003.

[36] W.-L. Chang, M. Ho, and M. Guo, “Molecular solutions for the subset-sum problem on DNA-based supercomputing,” *BioSystems*, vol. 73, no. 2, pp. 117–130, 2004.

[37] R. P. Feynman, *In Minaturization*, D. H. Gilbert, Ed. New York: Reinhold, 1961, pp. 282–296.

[38] F. Guarneiri, M. Fliss, and C. Bancroft, “Making DNA add,” *Science*, vol. 273, pp. 220–223, 1996.

[39] V. Gupta, S. Parthasarathy, and M. J. Zaki, “Arithmetic and logic operations with DNA,” in *Proc. 3rd DIMACS Workshop DNA Based Computers*, 1997, pp. 212–220.

[40] Z. F. Qiu and M. Lu, “Arithmetic and logic Operations with DNA computers,” in *Proc. 2nd IASTED Int. Conf. Parallel and Distributed Computing and Networks*, 1998, pp. 481–486.

[41] M. Ogihara and A. Ray, “Simulating Boolean circuits on a DNA computer,” Univ. Rochester, Rochester, NY, Tech. Rep. TR631, Aug. 1996.

[42] M. Amos and P. E. Dunne, “DNA simulation of Boolean circuits,” University of Liverpool, Liverpool, U.K., Tech. Rep. CTAG-97 009, Dec. 1997.

[43] A. Atanasiu, “Arithmetic with membrames,” in *Proc. Workshop Mutiset Processing*, 2000, pp. 1–17.

[44] P. Frisco, “Parallel arithmetic with splicing,” *Romanian J. Inf. Sci. Technol.*, vol. 3, pp. 113–128, 2000.

[45] H. Hug and R. Schuler, “DNA based parallel computation of simple arithmetic,” in *Proc. 7th Workshop DNA Based Computers*, 2001, pp. 159–166.

[46] R. S. Braich, C. Johnson, P. W. K. Rothemund, D. Hwang, N. Chelyapov, and L. M. Adleman, “Solution of a 20-variable 3-SAT problem on a DNA computer,” *Science*, vol. 296, no. 5567, pp. 499–502, Apr. 2002.

[47] J. Watson, M. Gilman, J. Witkowski, and M. Zoller, *Recombinant DNA*, 2nd ed. San Francisco, CA: Freeman, 1992.

[48] J. Watson, N. Hoplins, and J. Roberts *et al.*, *Molecular Biology of the Gene*. Menlo Park, CA: Benjamin/Cummings, 1987.

[49] G. M. Blackburn and M. J. Gait, *Nucleic Acids in Chemistry and Biology*. Washington, DC: IRL, 1990.

[50] F. Eckstein, *Oligonucleotides and Anologues*. Oxford, U.K.: Oxford Univ. Press, 1991.

[51] D. Boneh, C. Dunworth, and R. J. Lipton, “Breaking DES Using a Molecular Computer,” Princeton Univ., Princeton, NJ, Tech. Rep. CS-TR-489-95, 1995.

[52] L. Adleman, P. W. K. Rothemund, S. Roweis, and E. Winfree, “On applying molecular computation to the data encryption standard,” in *Proc. 2nd Annu. Workshop DNA Computing*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 1999, pp. 31–44.

[53] W.-L. Chang, M. Ho, and M. Guo, “Fast parallel molecular solution to the dominating-set problem on massively parallel bio-computing,” *Parallel Comput.*, vol. 30, no. 9–10, pp. 1109–1125, 2004.

Weng-Long Chang received the Ph.D. degree in computer science and information engineering from National Cheng Kung University in 1999.

He is currently an Assistant Professor with the Southern Taiwan University of Technology, Tainan. His research interests include molecular computing, and languages and compilers for parallel computing.

Minyi Guo (M’00) received the Ph.D. degree in information science from University of Tsukuba in 1998.

From 1998 to 2000, he was a Research Scientist with NEC Soft, Ltd. Japan. From 2001 to 2004, he was a visiting professor of Georgia State University, Hong Kong Polytechnic University, and the University of New South Wales. He is currently a Professor in the Department of Computer Software, University of Aizu, Aizu–Wakamatsu City, Japan. He is the Editor-in-Chief of the *Journal of Embedded Systems*. He is also on the Editorial Board of the *International Journal of High Performance Computing and Networking*, the *Journal of Embedded Computing*, the *Journal of Parallel and Distributed Scientific and Engineering Computing*, and the *International Journal of Computer and Applications*. His research interests include parallel and distributed processing, parallelizing compilers, data parallel languages, data mining, molecular computing, and software engineering. Dr. Guo is a member of the Association for Computing Machinery, the IEEE Computer Society, the Information Processing Society of Japan (IPSJ), and the Institute of Electronics, Information and Communication Engineers (IEICE). He has served as general chair, program committee, or organizing committee chair for many international conferences. He is listed in *Marquis Who’s Who in Science and Engineering*.

Michael Shan-Hui Ho received the M.S. degree from St. Mary’s University and the Ph.D. degree in information science/computer science with a management and accounting minor from the University of Texas, Austin.

He has 25 years industrial and academic experience in the computing field. He has worked as a Senior Software Engineer and Project Leader developing e-business Web and multimedia applications and a Senior Database Administrator for SQL clustered servers, Oracle, and DB2 databases including network/systems LAN/WAN system administration tomajor U.S. corporations and government organizations. He also has more than ten years of college teaching/research experience as an Assistant Professor and Research Analyst at Central Missouri State University, the University of Texas at Austin, and BPC International Institute. He is currently an Associate Professor of Southern Taiwan University of Technology, Tainan. His research interests include algorithm and computation theories, software engineer, database and data mining, parallel computing, quantum computing, and DNA computing.