J Supercomput (2011) 56: 129–163
DOI 10.1007/s11227-009-0347-9

Fast parallel DNA-based algorithms for molecular computation: discrete logarithm

Weng-Long Chang · Shu-Chien Huang · Kawuu Weicheng Lin · Michael (Shan-Hui) Ho

Published online: 11 November 2009

Abstract Diffie and Hellman (IEEE Trans. Inf. Theory 22(6):644–654, 1976) wrote the paper in which the concept of a trapdoor one-way function was first proposed. The Diffie–Hellman 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 Diffie–Hellman public-key cryptosystem is based on the difficulty of solving the problem of discrete logarithms. In this paper, we demonstrate that basic biological operations can be applied to solve the problem of discrete logarithms. In order to achieve this, we propose DNA-based algorithms that formally verify our designed molecular solutions for solving the problem of discrete logarithms. Furthermore, this work indicates that public-key cryptosystems based on the difficulty of solving the problem of discrete logarithms are perhaps insecure.

Keywords Discrete logarithm · The public-key cryptosystems · Cryptography · Security technologies · Molecular cryptography · Biological-based supercomputing · Molecular-based supercomputing · DNA-based supercomputing

W.-L. Chang (✉) · K.W. Lin
Department of Computer Science and Information Engineering, National Kaohsiung University of Applied Sciences, No. 415, Chien Kung Road, Kaohsiung City 807-78, Taiwan, Republic of China
e-mail: changwl@cc.kuas.edu.tw

K.W. Lin
e-mail: linwc@cc.kuas.edu.tw

S.-C. Huang
Department of Computer Science, National PingTung University of Education, No. 4-18 Ming Shen Road, Pingtung 900, Taiwan, Republic of China
e-mail: schuang@mail.npue.edu.tw

M. (S.-H.) Ho
Computer Center and Institute of Electrical Engineering, National Taipei University, 151, University Rd., San Shia 237, Taipei County, Taiwan, Republic of China
e-mail: MHoInCerritos@yahoo.com

1 Introduction

Feynman first proposed molecular computation in 1961, but his idea was not implemented by experiment for a few decades [1]. In 1994, Adleman [2] succeeded to solve an instance of the Hamiltonian path problem in a test tube, just by handling DNA strands. Diffie and Hellman [3] wrote the paper in which the concept of a trapdoor one-way function is proposed. The Diffie–Hellman public-key cryptosystem [3] is a popular cryptosystem and is one of the primary cryptosystems used for security on the Internet and World Wide Web.

DES (the United States Data Encryption Standard) is one of the most widely used cryptographic systems. It produces a 64-bit ciphertext from a 64-bit plaintext under the control of a 56-bit key. A cryptanalyst obtains a plaintext and its corresponding ciphertext and wishes to determine the key used to perform the encryption. The most naive approach to this problem is to try all 2^{56} keys, encrypting the plaintext under each key until a key that produces the ciphertext is found and is called the plaintext-ciphertext attack. Adleman and his co-authors [4] provided a description of such an attack using the *sticker* model of molecular computation. Start with approximately 2^{56} identical ssDNA *memory strands* each 11,580 nucleotides long. Each memory strand contains 579 contiguous blocks each 20 nucleotides long. As it is appropriate in the sticker model, there are 579 *stickers*—one complementary to each block. Memory strands with annealed stickers are called *memory complexes*. When the 2^{56} memory complexes have half of their sticker positions occupied at the end of the computation, they weigh approximately 0.7 g and, in solution at 5 g/liter, would occupy approximately 140 ml. Hence, the volume of the 1303 tubes needs be no more than 140 ml each. It follows that the 1303 tubes occupy, at most, 182 liters and can, for example, be arrayed in 1 m long and wide and 18 cm deep.

Adleman and his co-authors [4] indicated that at the end of computation for breaking DES, $2^{56} \times$ (56 key bits + 64 ciphertext bits) pairs were generated and processed. Adleman and his co-authors [4] also pointed out that this codebook for breaking DES has approximately 2^{63} (8×10^{18}) bits of information (the equivalent of approximately one billion 1 gigabyte CDs). The actual running time for the algorithm of breaking DES depends on how fast the operations can be performed. If each operation requires one day, then the computation for breaking DES will require 18 years. If each operation requires one hour, then the computation for breaking DES will require approximately nine months. If each operation can be completed in one minute, then the computation for breaking DES will take five days. Finally, if the effective duration of a step can be reduced to one second, then the effort for breaking DES will require two hours. While it has been argued that special purpose electronic hardware [4] or massively parallel supercomputers (the IBM Blue Gene/L machine is capable of 183.5 TFLOPS or 183.5×10^{12} floating-point operations per second) might be used to break DES in a reasonable amount of time, it appears that today's most powerful sequential machines would be unable to accomplish the task.

In this paper, we describe novel DNA-based algorithms for a range of binary operations, consisting of bitwise and full comparison, left shifter, addition, subtraction, modular arithmetic, and assignment. We also prove how these smaller modules may be combined to produce an algorithm for solving the problem of discrete logarithm.

The rest of the paper is organized as follows. In Sect. 2, we introduce the development of molecular computing. In Sect. 3, we provide the motivation of writing the article, and the formal model of computation within which the various algorithms are expressed. In Sect. 4, we give a high-level description of our algorithm to solve the problem of discrete logarithms. By means of breaking this down into sub-modules in Sect. 5, we show the operation of the various novel algorithms for comparative and arithmetic operations. In Sect. 6, we propose the attacking method to break the Diffie–Hellman public-key cryptosystem. In Sect. 7, we demonstrate that the time complexity of our algorithm is cubic on the input size. In Sect. 8, we show how the basic operations within our model may be implemented by means of using standard laboratory operations on DNA strands. In Sect. 9, we conclude with a brief discussion.

2 The development of molecular computing

From [5], it was demonstrated that optimal biological molecular solution of every NP-complete or NP-hard problem is determined due to its characteristic. Molecular dynamics and (sequential) membrane systems from the viewpoint of Markov chain theory were proposed in [6]. Reif and LaBean [7] overviewed the past and current state of a selected part of the emerging research area of the field of bio-molecular devices. There are DNA algorithms for solving many famous computational problems, including the 3-SAT problem [14], the binary integer programming problem [15], the dominating-set problem [16], three-vertex-coloring [17], the maximal clique and the set-packing problems [18], the set-splitting problem [19], the set-cover problem and the problem of exact cover by 3-sets [20], subset-production [21], real DNA experiments of Knapsack problems [22], and the set-partition problem [23]. One potentially significant area of application for DNA algorithms is the breaking of encryption schemes [24, 25, 27]. In [28], the design and experimental implementation of DNA-based digital logic circuits were reported, and AND, OR, and NOT gates, signal restoration, amplification, feedback, and cascading were also demonstrated. Kari et al. [29] recalled a list of known properties of DNA languages which are free of certain types of undesirable bonds, and then introduced a general framework in which they can characterize each of these properties by a solution of a uniform formal language inequation.

Wu and Seeman [8] described computation using a DNA strand as the basic unit and they had used this unit to achieve the function of multiplication. From [9], it was reported that a second-generation deoxyribozyme-based automaton, MAYA-II, which plays a complete game of tic-tac-toe according to a perfect strategy, integrating 128 deoxyribozyme-based logic gates, 32 input DNA molecules, and 8 two-channel fluorescent outputs across 8 wells. The first direct observations of tile-based DNA self-assembly in solution using fluorescent nanotubes composed of a single tile was presented from [10]. From [11], it was found that with increasing range of correlations the capacity to distinguish between the species on the basis of this correlation profile is getting better and requires ever shorter sequence segments for obtaining a full species separation. From [12], it was shown that "open" tweezers exist in a single conformation with minimal FRET efficiency. From [13], the first algorithm for calculating the partition function of an unpseudoknotted complex of multiple interacting nucleic acid strands was proposed. In [26], Zhang and Winfree presented an allosteric DNA molecule that, in its active configuration, catalyzes a noncovalent DNA reaction.

In [40], Kershner et al. described the use of electron-beam lithography and dry oxidative etching to create DNA origami-shaped binding sites on technologically useful materials, such as SiO_2 and diamond-like carbon. In buffer with $\sim$100 mM $MgCl_2$, DNA origami binds with high selectivity and good orientation: 70–95% of sites have individual origami aligned with an angular dispersion (± 1 s.d.) as low as $\pm 10°$ (on diamond-like carbon) or $\pm 20°$ (on SiO_2). In [41], Barish et al. presented a programmable DNA origami seed that can display up to 32 distinct binding sites and demonstrate the use of seeds to nucleate three types of algorithmic crystals. In the simplest case, the starting materials are a set of tiles that can form crystalline ribbons of any width; the seed directs assembly of a chosen width with $>$90% yield. Increased structural diversity is obtained by using tiles that copy a binary string from layer to layer; the seed specifies the initial string and triggers growth under near-optimal conditions where the bit copying error rate is $<$0.2%. Increased structural complexity is achieved by using tiles that generate a binary counting pattern; the seed specifies the initial value for the counter. Self-assembly proceeds in a one-pot annealing reaction involving up to 300 DNA strands containing $>$17 kb of sequence information.

3 Motivation and our model

The Diffie–Hellman public-key cryptosystem [3] 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 Diffie–Hellman public-key cryptosystem is based on the difficulty to solve the problem of discrete logarithm. No method in a reasonable amount of time can be applied to solve the problem of discrete logarithm.

In the following subsection, we now describe our formal model of computation, within which we express the various algorithms that are combined to form the overall method for solving the problem of discrete logarithm. We first introduce it only in terms of abstract operations performed on multisets of strings over some alphabet Σ. The presented biological implementation of the model is introduced in Sect. 8. Within our model, a computation starts and ends with zero or more multisets of strings. An algorithm is made of a sequence of operations performed on one or more multisets of strings. We note in passing that this model is sufficiently powerful to solve any problem in the complexity class NP [2, 4, 17, 30, 31].

3.1 Operations

Here we describe the basic legal operations on multisets (henceforth referred to as tubes) from [2, 4, 17]:

1. *Extract*. Given a tube T and a short single strand of DNA, s, the operation produces two new tubes, $+(T, s)$ and $-(T, s)$. Tube $+(T, s)$ is all of the molecules of DNA in T which contain s as a sub-strand and tube $-(T, s)$ is all of the molecules of DNA in T which do not contain s as a sub-strand.
2. *Merge*. Given any n tubes $T_1 \ldots T_n$, the operation yields $\mathit{Merge}(T_1, \ldots, T_n) = \bigcup_{i=1}^{n} T_i = T_1 \cup T_2 \cup \ldots \cup T_n$. This implies that it is to pour any n tubes into one, without any change in the individual strands.
3. *Discard*. Given a tube T, the operation sets T to be an empty set ($T \leftarrow \varnothing$).
4. *Detect*. Given a tube T, the operation returns *true* if T includes at least one DNA molecule ($T \neq \varnothing$), otherwise returns *false*.
5. *Amplify*. Given a tube T, the operation produces a number of identical copies, T_i, of tube T, and then $\mathit{discard}(T)$.
6. $\mathit{Concatenate}(s_1, s_2)$. Given two strands of DNA, s_1 and s_2, the operation returns a new strand of DNA, comprised of the concatenation of s_1 and s_2. If s_1 is a null strand of DNA, return s_2, and if s_2 is a null strand of DNA, return s_1.
7. $\mathit{Append\text{-}head}(T, s)$. Given a non-empty tube T and a short strand of DNA, s, the operation first creates a null tube, U, and then, in parallel, for each string $t_i \in T$ finishes the following: $T \leftarrow \mathit{Merge}(U, \mathit{Concatenate}(s, t_i))$. If T is initially empty, then after the operation is performed, T contains only s.
8. *Read*. Given a tube T, the operation is used to describe a single molecule, which is contained in tube T. Even if T contains many different molecules, each encoding a different set of bases, the operation can give an explicit description of exactly one of them.

3.2 Representation scheme

We now introduce our scheme for the representation of unsigned integers. Because this scheme consists of specific features required by the biological implementation of our algorithms, we denote $\Sigma = \{A, G, C, T\}$. An unsigned integer of k bits, e, is represented as a k-bit binary number, $e_{k-1} \ldots e_0$, where the value of each bit e_j is either 1 or 0 for $0 \leq j \leq k-1$. The bits e_{k-1} and e_0 represent, respectively, the most significant bit and the least significant bit for e. From [30, 31], for each bit e_j, two *distinct* 15 base value sequences over the alphabet Σ are designed. One represents the value "0" for e_j and the other represents the value "1" for e_j. For the sake of convenience in our presentation, assume that e_j^1 denotes the value of e_j to be 1 and e_j^0 defines the value of e_j to be 0.

4 Molecular solutions of discrete logarithms

In Sect. 4.1, we introduce definition of discrete logarithm. In Sect. 4.2, we describe a pseudo algorithm to solve the problem of discrete logarithm. In Sect. 4.3, we propose a DNA-based algorithm to solve the problem of discrete logarithm.

4.1 The introduction of discrete logarithms

For any integer d and any positive integer n, there are unique integers s and r such that $0 \le r < n$ and $d = s * n + r$. The value $s = d/n$ is the *quotient* of the division. The value $r = d$ mod n is the remainder of the division. We have that $n|d$ if and only if d mod $n = 0$. Given a well-defined notion of the remainder one integer when divided by another, it is convenient to provide special notation to indicate equality of remainders. If $(d \bmod n) = (b \bmod n)$, we write $d \equiv b(\text{mod } n)$ and say that d is equivalent to b, modulo n. In other words, $d \equiv b(\text{mod } n)$ if d and b have the same remainder when divided by n. The integer can be divided into n equivalence classes according to their remainders modulo n. The equivalence class modulo n containing an integer d is $[d]_n = \{d + h * n$, where h is an integer$\}$. The set of all such equivalence classes is $\boldsymbol{Z}_n = \{[d]_n : 0 \le d \le n-1\}$. One often sees the definition $\boldsymbol{Z}_n = \{0, 1, \ldots, n-1\}$ [32].

The *greatest common divisor* of two integers d and n, not both zero, is the largest of the common divisors of d and n; it is denoted $\gcd(d, n)$. Two integers d and n are said to be *relatively prime* if their only common divisor is 1, that is, if $\gcd(d, n) = 1$. Because the equivalence class of two integers uniquely determines the equivalence class of their product, thus we define multiplication modulo n, denoted $*_n$, as follows: $[d]_n *_n [h]_n = [d * h]_n$. Using the definition of multiplication modulo n, we define the multiplicative group modulo n as $(\boldsymbol{Z}_n^*, *_n)$, where $\boldsymbol{Z}_n^* = \{[d]_n \in \boldsymbol{Z}_n\colon \gcd(d, n) = 1\}$.

Just as it is natural to consider the multiples of a given element d modulo n, it is often natural to consider the sequence of power of d modulo n, where $d \in \boldsymbol{Z}_n$: $d^0, d^1, d^2, \ldots,$ modulo n. Indexing from 0, its value in this sequence is $d^0 \bmod n = 1$, and the ith value is $d^i \bmod n$. We denote $\langle d \rangle$ as the subgroup of $\boldsymbol{Z}_n^*$ generated by d, and we also denote $\text{ord}_n(d)$ (the "order of d, modulo n") as the order of d in $\boldsymbol{Z}_n^*$. For example, $\langle 2 \rangle = \{1, 2, 4\}$ in $\boldsymbol{Z}_7^*$, and $\text{ord}_7(2) = 3$.

If $\text{ord}_n(M)$ is equal to the number of elements in $\boldsymbol{Z}_n^*$, then every element in $\boldsymbol{Z}_n^*$ is a power of M modulo n, and we say that M is a *primitive root* or a *generator* of $\boldsymbol{Z}_n^*$ [32]. For example, there is a primitive root modulo 7 and $\langle 3 \rangle = \{1, 3, 2, 6, 4, 5\}$. If $\boldsymbol{Z}_n^*$ possesses a primitive root, we say that the group $\boldsymbol{Z}_n^*$ is *cyclic*. If M is a primitive root of $\boldsymbol{Z}_n^*$ and C is any element of $\boldsymbol{Z}_n^*$, then there exists an e such that $M^e \equiv C(\text{mod } n)$. This e is called the discrete logarithm of C modulo n, to the base M. No method in a reasonable amount of time can be applied to solve the problem of discrete logarithm. The following method is used to figure out $M^e \equiv C(\text{mod } n)$ [33].

```
**Procedure Encryption**$(M, e, n)$
(1) Let $e_{k-1} \ldots e_0$ be the binary representation of $e$.
(2) $C = 1$.
(3) **For** $i = k-1$ **down to** 0
    (3a) Set $C$ to the remainder of $(C^2)$ when divided by $n$.
    (3b) If $e_i = 1$ then
    (3c) Set $C$ to the remainder of $(C * M)$ when divided by $n$.
    **EndFor**
(4) Halt. Now $C$ is the result of $M^e(\text{mod } n)$.
**EndProcedure**
```

4.2 The pseudo algorithm for solving discrete logarithms

Assume that the length of e is k bits. Also suppose that e is represented as a k-bit binary number, $e_{k-1} \ldots e_0$, where the value of each bit e_j is either 1 or 0 for $0 \leq j \leq k-1$. The bits e_{k-1} and e_0 represent the most significant bit and the least significant bit for e, respectively. The form of an expression, $M^e(\text{mod } n)$, can be transformed into another form: $(\ldots((1 * M^{e_{k-1}}(\text{mod } n))^2 * M^{e_{k-2}}(\text{mod } n))^2 * M^{e_{k-3}}(\text{mod } n)\ldots)^2 * M^{e_0}(\text{mod } n)$. In the Diffie–Hellman public-key cryptosystem, n is a prime number. Therefore, in this paper, we also assume that n is a prime number. Because n is a prime number, $\langle M \rangle = \{M^0(\text{mod } n), M^1(\text{mod } n) \ldots M^{n-2}(\text{mod } n)\}$. That is to say that $0 \leq e \leq n-2$. The following pseudo algorithm is applied to solve the problem of discrete logarithm.

Method 1 Solving the problem of discrete logarithm.

```
(1) All of the computations for $M^0(\text{mod } n)$, $M^1(\text{mod } n) \ldots M^{n-2}(\text{mod } n)$ are simultaneously performed on a molecular computer.
(2) For any given $C$, from the result finished in Step (1), find $M^e \equiv C(\text{mod } n)$.
(3) Output("discrete logarithm is:", $e$).
```

EndMethod

Proof Step (1) in Method 1 is used to simultaneously complete all of the computations for $M^0(\text{mod } n)$, $M^1(\text{mod } n) \ldots M^{n-2}(\text{mod } n)$. This implies that the value of every element in $\langle M \rangle$ is determined after Step (1) is carried out.

Then, Step (2) in Method 1 is applied to search C among $(n-1)$ elements in $\langle M \rangle$. When the value of the eth element in $\langle M \rangle$ is equal to C, e is the answer (discrete logarithm of C). Finally, Step (3) in Method 1 is employed to describe the answer. □

4.3 The algorithm for computation of discrete logarithms

The procedure, **Encryption**(M, e, n), denoted in Sect. 4.1, is used to finish computation of an exponential modular operation. The following DNA algorithm is applied to implement the procedure, **Encryption**(M, e, n).

Algorithm 1 Implementing the procedure, **Encryption**(M, e, n)

```
(0) $T_0 \leftarrow \varnothing$; $T_\theta \leftarrow \varnothing$; $T_n \leftarrow \varnothing$; $T_1 \leftarrow \varnothing$.
(1) Init($T_0$).
(2) SelectDiscreteLogarithm($T_0$, $T_\theta$).
(3) MakeValue($T_n$).
(4) InitialValue($T_0$).
(5) For $j = k-1$ down to 0
    (5a) ModularMultiplication($T_0$, $T_n$, $(2 * (k-1-j)) * (4 * k + 1) + 1, 2 * (k-j), C, C$).
    (5b) $T_0 = +(T_0, e_j^1)$ and $T_1 = -(T_0, e_j^1)$.
    (5c) ModularMultiplication($T_0$, $T_n$, $(2 * (k-1-j) + 1) * (4 * k + 1) + 1, 2 * (k-j) + 1, C, M$).
    (5d) **For** $r = 0$ **to** $4 * k$
        (5e) **ReservedValue**$(T_1, (2 * (k - 1 - j) + 1) * (4 * k + 1) + r)$.
    **EndFor**
    (5f) **AssignmentOperator**$(T_1, (2 * (k - 1 - j) + 1) * (4 * k + 1) + 1 + 4 * k, 2 * (k - j) + 1)$.
    (5g) $T_0 = \bigcup(T_0, T_1)$.
**EndFor**
**EndAlgorithm**
```

Theorem 1 *From the steps in* Algorithm 1, *the problem of discrete logarithm can be solved.*

Proof From the execution of Step (0), tubes T_0, T_θ, T_n, and T_1 are set to empty tubes. On the execution of Step (1), it calls **Init**(T_0) to construct solution space for 2^k possible discrete logarithms. This means that tube T_0 includes strands encoding 2^k possible discrete logarithms. Next, the execution of Step (2) calls **SelectDiscreteLogarithm**(T_0, T_θ) to perform selection of legal discrete logarithms with its range is from 0 to $n - 2$. This implies that these legal discrete logarithms are encoded in tube T_0. On the execution of Step (3), it calls **MakeValue**(T_n) to encode a prime number, n. This indicates that tube T_n contains a strand encoding it. Next, the execution of Step (4) calls **InitialValue**(T_0) to finish the execution of Step (2) in the procedure, **Encryption**(M, e, n). This is to say that the initial value for C is set to one.

Step (5) is a loop and is mainly used to finish the function of the only loop (Step (3)) in the procedure, **Encryption**(M, e, n). Next, the first execution of Step (5a) calls **ModularMultiplication**$(T_0, T_n, (2 * (k - 1 - j)) * (4 * k + 1) + 1, 2 * (k - j), C, C)$ to perform Step (3a) in **Encryption**(M, e, n). On the first execution of Step (5b), it employs the *extract* operation to form two tubes: T_0 and T_1. The first tube T_0 includes all of the strands that have $e_j = 1$. The second tube T_1 consists of all of the strands that have $e_j = 0$. This indicates that the execution of the step finishes Step (3b) in **Encryption**(M, e, n). Because the jth bit of e encoded in tube T_0 is one, next, the first execution of Step (5c) calls **ModularMultiplication**$(T_0, T_n, (2 * (k - 1 - j) + 1) * (4 * k + 1) + 1, 2 * (k - j) + 1, C, M)$ to perform Step (3c) in **Encryption**(M, e, n). Since the jth bit of e encoded in tube T_1 is zero, Step (5d) is the loop and is mainly used to maintain the consistency of the intermediate value for Y. On the first execution of Step (5e), it calls **ReservedValue**$(T_1, (2 * (k - 1 - j) + 1) * (4 * k + 1) + r)$ to copy the current intermediate value of Y to the next intermediate value of Y. Repeat to execute Step (5e) until the value of r reaches $(4 * k)$. Next, the first execution of Step (5f) calls **AssignmentOperator**$(T_1, (2 * (k - 1 - j) + 1) * (4 * k + 1) + 1 + 4 * k, 2 * (k - j) + 1)$ to perform updating of the value for C. Because the jth bit of e encoded in tube T_1 is zero, the updated value of C is still equal to the previous value.

On the first execution of Step (5g), it uses the *merge* operation to pour tube T_1 into T_0. Repeat execution of Steps (5a) through (5g) until the value of j is zero. After all of the steps are processed, every strand in tube T_0 performs computation of an exponential modular operation, $M^e \pmod n$. This implies that Algorithm 1 performs Step (1) in the pseudo algorithm, Method 1, in Sect. 4.2. Therefore, the problem of discrete logarithm can be solved from those steps in Algorithm 1. □

5 Algorithm modules

We now describe, in detail, the various modules that are combined to form the overall DNA-based algorithm for solving the problem of discrete logarithm.

5.1 Construction of initial solution space for discrete logarithms

From [30, 31], for every bit e_j in discrete logarithm e, two *distinct* 15 base value sequences are designed. One represents the value "0" for e_j and the other represents the value "1" for e_j. For the sake of convenience in our presentation, assume that e_j^1 denotes the value of e_j to be 1 and e_j^0 defines the value of e_j to be 0. We first describe the module **Init**(T_0), which constructs an initial multiset of 2^k binary strings, each representing a possible discrete logarithm. After the initial construction has been completed, we return a tube T_0 containing binary strings encoding the possible discrete logarithm $0 \ldots 2^k - 1$.

```
Procedure Init($T_0$)
(0) $T_1 \leftarrow \varnothing$; $T_2 \leftarrow \varnothing$.
    (0a) Append-head($T_1, e_0^1$).
    (0b) Append-head($T_2, e_0^0$).
    (0c) $T_0 = \bigcup(T_1, T_2)$.
For $j = 1$ to $k - 1$
    (1a) Amplify($T_0, T_1, T_2$).
    (1b) Append-head($T_1, e_j^1$).
    (1c) Append-head($T_2, e_j^0$).
    (1d) $T_0 = \bigcup(T_1, T_2)$.
EndFor
EndProcedure
```

Lemma 1 *Solution space for* 2^k *possible discrete logarithms can be constructed from the algorithm* **Init**(T_0).

Proof The algorithm, **Init**(T_0), is implemented via the *amplify*, *append-head* and *merge* operations. On the execution of Step (0), it sets tubes T_1 and T_2 to empty tubes. Next, from the execution of Step (0a), it is used to append a DNA sequence, representing the value 1 for e_0, onto the head of every strand in tube T_1. This indicates that possible discrete logarithms consisting of the value 1 to the *first* bit appear in tube T_1. Next, on the execution of Step (0b), it is also applied to append a DNA sequence, representing the value 0 for e_0, onto the head of every strand in tube T_2. This is to say that possible discrete logarithms including the value 0 to the *first* bit appear in tube T_2. Next, Step (0c) is used to pour tube T_1 and T_2 into tube T_0. This implies that DNA strands in tube T_0 contain DNA sequences of $e_0 = 1$ and $e_0 = 0$, tube $T_1 = \varnothing$, and tube $T_2 = \varnothing$.

Each time 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 becomes empty. Then, Step (1b) is applied to append a DNA sequence, representing the value 1 for e_j, onto the head of every strand in tube T_1. This means that possible discrete logarithms containing the value 1 to the $(j+1)$th bit appear in tube T_1. Step (1c) is also employed to append a DNA sequence, representing the value 0 for e_j, onto the head of every strand in tube T_2. That implies that possible discrete logarithms containing the value 0 to the $(j+1)$th bit appear in tube T_2. Next, Step (1d) is used to pour tube T_1 and T_2 into tube T_0. This indicates that DNA strands in tube T_0 include DNA sequences of $e_j = 1$ and $e_j = 0$. After repeating execution of Steps (1a) to (1d), it finally produces tube T_0 that consists of 2^k DNA sequences representing 2^k possible discrete logarithms, tube $T_1 = \varnothing$, and tube $T_2 = \varnothing$. Therefore, it is inferred that solution space for 2^k possible discrete logarithms can be constructed. □

5.2 Solution space for $\text{Order}_n(M)$

Because $\text{Order}_n(M)$ is equal to $n-1$, suppose that $n-1$ is represented as a k-bit binary number, $\theta_{k-1}\ldots\theta_0$, where the value of each bit θ_j is either 1 or 0 for $0 \le j \le k-1$. The bits θ_{k-1} and θ_0 are used to represent the most significant bit and the least significant bit for $n-1$, respectively. From [30, 31], for every bit θ_j, two *distinct* 15 base value sequences are designed. One represents the value "0" for θ_j and the other represents the value "1" for θ_j. For the sake of convenience in our presentation, assume that θ_j^1 denotes the value of θ_j to be 1 and θ_j^0 defines the value of θ_j to be 0. The following algorithm, **SelectDiscreteLogarithm**(T_0, T_θ), is proposed to construct a DNA strand for encoding $n-1$ and select legal discrete logarithms.

```
Procedure SelectDiscreteLogarithm($T_0$, $T_\theta$)
(1) For $j = 0$ to $k-1$
    (1a) Append-head($T_\theta, \theta_j$).
    EndFor
(2) For $j = k-1$ down to 0
    (2a) $T_0^{ON} = +(T_0, e_j^1)$ and $T_0^{OFF} = -(T_0, e_j^1)$.
    (2b) $T_\theta^{ON} = +(T_\theta, \theta_j^1)$ and $T_\theta^{OFF} = -(T_\theta, \theta_j^1)$.
    (2c) If (Detect($T_\theta^{ON}$) == true) then
            (2d) $T_0^{=} = \bigcup(T_0^{=}, T_0^{ON})$ and $T_0^{<} = \bigcup(T_0^{<}, T_0^{OFF})$.
         Else
            (2e) $T_0^{>} = \bigcup(T_0^{>}, T_0^{ON})$ and $T_0^{=} = \bigcup(T_0^{=}, T_0^{OFF})$.
         EndIf
    (2f) $T_\theta = \bigcup(T_\theta^{ON}, T_\theta^{OFF})$.
    (2g) Discard($T_0^{>}$).
    (2h) $T_0 = \bigcup(T_0, T_0^{=})$.
    EndFor
(3) Discard($T_0$).
(4) $T_0 = \bigcup(T_0, T_0^{<})$.
EndProcedure
```

Lemma 2 *The algorithm*, **SelectDiscreteLogarithm**(T_0, T_θ), *can be applied to encode* $n-1$ *and perform selection of legal discrete logarithms*, *with its range is from* 0 *to* $n-2$, *from solution space.*

Proof The algorithm, **SelectDiscreteLogarithm**(T_0, T_θ), is implemented via the *append-head*, *extract*, *detect*, *merge* and *discard* operations. The first loop in the algorithm is mainly used to construct a DNA strand for $n-1$. Each time Step (1a) is used, it appends a DNA sequence, encoding the value "1" or "0" of θ_j, onto the head of every strand in tube T_θ. After repeating execution of Step (1a), it finally produces tube T_θ that includes a DNA strand encoding $n-1$. Therefore, it is inferred that solution space for $n-1$ can be constructed.

The second loop is mainly used to finish selection of legal discrete logarithms. Each execution of Step (2a) employs the *extract* operation to form two test tubes: T_0^{ON} and T_0^{OFF}. The first tube T_0^{ON} includes all of the strands that have $e_j = 1$. The second tube T_0^{OFF} consists of all of the strands that have $e_j = 0$. On each execution of Step (2b), it uses the *extract* operation to form two test tubes: T_θ^{ON} and T_θ^{OFF}. The first tube T_θ^{ON} includes all of the strands that have $\theta_j = 1$. The second tube T_θ^{OFF} consists of all of the strands that have $\theta_j = 0$. Next, each execution of Step (2c) uses the *detect* operation to check whether there is any DNA sequence in tube T_θ^{ON}. If it returns a *true*, this indicates that the value of the jth bit in $n-1$ is one. On each execution of Step (2d), it uses the *merge* operations to pour T_0^{ON} into $T_0^{=}$ and also to pour T_0^{OFF} into $T_0^{<}$. If the *detect* operation in Step (2c) returns a *false*, this indicates that the value of the jth bit in $n-1$ is zero. Hence, next, each execution of Step (2e) applies the *merge* operations to pour T_0^{ON} into $T_0^{>}$ and also to pour T_0^{OFF} into $T_0^{=}$. On each execution of Step (2f), it applies the *merge* operations to pour T_θ^{ON} and T_θ^{OFF} into T_θ. Then, because the encoded value of DNA strands in tube $T_0^{>}$ is great than $n-1$, each execution of Step (2g) employs the *discard* operation to discard $T_0^{>}$. On each execution of Step (2h), it applies the *merge* operations to pour $T_0^{=}$ into T_0. After repeating execution of Steps (2a) to (2h), it finally produces tubes T_0 and $T_0^{<}$. Tube T_0 contains the encoded value of DNA strands to be equal to $n-1$. Tube $T_0^{<}$ includes the encoded value of DNA strands to be less than $n-1$. Next, on the execution of Step (3), it applies the *discard* operation to discard T_0 that contains the encoded value of a DNA strand to be equal to $n-1$. Finally, the execution of Step (4) uses the *merge* operations to pour $T_0^{<}$ into T_0. This indicates that DNA strands encoding legal discrete logarithms are reserved in T_0. Therefore, it is inferred that selection of legal discrete logarithms with its range is from 0 to $n-2$ can be performed. □

5.3 Solution space for module n

Assume that the length of n denoted in Sect. 3.1 is k bits. Also suppose that n is represented as a k-bit binary number, $n_{k-1} \ldots n_0$, where the value of each bit n_j is either 1 or 0 for $0 \le j \le k-1$. The bits n_{k-1} and n_0 represent the most significant bit and the least significant bit for n, respectively. From [30, 31], for every bit n_j, two *distinct* 15 base value sequences are designed. One represents the value "0" for n_j and the other represents the value "1" for n_j. For the sake of convenience in our presentation, assume that n_j^1 denotes the value of n_j to be 1 and n_j^0 defines the value of n_j to be 0. The following algorithm, **MakeValue**(T_n), is proposed to construct a DNA strand for encoding n.

```
Procedure MakeValue($T_n$)
For $j = 0$ to $k - 1$
  (1a) Append-head($T_n, n_j$).
EndFor
EndProcedure
```

Lemma 3 *Solution space of n can be constructed from the algorithm*, **MakeValue**(T_n).

Proof Similar to Lemmas 1 and 2. □

5.4 Solution space for a primitive root M and the result of an exponential modular operation C

Suppose that the length of a primitive root M for Z_n^* is k bits. Also assume that M is represented as a k-bit binary number, $m_{k-1} \ldots m_0$, where the value of each bit m_j is either 1 or 0 for $0 \le j \le k - 1$. The bits m_{k-1} and m_0 represent the most significant bit and the least significant bit for M, respectively. From [30, 31], for every bit m_j, two *distinct* 15 base value sequences are designed. One represents the value "0" for m_j and the other represents the value "1" for m_j. For the sake of convenience in our presentation, assume that m_j^1 denotes the value of m_j to be 1 and m_j^0 defines the value of m_j to be 0.

Assume that the length of C, the result of an exponential modular operation denoted in Sect. 4.1, is k bits. From the procedure Encryption(M, e, n), C is finally obtained after at most updating $(2 * k + 1)$ times of the value for C. Therefore, suppose that C is represented as a k-bit binary number, $c_{a,k-1} \ldots c_{a,0}$, where the value of each bit $c_{a,j}$ is either 1 or 0 for $1 \le a \le (2 * k + 1)$ and $0 \le j \le k - 1$. The bits, $c_{a,k-1}$ and $c_{a,0}$, represent the most significant bit and the least significant bit for C, respectively. The first k-bit binary number, $c_{1,k-1} \ldots c_{1,0}$, is used to represent the initial value to C. The last k-bit binary number, $c_{(2*k+1),k-1} \ldots c_{(2*k+1),0}$, is used to represent the final result of C. For other k-bit binary numbers, they are applied to represent the intermediate computed form of C. From [30, 31], for every bit $c_{a,j}$, two *distinct* 15 base value sequences were designed. One represents the value "0" for $c_{a,j}$ and the other represents the value "1" for $c_{a,j}$. For the sake of convenience in our presentation, assume that $c_{a,j}^1$ denotes the value of $c_{a,j}$ to be 1 and $c_{a,j}^0$ defines the value of $c_{a,j}$ to be 0. The following algorithm is used to construct solution space for the initial value for C and the primitive root M.

```
Procedure InitialValue($T_0$)
(1) For $j = 0$ to $k - 1$
      (1a) Append-head($T_0, m_j$).
    EndFor
(2) Append-head($T_0, c_{1,0}^1$).
(3) For $j = 1$ to $k - 1$
```

```
    (3a) Append-head($T_0$, $c^0_{1,j}$).
EndFor
EndProcedure
```

Lemma 4 *Solution space for the initial value of C and the primitive root M can be constructed from the algorithm,* **InitialValue**(T_0).

Proof Similar to Lemmas 1 and 2. □

5.5 The algorithm for computation of a modular multiplication

The procedure, **Encryption**(M, e, n), denoted in Sect. 4.1, is used to finish computation of an exponential modular operation. In the procedure, it uses successive operations of square and multiplication to perform the exponential modular operation. We now give details of the **ModularMultiplication**$(T_0, T_n, f, a, \alpha, \beta)$ module used by the main algorithm. The following DNA-based algorithm, **ModularMultiplication**$(T_0, T_n, f, a, \alpha, \beta)$, is applied to perform all of the steps to a modular multiplication. This implies that Steps (3a) and (3c) in the procedure, **Encryption**(M, e, n), are performed through the following DNA-based algorithm, **ModularMultiplication**$(T_0, T_n, f, a, \alpha, \beta)$. The two parameters, α and β, in **ModularMultiplication**$(T_0, T_n, f, a, \alpha, \beta)$ represent the multiplicand and the multiplier of a modular multiplication. Assume that β^1_j is applied to represent the value of "1" for the jth bit of the multiplier (β).

Procedure ModularMultiplication$(T_0, T_n, f, a, \alpha, \beta)$

```
(1) InitialSet($T_0$, $f$).
(2) For $j = k - 1$ down to 0
    (2a) ParallelLeftShifter($T_0$, $f + (k - 1 - j) * 4$).
    (2b) ParallelComparator($T_0$, $T_n$, $T_0^>$, $T_0^=$, $T_0^<$, $f + (k - 1 - j) * 4 + 1$).
    (2c) $T_0 = \bigcup(T_0^>, T_0^=)$.
    (2d) BinaryParallelSubtractor($T_0$, $f + (k - 1 - j) * 4 + 1$).
    (2e) ReservedValue($T_0^<$, $f + (k - 1 - j) * 4 + 1$).
    (2f) $T_0 = \bigcup(T_0, T_0^<)$.
    (2g) $T_0 = +(T_0, \beta^1_j)$ and $T_1 = -(T_0, \beta^1_j)$.
    (2h) If (Detect($T_0$) == $true$) then
        (2i) BinaryParallelAdder($T_0$, $f + (k - 1 - j) * 4 + 2, a$).
        (2j) ParallelComparator($T_0$, $T_n$, $T_0^>$, $T_0^=$, $T_0^<$, $f + (k - 1 - j) * 4 + 3$).
        (2k) $T_0 = \bigcup(T_0^>, T_0^=)$.
        (2l) BinaryParallelSubtractor($T_0$, $f + (k - 1 - j) * 4 + 3$).
        (2m) ReservedValue($T_0^<$, $f + (k - 1 - j) * 4 + 3$).
        (2n) $T_0 = \bigcup(T_0, T_0^<)$.
    EndIf
    (2o) If (Detect($T_1$) == $true$) then
        (2p) ReservedValue($T_1$, $f + (k - 1 - j) * 4 + 2$).
        (2q) ReservedValue($T_1$, $f + (k - 1 - j) * 4 + 3$).
```

```
            **EndIf**
    (2r) $T_0 = \bigcup(T_0, T_1)$.
    **EndFor**
    (2s) **AssignmentOperator**$(T_0, f + k * 4, a)$.
**EndProcedure**.
```

Lemma 5 *The algorithm,* **ModularMultiplication**$(T_0, T_n, f, a, \alpha, \beta)$, *can be used to finish computation of a modular multiplication.*

Proof On the execution of Step (1), it calls **InitialSet**(T_0, f) to set the initial value to zero for computation of a modular multiplication in solution space. This means that the strands encoding the initial value are appended onto the head of every strand in tube T_0. Step (2) is a loop and is mainly used to finish the function of Steps (3a) and (3c) in the procedure, **Encryption**(M, e, n). Next, the first execution of Step (2a) calls **ParallelLeftShifter**$(T_0, f + (k - 1 - j) * 4)$ to perform left shift of one time to the initial value. This indicates that the new (intermediate) value is equal to twice of the initial value. On the first execution of Step (2b), it calls **ParallelComparator**$(T_0, T_n, T_0^>, T_0^=, T_0^<, f + (k - 1 - j) * 4 + 1)$ to compare the new value with the value of n. After it is performed, 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 (“<”).

Next, the first execution of Step (2c) applies the *merge* operation to pour tubes $T_0^>$ and $T_0^=$ into T_0. This implies that tube T_0 contains with the comparative result of greater than (“>”) or equal (“=”). On the first execution of Step (2d), it calls **BinaryParallelSubtractor**$(T_0, f + (k - 1 - j) * 4 + 1)$ to finish the function of one subtraction. This is to say that every intermediate value encoded in tube T_0 is subtracted from the value of n. Because every intermediate value encoded in tube $T_0^<$ is less than the value of n, the first execution of Step (2e) calls **ReservedValue**$(T_0^<, f + (k - 1 - j) * 4 + 1)$ to reserve the values encoded in $T_0^<$. Then, on the first execution of Step (2f), it uses the *merge* operation to pour tube $T_0^<$ into T_0. The first execution of Step (2g) employs the *extract* operation to form two test tubes: T_0 and T_1. The first tube T_0 includes all of the strands that have $\beta_j = 1$, and the second tube T_1 consists of all of the strands that have $\beta_j = 0$.

Next, the first execution of Step (2h) employs the *detect* operation to check whether there is any DNA sequence in tube T_0. If it returns a *true*, this indicates that the value of the jth bit in C or M (they are multipliers) is one. On the first execution of Step (2i), it calls **BinaryParallelAdder**$(T_0, f + (k - 1 - j) * 4 + 2, a)$ to perform the function of one addition. This means that the value of C (it is also a multiplicand) is added to every intermediate value encoded in tube T_0. Then, the first execution of Step (2j) calls **ParallelComparator**$(T_0, T_n, T_0^>, T_0^=, T_0^<, f + (k - 1 - j) * 4 + 3)$ to compare the intermediate value with the value of n. After it is performed, 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 (“<”).

On the first execution of Step (2k), it uses the *merge* operation to pour tubes $T_0^>$ and $T_0^=$ into T_0. This implies that tube T_0 contains with the comparative result of greater than ("$>$") or equal ("$=$"). Then, the first execution of Step (2l), it calls **BinaryParallelSubtractor**$(T_0, f + (k - 1 - j) * 4 + 3)$ to finish the function of one subtraction. This is to say that every intermediate value encoded in tube T_0 is subtracted from the value of n. Because every intermediate value encoded in tube $T_0^<$ is less than the value of n, next, the first execution of Step (2m) calls **ReservedValue**$(T_0^<, f + (k - 1 - j) * 4 + 3)$ to reserve the values encoded in $T_0^<$. On the first execution of Step (2n), it uses the *merge* operation to pour tube $T_0^<$ into T_0.

Next, the first execution of Step (2o) applies the *detect* operation to check whether there is any DNA sequence in tube T_1. If it returns a *true*, this indicates that the value of the jth bit in C or M (they are multipliers) is zero. Therefore, for the consistency of encoding every intermediate value in tubes T_0 and T_1, on the first execution of Step (2p), it calls **ReservedValue**$(T_1, f + (k - 1 - j) * 4 + 2)$ to reserve the intermediate values encoded in T_1. Then, similarly, the first execution of Step (2q), it also calls **ReservedValue**$(T_1, f + (k - 1 - j) * 4 + 3)$ to reserve the intermediate values encoded in T_1.

On the first execution of Step (2r), it uses the *merge* operation to pour tube T_1 into T_0. Repeat execution of Steps (2a) through (2r) until the value of j is zero. After all of the steps in the only loop are processed, every strand in tube T_0 performs computation of a modular multiplication, $(C * C) \bmod n$ or $(C * M) \bmod n$. Finally, the execution of Step (2s) calls **AssignmentOperator**$(T_0, f + k * 4, a)$ to perform the ath updating of the value for C. □

5.6 Solution space for the initial value for computation of a modular multiplication

For any given two positive integers d and b, Blakley [34] proposed the fastest method to perform computation of $(d * b) \pmod n$. Blakley used adder and subtractor of $(4 * k)$ times to perform computation of $(d * b) \pmod n$. Assume that $Y \equiv (d * b) \pmod n$ and the length of Y is k bits. From Blakley's method, Y is finally obtained after at most updating $(4 * k + 1)$ times of the value for Y. From the procedure **Encryption**(M, e, n), Blakley's method is at most called $(2 * k)$ times. That is to say, at most updating $(8 * k^2 + 2 * k)$ times of the value for Y are completed. Therefore, suppose that Y is represented as a k-bit binary number, $y_{f,k-1} \ldots y_{f,0}$, where the value of each bit $y_{f,g}$ is either 1 or 0 for $1 \le f \le (8 * k^2 + 2 * k)$ and $0 \le g \le k - 1$. The bits, $y_{f,k-1}$ and $y_{f,0}$, represent the most significant bit and the least significant bit for Y, respectively. If updating of fth time for Y is finished through an adder, then two binary numbers $y_{f,k-1} \ldots y_{f,0}$ and $y_{f+1,k-1} \ldots y_{f+1,0}$ represent the augend and the sum of the fth updating, respectively. If updating of fth time for Y is finished through a subtractor, then two binary numbers $y_{f,k-1} \ldots y_{f,0}$ and $y_{f+1,k-1} \ldots y_{f+1,0}$ represent the minuend and the difference of the fth updating, respectively.

From [30, 31], for every bit $y_{f,g}$, two *distinct* 15 base value sequences were designed. One represents the value "0" for $y_{f,g}$ and the other represents the value "1" for $y_{f,g}$. For the sake of convenience in our presentation, assume that $y_{f,g}^1$ denotes the value of $y_{f,g}$ to be 1 and $y_{f,g}^0$ defines the value of $y_{f,g}$ to be 0. The following algorithm is used to construct solution space for the initial value to computation of a modular multiplication.

```
Procedure InitialSet($T_0$, $f$)
(1) For $g = 0$ to $k - 1$
        (1a) Append-head($T_0$, $y^0_{f,g}$).
    EndFor
EndProcedure
```

Lemma 6 *Solution space for the initial value to computation of a modular multiplication can be constructed from the algorithm,* **InitialSet**(T_0, f).

Proof Similar to Lemmas 1 and 2. □

5.7 The construction of a left shifter

The **ModularMultiplication**$(T_0, T_n, f, a, \alpha, \beta)$ module uses, as a sub-module, a parallel left shifter. We now describe its construction in detail. A left shifter is an instruction of two operands of k bits that the second operand is applied to represent the number of the left shift to the first operand. Suppose that the one-bit binary numbers $y_{f,g}$ denoted in Sect. 5.6, represent the first operand of a left shifter for $1 \leq f \leq (8 * k^2 + 2 * k)$ and $0 \leq g \leq k - 1$. Because computation of $(d * b)(\text{mod } n)$ denoted in Sect. 5.6 cited from [34] only needs to perform left shift of one time, the second operand actually is equal to one. The following algorithm is used to construct a parallel left shifter.

```
Procedure ParallelLeftShifter($T_0$, $f$)
(1) Append-head($T_0$, $y^0_{f+1,0}$).
(2) For $j = 0$ to $k - 2$
    (2a) $T_1 = +(T_0, y^1_{f,j})$ and $T_2 = -(T_0, y^1_{f,j})$.
    (2b) Append-head($T_1$, $y^1_{f+1,j+1}$).
    (2c) Append-head($T_2$, $y^0_{f+1,j+1}$).
    (2d) $T_0 = \bigcup(T_1, T_2)$.
    EndFor
EndProcedure
```

Lemma 7 *The algorithm,* **ParallelLeftShifter**(T_0, f), *can be applied to finish the function of a parallel left shifter.*

Proof On the execution of Step (1), it uses the *append-head* operations to append $y^0_{f+1,0}$ onto the head of every strand in T_0. This is to say that the least significant position of every value encoded is filled out zero. Step (2) is a loop and is employed to finish the main operations of a parallel left shifter for tube T_0. Each execution of Step (2a), 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 $y_{f,j} = 1$, and the second tube T_2 consists of all of the strands that have $y_{f,j} = 0$. Next, with each execution of Step (2b) it uses the *append-head* operations to append $y^1_{f+1,j+1}$ onto the head of every strand in T_1. On each execution of Step (2c) applies the *append-head* operations to append $y^0_{f+1,j+1}$ onto the head of every strand in T_2. Then, each execution of Step (2d) applies the *merge* operation to pour tubes T_1 and T_2 into T_0. Tube T_0 contains the strands finishing left shift of a bit. Repeat execution of Steps (2a) through (2d) until the k bits are processed. Tube T_0 contains every strand encoding the corresponding value that is twice of the original encoded value. This indicates that tube T_0 contains the strands finishing left shift of one time. □

5.8 The construction of a parallel comparator

The **ModularMultiplication**$(T_0, T_n, f, a, \alpha, \beta)$ module uses, as a sub-module, a parallel comparator. We now describe its construction in detail. A modular multiplication denoted in Sect. 5.6 cited from [34] is carried out by means of successive shift, compared, to addition, and subtract operations. This indicates that shift, and compared operations must be finished before the corresponding addition and subtraction are done. A one-bit parallel comparator is a Boolean function that performs compared operation of two input bits. The first input bit and the second input bit, respectively, represent bits of Y and n to be compared. From compared results in a one-bit parallel comparator, DNA strands encoding those pairs (Y, n) with compared results "$>$", DNA strands encoding those pairs (Y, n) with compared results "$=$" and DNA strands encoding those pairs (Y, n) with compared results "$<$" are, respectively, put into three different tubes.

Therefore, the sub-module, **OneBitComparator**$(T_0, T_n, T_0^{>}, T_0^{=}, T_0^{<}, f, g, j)$ is presented to compute the function of a one-bit parallel comparator. The first parameter, T_0, consists of those DNA strands encoding each possible value of Y. The second parameter, T_n, contains the only DNA strand encoding the value of n. The third parameter, $T_0^{>}$, includes those DNA strands with the comparative result of greater than ("$>$") between Y and n. The fourth parameter, $T_0^{=}$, contains those DNA strands with the comparative result of equal ("$=$") between Y and n. The fifth parameter, $T_0^{<}$, consists of those DNA strands with the comparative result of less than ("$<$") between Y and n. The sixth parameter, f, is used to represent the fth comparison between Y and n. The seventh parameter, g, is applied to represent the gth bit of Y in the fth comparison. The eighth parameter, j, is employed to represent the jth bit of n in the fth comparison. The module, **ParallelComparator**$(T_0, T_n, T_0^{>}, T_0^{=}, T_0^{<}, f)$, is proposed to compute the function of a k-bit parallel comparator.

Procedure OneBitComparator$(T_0, T_n, T_0^{>}, T_0^{=}, T_0^{<}, f, g, j)$

```
(1) $T_0^{ON} = +(T_0, y^1_{f,g})$ and $T_0^{OFF} = -(T_0, y^1_{f,g})$.
(2) $T_n^{ON} = +(T_n, n^1_j)$ and $T_n^{OFF} = -(T_n, n^1_j)$.
(3) If (Detect($T_n^{ON}$) == true) then
        (3a) $T_0^{=} = \bigcup(T_0^{=}, T_0^{ON})$ and $T_0^{<} = \bigcup(T_0^{<}, T_0^{OFF})$.
    Else
        (3b) $T_0^{>} = \bigcup(T_0^{>}, T_0^{ON})$ and $T_0^{=} = \bigcup(T_0^{=}, T_0^{OFF})$.
    EndIf
```

```
(4) $T_n = \bigcup(T_n^{\mathrm{ON}}, T_n^{\mathrm{OFF}})$.
EndProcedure
```

Lemma 8 *The algorithm,* **OneBitComparator**$(T_0, T_n, T_0^{>}, T_0^{=}, T_0^{<}, f, g, j)$, *can be applied to finish the function of a one-bit parallel comparator.*

Proof The algorithm, **OneBitComparator**$(T_0, T_n, T_0^{>}, T_0^{=}, T_0^{<}, f, g, j)$, is implemented via the *extract*, *detect* and *merge* operations. The execution of Step (1) employs the *extract* operation to form two test tubes: T_0^{ON} and T_0^{OFF}. The first tube T_0^{ON} includes all of the strands that have $y_{f,g} = 1$. The second tube T_0^{OFF} consists of all of the strands that have $y_{f,g} = 0$. The execution of Step (2) also uses the *extract* operation to form two test tubes: T_n^{ON} and T_n^{OFF}. The first tube T_n^{ON} includes all of the strands that have $n_j = 1$. The second tube T_n^{OFF} consists of all of the strands that have $n_j = 0$. Next, the execution of Step (3) uses the *detect* operation to check whether there is any DNA sequence in tube T_n^{ON}. If it returns *true*, this indicates that the value of the jth bit in n is one. On the execution of Step (3a), it uses the *merge* operations to pour T_0^{ON} into $T_0^{=}$ and also to pour T_0^{OFF} into $T_0^{<}$. If the *detect* operation in Step (3) returns a *false*, this implies that the value of the jth bit in n is zero. Next, the execution of Step (3b) applies the *merge* operations to pour T_0^{ON} into $T_0^{>}$ and also to pour T_0^{OFF} into $T_0^{=}$. Finally, the execution of Step (4) employs the *merge* operations to pour T_n^{ON} and T_n^{OFF} into T_n. This indicates that the DNA strand encoding n is reserved in T_n and will be used for comparator of the next bit. □

```
Procedure ParallelComparator($T_0, T_n, T_0^{>}, T_0^{=}, T_0^{<}, f$)
    (1) For $j = k - 1$ down to 0
        (1a) OneBitComparator($T_0, T_n, T_0^{>}, T_0^{=}, T_0^{<}, f, j, j$).
        (1b) If (Detect($T_0^{=}$) == false) then
                (1c) Terminate the execution of the loop.
             EndIf
        EndFor
EndProcedure
```

Lemma 9 *The algorithm,* **ParallelComparator**$(T_0, T_n, T_0^{>}, T_0^{=}, T_0^{<}, f)$ *can be used to finish the function of a k-bit parallel comparator.*

Proof The only loop is used to implement the function of a k-bit parallel comparator. The first execution of Step (1a) calls the algorithm, **OneBitComparator**$(T_0, T_n, T_0^{>}, T_0^{=}, T_0^{<}, f, j, j)$, to finish the comparative result of the corresponding bit for Y and n. On the first execution of Step (1b), it uses the *detect* operations to check whether there is any DNA sequence in $T_0^{=}$. If it returns a *false*, then the execution of Step (1c) will terminate the execution of the loop. Otherwise it repeats execution of Steps (1a) through (1c) until the corresponding bits are all processed. Therefore, 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 ("$=$"). Tube $T_0^{<}$ consists of the strands with the comparative result of less than ("$<$"). □

5.9 The construction of a binary parallel subtractor

The **ModularMultiplication**$(T_0, T_n, f, a, \alpha, \beta)$ module uses, as a sub-module, a parallel subtractor. The one-bit subtractor (later introduced in Sect. 5.10) figures out the difference bit and the borrow bit for two input bits and a previous borrow. Two k-bit binary numbers can finish subtractions of k times by means of this one-bit subtractor. A binary parallel subtractor is a Boolean function that finishes the arithmetic subtraction for two k-bit binary numbers. The following algorithm is proposed to finish the Boolean function of a binary parallel subtractor.

```
Procedure BinaryParallelSubtractor($T_0$, $f$)
(1) Append − head($T_0$, $b^0_{f,-1}$).
(2) For $j = 0$ to $k - 1$
        (2a) ParallelOneBitSubtractor($T_0$, $f$, $j$, $j$).
    EndFor
EndProcedure
```

Lemma 10 *The algorithm*, **BinaryParallelSubtractor**(T_0, f), *can be applied to finish the Boolean function of a binary parallel subtractor.*

Proof When the first subtract operation occurs, the least significant position for a minuend of k bits and a subtrahend of k bits is subtracted, the input borrow bit must be 0. So, Step (1) uses the *append-head* operation to append 15-based DNA sequences for representing $b^0_{f,-1}$ onto the head of every strand in T_0. Step (2) is the only loop and is mainly used to finish the Boolean function of a binary parallel subtractor. On the first execution of Step (2a), it calls the procedure, **ParallelOneBit-Subtractor**(T_0, f, j, j), to compute the arithmetic subtraction of the least significant bit to the minuend and the subtrahend of k bits. Repeat execution of Step (2a) until the most significant bit in the minuend and the subtrahend of k bits is processed. Tube T_0 contains the strands finishing the arithmetic subtraction of k bits. □

5.10 The construction of a parallel one-bit subtractor

A one-bit subtractor is a Boolean function that forms the arithmetic subtraction of three input bits. It consists of three inputs and two outputs. Two of the input bits 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 one-bit subtractor is as follows:

Suppose that the two one-bit binary numbers $y_{f,g}$ and $y_{f+1,g}$ denoted in Sect. 5.6, represent the first input and the first output of a one-bit subtractor for $1 \le f \le (8 * k^2 + 2 * k)$ and $0 \le g \le k - 1$. Also suppose a one-bit binary number n_j denoted in Sect. 5.3, represents the second input of a one-bit subtractor for $0 \le j \le k - 1$, and two one-bit binary numbers $b_{f,g}$ and $b_{f,g-1}$ represent the second output and the third input of a one-bit subtractor.

Table 1 The truth table of a one-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

Two *distinct* DNA sequences are designed to encode every bit $b_{f,g-1}$ and $b_{f,g}$. For the sake of convenience in our presentation, assume that $b^1_{f,g}$ contains the value of $b_{f,g}$ to be 1 and $b^0_{f,g}$ contains the value of $b_{f,g}$ to be 0. Similarly, also suppose that $b^1_{f,g-1}$ contains the value of $b_{f,g-1}$ to be 1 and $b^0_{f,g-1}$ contains the value of $b_{f,g-1}$ to be 0. The following algorithm is proposed to finish the Boolean function of a parallel one-bit subtractor.

```
Procedure ParallelOneBitSubtractor($T_0$, $f$, $g$, $j$)
   (1) $T_1 = +(T_0, y^1_{f,g})$ and $T_2 = -(T_0, y^1_{f,g})$.
   (2) $T_3 = +(T_1, n^1_j)$ and $T_4 = -(T_1, n^1_j)$.
   (3) $T_5 = +(T_2, n^1_j)$ and $T_6 = -(T_2, n^1_j)$.
   (4) $T_7 = +(T_3, b^1_{f,g-1})$ and $T_8 = -(T_3, b^1_{f,g-1})$.
   (5) $T_9 = +(T_4, b^1_{f,g-1})$ and $T_{10} = -(T_4, b^1_{f,g-1})$.
   (6) $T_{11} = +(T_5, b^1_{f,g-1})$ and $T_{12} = -(T_5, b^1_{f,g-1})$.
   (7) $T_{13} = +(T_6, b^1_{f,g-1})$ and $T_{14} = -(T_6, b^1_{f,g-1})$.
  (8a) If (Detect($T_7$) == true) then
           (8) Append-head($T_7$, $y^1_{f+1,g}$) and Append-head($T_7$, $b^1_{f,g}$).
       EndIf
  (9a) If (Detect($T_8$) == true) then
           (9) Append-head($T_8$, $y^0_{f+1,g}$) and Append-head($T_8$, $b^0_{f,g}$).
       EndIf
 (10a) If (Detect($T_9$) == true) then
           (10) Append-head($T_9$, $y^0_{f+1,g}$) and Append-head($T_9$, $b^0_{f,g}$).
       EndIf
 (11a) If (Detect($T_{10}$) == true) then
           (11) Append-head($T_{10}$, $y^1_{f+1,g}$) and Append-head($T_{10}$, $b^0_{f,g}$).
       EndIf
 (12a) If (Detect($T_{11}$) == true) then
           (12) Append-head($T_{11}$, $y^0_{f+1,g}$) and Append-head($T_{11}$, $b^1_{f,g}$).
```

```
        EndIf
(13a)   If (Detect($T_{12}$) == true) then
            (13) Append-head($T_{12}$, $y^1_{f+1,g}$) and Append-head($T_{12}$, $b^1_{f,g}$).
        EndIf
(14a)   If (Detect($T_{13}$) == true) then
            (14) Append-head($T_{13}$, $y^1_{f+1,g}$) and Append-head($T_{13}$, $b^1_{f,g}$).
        EndIf
(15a)   If (Detect($T_{14}$) == true) then
            (15) Append-head($T_{14}$, $y^0_{f+1,g}$) and Append-head($T_{14}$, $b^0_{f,g}$).
        EndIf
(16)    $T_0 = \bigcup(T_7, T_8, T_9, T_{10}, T_{11}, T_{12}, T_{13}, T_{14})$.
EndProcedure
```

Lemma 11 *The algorithm*, **ParallelOneBitSubtractor**(T_0, f, g, j), *can be applied to finish the Boolean function of a parallel one-bit subtractor.*

Proof The algorithm, **ParallelOneBitSubtractor**(T_0, f, g, j), is implemented by means of the *extract, append-head* and *merge* operations. From the execution of Steps (1) through (7), they use the *extract* operations to form some different test tubes including different strands (T_1 to T_{14}). That is, T_1 contains all of the strands that have $y_{f,g} = 1$, T_2 consists of all of the strands that have $y_{f,g} = 0$, T_3 includes those that have $y_{f,g} = 1$ and $n_j = 1$, T_4 contains those that have $y_{f,g} = 1$ and $n_j = 0$, T_5 consists of those that have $y_{f,g} = 0$ and $n_j = 1$, T_6 includes those that have $y_{f,g} = 0$ and $n_j = 0$, T_7 contains those that have $y_{f,g} = 1, n_j = 1$ and $b_{f,g-1} = 1$, T_8 consists of those that have $y_{f,g} = 1, n_j = 1$ and $b_{f,g-1} = 0$, T_9 includes those that have $y_{f,g} = 1, n_j = 0$ and $b_{f,g-1} = 1$, T_{10} consists of those that have $y_{f,g} = 1, n_j = 0$ and $b_{f,g-1} = 0$, T_{11} includes those that have $y_{f,g} = 0, n_j = 1$ and $b_{f,g-1} = 1$, T_{12} contains those that have $y_{f,g} = 0, n_j = 1$ and $b_{f,g-1} = 0$, T_{13} consists of those that have $y_{f,g} = 0, n_j = 0$ and $b_{f,g-1} = 1$, and finally, T_{14} includes those that have $y_{f,g} = 0, n_j = 0$ and $b_{f,g-1} = 0$. Having finished Steps (1) through (7), this implies that eight different inputs of a one-bit subtractor as shown in Table 1 were poured into tubes T_7 through T_{14}, respectively.

Next, Steps (8a), (9a), (10a), (11a), (12a), (13a), (14a) and (15a) are used to detect whether there are DNA strands from tubes T_7 through T_{14}. If a true is returned from each operation, then the corresponding Steps (8a1) through (15a1) use the *append-head* operations to append $y^1_{f+1,g}$ or $y^0_{f+1,g}$, and $b^1_{f,g}$ or $b^0_{f,g}$ onto the head of every strand in the corresponding test tubes. After finishing Steps (8a1) through (15a1), we can say that eight different outputs of a one-bit subtractor in Table 1 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 tube T_0. Tube T_0 contains the strands finishing the subtraction of a bit. □

5.11 Reserving the result for intermediate computation of a modular multiplication

The procedure, **Encryption**(M, e, n), denoted in Sect. 4.1, is applied to perform computation of an exponential modular operation. The following DNA-based algorithm,

ReservedValue(T_2, f), is employed to reserve the result to intermediate computation of a modular multiplication.

```
Procedure ReservedValue($T_2, f$)
(1) For $j = 0$ to $k - 1$
    (1a) $T_3 = +(T_2, y^1_{f,j})$ and $T_4 = -(T_2, y^1_{f,j})$.
    (1b) Append-head($T_3, y^1_{f+1,j}$).
    (1c) Append-head($T_4, y^0_{f+1,j}$).
    (1d) $T_2 = \bigcup(T_3, T_4)$.
EndFor
EndProcedure
```

Lemma 12 *The algorithm*, **ReservedValue**(T_2, f), *can be applied to finish the function of reserving the intermediate result for computation of a modular multiplication.*

Proof Refer to Lemmas 1 through 11. □

5.12 The construction of a binary parallel adder

The one-bit adder (later introduced in Sect. 5.13) figures out the sum and the carry of two input bits and a previous carry. Two k-bit binary numbers each can be added by means of this one-bit adder. A binary parallel adder is also a Boolean function that finishes the arithmetic sum for two k-bit binary numbers. The following algorithm is proposed to finish the Boolean function of a binary parallel adder.

```
Procedure BinaryParallelAdder($T_0, f, a$)
(1) Append-head($T_0, z^0_{f,-1}$).
(2) For $j = 0$ to $k - 1$
        (2a) ParallelOneBitAdder($T_0, f, j, a, j$).
    EndFor
EndProcedure
```

Lemma 13 *The algorithm*, **BinaryParallelAdder**(T_0, f, a), *can be applied to finish the Boolean function of a binary parallel adder.*

Proof With the addition, the least significant position of the augend and the addend of k bits is added; the input carry must be 0. So, Step (1) uses the *append-head* operation to append 15-based DNA sequences for representing $z^0_{f,-1}$ onto the head of every strand in T_0. Step (2) is the main loop and is mainly used to finish the Boolean function of a binary parallel adder. Each execution of Step (2a) calls the procedure, **ParallelOneBitAdder**(T_0, f, j, a, j), to compute the arithmetic sum of one bit for the augend and the addend. Repeat execution of Step (2a) until the most significant bit for the augend and the addend is processed. Tube T_0 contains the strands finishing the arithmetic sum of k bits. □

5.13 The construction of a parallel one-bit adder

A one-bit adder is a Boolean function that forms the arithmetic sum of three input bits. It consists of three inputs and two outputs. Two of the input bits represent augend and addend bits to be added, respectively. The third input represents the carry from the previous lower significant position. The first output gives the value of the sum for augend and addend bits to be added. The second output gives the value of the carry to augend and addend bits to be added. The truth table of the one-bit adder is as follows:

Table 2 The truth table of a one-bit adder

Augend bit	Addend bit	Previous carry bit	Sum bit	Carry bit
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Suppose that two one-bit binary numbers denoted in Sect. 5.6, $y_{f,g}$ and $y_{f+1,g}$, represent the first input of a one-bit adder for $1 \leq f \leq (8 * k^2 + 2 * k)$ and $0 \leq g \leq k - 1$, and the first output of a one-bit adder, respectively, a one-bit binary number denoted in Sect. 5.4, $c_{a,j}$, represents the second input of a one-bit adder for $1 \leq a \leq (2 * k + 1)$ and $0 \leq j \leq k - 1$, and two one-bit binary numbers, $z_{f,g}$ and $z_{f,g-1}$, represent the second output and the third input of a one-bit adder, respectively. From [30, 31], two *distinct* DNA sequences are designed to represent the value "0" or "1" of every bit $z_{f,g-1}$ and $z_{f,g}$ for $1 \leq f \leq (8 * k^2 + 2 * k)$ and $0 \leq g \leq k - 1$. For the sake of convenience in our presentation, assume that $z^1_{f,g}$ contains the value of $z_{f,g}$ to be 1 and $z^0_{f,g}$ contains the value of $z_{f,g}$ to be 0. Also suppose that $y^1_{f+1,g}$ denotes the value of $y_{f+1,g}$ to be 1 and $y^0_{f+1,g}$ defines the value of $y_{f+1,g}$ to be 0. Similarly, assume that $z^1_{f,g-1}$ contains the value of $z_{f,g-1}$ to be 1 and $z^0_{f,g-1}$ contains the value of $z_{f,g-1}$ to be 0. The following algorithm is proposed to finish the Boolean function of a parallel one-bit adder.

```
Procedure ParallelOneBitAdder($T_0, f, g, a, j$)
    (1) $T_1 = +(T_0, y^1_{f,g})$ and $T_2 = -(T_0, y^1_{f,g})$.
    (2) $T_3 = +(T_1, c^1_{a,j})$ and $T_4 = -(T_1, c^1_{a,j})$.
    (3) $T_5 = +(T_2, c^1_{a,j})$ and $T_6 = -(T_2, c^1_{a,j})$.
    (4) $T_7 = +(T_3, z^1_{f,g-1})$ and $T_8 = -(T_3, z^1_{f,g-1})$.
```

```
(5) $T_9 = +(T_4, z^1_{f,g-1})$ and $T_{10} = -(T_4, z^1_{f,g-1})$.
(6) $T_{11} = +(T_5, z^1_{f,g-1})$ and $T_{12} = -(T_5, z^1_{f,g-1})$.
(7) $T_{13} = +(T_6, z^1_{f,g-1})$ and $T_{14} = -(T_6, z^1_{f,g-1})$.
(8a) If (Detect($T_7$) == true) then
        (8) Append-head($T_7$, $y^1_{f+1,g}$) and Append-head($T_7$, $z^1_{f,g}$).
     EndIf
(9a) If (Detect($T_8$) == true) then
        (9) Append-head($T_8$, $y^0_{f+1,g}$) and Append-head($T_8$, $z^1_{f,g}$).
     EndIf
(10a) If (Detect($T_9$) == true) then
        (10) Append-head($T_9$, $y^0_{f+1,g}$) and Append-head($T_9$, $z^1_{f,g}$).
     EndIf
(11a) If (Detect($T_{10}$) == true) then
        (11) Append-head($T_{10}$, $y^1_{f+1,g}$) and Append-head($T_{10}$, $z^0_{f,g}$).
     EndIf
(12a) If (Detect($T_{11}$) == true) then
        (12) Append-head($T_{11}$, $y^0_{f+1,g}$) and Append-head($T_{11}$, $z^1_{f,g}$).
     EndIf
(13a) If (Detect($T_{12}$) == true) then
        (13) Append-head($T_{12}$, $y^1_{f+1,g}$) and Append-head($T_{12}$, $z^0_{f,g}$).
     EndIf
(14a) If (Detect($T_{13}$) == true) then
        (14) Append-head($T_{13}$, $y^1_{f+1,g}$) and Append-head($T_{13}$, $z^0_{f,g}$).
     EndIf
(15a) If (Detect($T_{14}$) == true) then
        (15) Append-head($T_{14}$, $y^0_{f+1,g}$) and Append-head($T_{14}$, $z^0_{f,g}$).
     EndIf
(16) $T_0 = \bigcup(T_7, T_8, T_9, T_{10}, T_{11}, T_{12}, T_{13}, T_{14})$.
EndProcedure
```

Lemma 14 *The algorithm,* **ParallelOneBitAdder**(T_0, f, g, a, j), *can be applied to finish the Boolean function of a parallel one-bit adder.*

Proof The algorithm **ParallelOneBitAdder**(T_0, f, g, a, j) is implemented by means of the *extract, append-head* and *merge* operations. Steps (1) through (7) employ the *extract* operations to form some different test tubes including different strands (T_1 to T_{14}). That is, T_1 includes all of the strands that have $y_{f,g} = 1, T_2$ includes all of the strands that have $y_{f,g} = 0, T_3$ includes those that have $y_{f,g} = 1$ and $c_{a,j} = 1, T_4$ includes those that have $y_{f,g} = 1$ and $c_{a,j} = 0, T_5$ includes those that have $y_{f,g} = 0$ and $c_{a,j} = 1, T_6$ includes those that have $y_{f,g} = 0$ and $c_{a,j} = 0, T_7$ includes those that have $y_{f,g} = 1, c_{a,j} = 1$ and $z_{f,g-1} = 1, T_8$ includes those that have $y_{f,g} = 1, c_{a,j} = 1$ and $z_{f,g-1} = 0, T_9$ includes those that have $y_{f,g} = 1, c_{a,j} = 0$ and $z_{f,g-1} = 1, T_{10}$ consists of those that have $y_{f,g} = 1, c_{a,j} = 0$ and $z_{f,g-1} = 0, T_{11}$ includes those that have $y_{f,g} = 0, c_{a,j} = 1$ and $z_{f,g-1} = 1, T_{12}$ includes those that have $y_{f,g} = 0, c_{a,j} = 1$ and $z_{f,g-1} = 0, T_{13}$ includes those that have $y_{f,g} = 0, c_{a,j} = 0$

and $z_{f,g-1} = 1$, and finally, T_{14} consists of those that have $y_{f,g} = 0, c_{a,j} = 0$ and $z_{f,g-1} = 0$. Having finished Steps (1) through (7), this implies that eight different inputs of a one-bit adder as shown in Table 2 were poured into tubes T_7 through T_{14}, respectively.

Next, Steps (8a), (9a), (10a), (11a), (12a), (13a), (14a) and (15a) are used to detect whether there are DNA strands from tubes T_7 through T_{14}. If a true is returned from each operation, then the corresponding steps (8a1) through (15a1) use the *append-head* operations to append $y^1_{f+1,g}$ or $y^0_{f+1,g}$, and $z^1_{f,g}$ or $z^0_{f,g}$ onto the head of every strand in the corresponding test tubes. After finishing Steps (8a1) through (15a1), we can say that eight different outputs of a one-bit adder in Table 2 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 tube T_0. Tube T_0 contains the strands finishing the addition of a bit. □

5.14 The construction of an assignment operator

An assignment operator is an instruction of the first operand of k bits and the second operand of k bits that the value of the first operand is set to the value of the second operand. The following algorithm is applied to construct an assignment operator. This implies that the assignment operator can be used to update the value of C denoted in Sect. 5.4. The third parameter, a, in the algorithm is used to represent the ath updating for C.

```
Procedure AssignmentOperator($T_0$, $f$, $a$)
(1) For $j = 0$ to $k - 1$
        (1a) $T_1 = +(T_0, y^1_{f,j})$ and $T_2 = -(T_0, y^1_{f,j})$.
        (1b) Append-head($T_1$, $c^1_{a,j}$).
        (1c) Append-head($T_2$, $c^0_{a,j}$).
        (1d) $T_0 = \bigcup(T_1, T_2)$.
    EndFor
EndProcedure
```

Lemma 15 *The algorithm*, **AssignmentOperator**(T_0, f, a), *can be applied to finish the function of an assignment operator.*

Proof Step (1) is a loop and is used to perform the function of an assignment operator for tube T_0. On each execution of Step (1a), 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 $y_{f,j} = 1$, and the second tube T_2 consists of all of the strands that have $y_{f,j} = 0$. Next, each execution of Step (1b) uses the *append-head* operations to append $c^1_{a,j}$ onto the head of every strand in T_1. On each execution of Step (1c), it applies the *append-head* operations to append $c^0_{a,j}$ onto the head of every strand in T_2. Then, each execution of Step (1d) applies the *merge* operation to pour tubes T_1 and T_2 into T_0. Tube T_0 contains the strands finishing assignment of a bit. Repeat execution of Steps (1a) through (1d) until the k bits are processed. Tube T_0 contains the strands finishing assignment of k bits. This indicates that the ath updating of the value to C is finished. □

6 The attacking plan of breaking the Diffie–Hellman public-key cryptosystem

The Diffie–Hellman 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 public key between two communicating parties is represented as a k-bit binary number, $c_{(2*k+1),k-1} \ldots c_{(2*k+1),0}$, denoted in Sect. 5.4. An eavesdropper only needs to use the following algorithm to figure out the corresponding secret key.

Algorithm 2 The attacking plan of breaking the Diffie–Hellman public-key cryptosystem.

```
(1) Call Algorithm 1.
(2) **For** $j = 0$ **to** $k - 1$
        (2a) $T_1 = +(T_0, c_{(2*k+1),j})$ and $T_2 = -(T_0, c_{(2*k+1),j})$.
        (2b) $T_0 = \bigcup(T_0, T_1)$.
    **EndFor**
(3) **If** (Detect($T_0$) $==$ *true*) **then**
        (3a) Read($T_0$).
    **EndIf**
**EndAlgorithm**
```

Theorem 2 *From those steps in* Algorithm 2, *an eavesdropper can compute the corresponding secret key.*

Proof After the execution of Step (1) is performed, each pair (the public key, the corresponding secret key) is encoded by DNA strands in tube T_0. From the encrypted message overheard by an eavesdropper, after each execution of Steps (2a) and (2b) is finished, the corresponding secret key (the corresponding discrete logarithm) is stored in tube T_0. Therefore, a *true* is returned from the execution of Step (3). Next, from the execution of Step (3a), the answer is found from tube T_0. □

6.1 The power of Algorithm 2 for figuring out the secret key in the Diffie–Hellman public-key cryptosystem

It is assumed that between two communicating parties an eavesdropper overhears the encrypted message $c^0_{7,2}c^0_{7,1}c^1_{7,0}$. The eavesdropper uses Algorithm 2 to find the corresponding secret key (discrete logarithm of the encrypted message). When Step (1) in Algorithm 2 is executed, it invokes Algorithm 1 in Sect. 4.3. The first parameter in Algorithm 1 is a primitive root M for Z^*_7 and its length is three bits. Therefore, its binary value is $m^0_2m^1_1m^1_0$. The second parameter in Algorithm 1 is applied to encode all of the possible discrete logarithms for the encrypted message $c^0_{7,2}c^0_{7,1}c^1_{7,0}$, and its length is three bits. Therefore, the range of its value is from zero ($e^0_2e^0_1e^0_0$) through seven ($e^1_2e^1_1e^1_0$). The third parameter in Algorithm 1 is the divisor of the modular operation, and its length is three bits. Hence, its binary value is $n^1_2n^1_1n^1_0$.

Table 3 The result for tube T_0 is generated by **Init**(T_0)

Tube	The result is generated by **Init**(T_0)
T_0	$\{e_2^0e_1^0e_0^0, e_2^0e_1^0e_0^1, e_2^0e_1^1e_0^0, e_2^0e_1^1e_0^1, e_2^1e_1^0e_0^0, e_2^1e_1^0e_0^1, e_2^1e_1^1e_0^0, e_2^1e_1^1e_0^1\}$

Table 4 The results for tubes T_0 and T_θ are yielded by **SelectDiscreteLogarithm**(T_0, T_θ)

Tube	The result is generated by **SelectDiscreteLogarithm**(T_0, T_θ)
T_0	$\{e_2^0e_1^0e_0^0, e_2^0e_1^0e_0^1, e_2^0e_1^1e_0^0, e_2^0e_1^1e_0^1, e_2^1e_1^0e_0^0, e_2^1e_1^0e_0^1\}$
T_θ	$\{\theta_2^1\theta_1^1\theta_0^0\}$

Table 5 The result for tube T_n is generated by **MakeValue**(T_n)

Tube	The result is generated by **MakeValue**(T_n)
T_n	$\{n_2^1n_1^1n_0^1\}$

Table 6 The result for tube T_0 is generated by **InitialValue**(T_0)

Tube	The result is generated by **InitialValue**(T_0)
T_0	$\{c_{1,2}^0c_{1,1}^0c_{1,0}^1m_2^0m_1^1m_0^1e_2^0e_1^0e_0^0, c_{1,2}^0c_{1,1}^0c_{1,0}^1m_2^0m_1^1m_0^1e_2^0e_1^0e_0^1,$ $c_{1,2}^0c_{1,1}^0c_{1,0}^1m_2^0m_1^1m_0^1e_2^0e_1^1e_0^0, c_{1,2}^0c_{1,1}^0c_{1,0}^1m_2^0m_1^1m_0^1e_2^0e_1^1e_0^1,$ $c_{1,2}^0c_{1,1}^0c_{1,0}^1m_2^0m_1^1m_0^1e_2^1e_1^0e_0^0, c_{1,2}^0c_{1,1}^0c_{1,0}^1m_2^0m_1^1m_0^1e_2^1e_1^0e_0^1\}$

After the first execution of Step (0) in Algorithm 1 is completed, tubes T_0, T_θ, T_n, and T_1 are set to empty tubes. Next, after the first execution of Step (1) in Algorithm 1 is completed, the result for tube T_0 is shown in Table 3.

Next, after the first execution of Step (2) in Algorithm 1 is completed, the results for tubes T_0 and T_θ are shown in Table 4.

Next, after the first execution of Step (3) in Algorithm 1 is completed, the result for tube T_n is shown in Table 5.

Next, after the first execution of Step (4) in Algorithm 1 is completed, the result for tube T_0 is shown in Table 6.

Next, Steps (5a) through (5g) in Step (5) in Algorithm 1 are used to complete all of the computations for $M^0(\text{mod } n)$, $M^1(\text{mod } n) \ldots M^{n-2}(\text{mod } n)$, where the value of M is equal to three, and the value of n is equal to seven. This is to say that those operations are employed to perform all of the computations for $3^0(\text{mod } 7)$, $3^1(\text{mod } 7)$, $3^2(\text{mod } 7)$, $3^3(\text{mod } 7)$, $3^4(\text{mod } 7)$ and $3^5(\text{mod } 7)$. After those operations are all completed, the results for tubes T_0, T_1, and T_n are shown in Table 7. In Table 7 in tube T_0, $c_{7,2}^0c_{7,1}^0c_{7,0}^1$, $c_{7,2}^0c_{7,1}^1c_{7,0}^1$, $c_{7,2}^0c_{7,1}^1c_{7,0}^0$, $c_{7,2}^1c_{7,1}^1c_{7,0}^0$, $c_{7,2}^1c_{7,1}^0c_{7,0}^0$ and $c_{7,2}^1c_{7,1}^0c_{7,0}^1$ are, respectively, applied to store the final output of those modular operations. The immediate result generated by Steps (5a) through (5g) is omitted.

Next, because each operation in Algorithm 1 is completed, an eavesdropper can continue to execute Step (2) in Algorithm 2. The value of k is equal to three, so Steps (2a) and (2b) will be executed three times. The eavesdropper overhears the

Table 7 The results for tubes T_0, T_1, and T_n are generated by Steps (5a) through (5g) in Algorithm 1

Tube	The result is generated by Steps (5a) through (5g)
T_0	$\{c^0_{7,2}c^0_{7,1}c^1_{7,0}\cdots c^0_{1,2}c^0_{1,1}c^1_{1,0}m^0_2m^1_1m^1_0e^0_2e^0_1e^0_0,$ $c^0_{7,2}c^1_{7,1}c^1_{7,0}\cdots c^0_{1,2}c^0_{1,1}c^1_{1,0}m^0_2m^1_1m^1_0e^0_2e^0_1e^1_0,$ $c^0_{7,2}c^1_{7,1}c^0_{7,0}\cdots c^0_{1,2}c^0_{1,1}c^1_{1,0}m^0_2m^1_1m^1_0e^0_2e^1_1e^0_0,$ $c^1_{7,2}c^1_{7,1}c^0_{7,0}\cdots c^0_{1,2}c^0_{1,1}c^1_{1,0}m^0_2m^1_1m^1_0e^0_2e^1_1e^1_0,$ $c^1_{7,2}c^0_{7,1}c^0_{7,0}\cdots c^0_{1,2}c^0_{1,1}c^1_{1,0}m^0_2m^1_1m^1_0e^1_2e^0_1e^0_0,$ $c^1_{7,2}c^0_{7,1}c^1_{7,0}\cdots c^0_{1,2}c^0_{1,1}c^1_{1,0}m^0_2m^1_1m^1_0e^1_2e^0_1e^1_0\}$
T_1	$\varnothing$
T_n	$\{n^1_2n^1_1n^1_0\}$

Table 8 The result for tube T_0 is generated by Steps (2a) and (2b) in Algorithm 2

Tube	The result is generated by Steps (2a) and (2b)
T_0	$\{c^0_{7,2}c^0_{7,1}c^1_{7,0}\cdots c^0_{1,2}c^0_{1,1}c^1_{1,0}m^0_2m^1_1m^1_0e^0_2e^0_1e^0_0\}$

encrypted message $c^0_{7,2}c^0_{7,1}c^1_{7,0}$, and the encrypted message $c^0_{7,2}c^0_{7,1}c^1_{7,0}$ is used to find the corresponding secret key (discrete logarithm). Therefore, after each operation of Steps (2a) and (2b) is completed, the result for tube T_0 is shown in Table 8. Next, a *true* is returned from the first execution of Step (3), so the first execution of Step (3a) is applied to obtain the answer $e^0_2e^0_1e^0_0$. That is to say that for the encrypted message $c^0_{7,2}c^0_{7,1}c^1_{7,0}$ the corresponding secret key (discrete logarithm) is $e^0_2e^0_1e^0_0$.

7 Complexity assessment

Theorem 3 *Suppose that the length of a secret key (discrete logarithm) in the Diffie–Hellman public-key cryptosystem is k bits. The Diffie–Hellman public-key cryptosystem can be broken with $O(k^3)$ biological operations proposed by Adleman* [2, 30, 31] *from solution space.*

Proof Refer to Algorithm 1. □

Theorem 4 *Suppose that the length of a secret key (discrete logarithm) in the Diffie–Hellman public-key cryptosystem is k bits. The Diffie–Hellman public-key cryptosystem can be broken with $O(2^k)$ library strands from solution space.*

Proof Refer to Algorithm 1. □

Theorem 5 *Suppose that the length of a secret key (discrete logarithm) in the Diffie–Hellman public-key cryptosystem is k bits. The Diffie–Hellman public-key cryptosystem can be broken with $O(1)$ tubes from solution space.*

Proof Refer to Algorithm 1. □

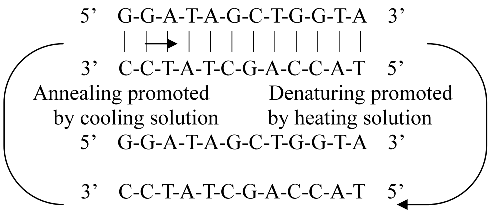
5' G-G-A-T-A-G-C-T-G-G-T-A 3'
3' C-C-T-A-T-C-G-A-C-C-A-T 5'
Annealing promoted by cooling solution
Denaturing promoted by heating solution
5' G-G-A-T-A-G-C-T-G-G-T-A 3'
3' C-C-T-A-T-C-G-A-C-C-A-T 5'

Fig. 1 DNA denaturing and annealing

Theorem 6 *Suppose that the length of a secret key* (*discrete logarithm*) *in the Diffie–Hellman public-key cryptosystem is k bits. The Diffie–Hellman public-key cryptosystem can be broken with the longest library strand*, $O(k^3)$, *from solution space.*

Proof Refer to Algorithm 1. □

8 Biological implementation

8.1 DNA structure

The genetic information of cellular organisms is encoded by DNA (deoxyribonucleic acid) [17, 35, 36]. DNA includes polymerchains which are commonly regarded as DNA strands. By means of an automated process, DNA strands may be synthesized to order. Each strand may be made of a sequence of nucleotides, or bases, attached to a sugar-phosphate "backbone". The four DNA nucleotides are adenine, guanine, cytosine and thymine, commonly abbreviated to A, G, C and T respectively. From chemical convention each strand has a $5'$ end and a $3'$ end. Because one end of the single strand has a free (i.e., unattached to another nucleotide) $5'$ phosphate group, and the other has a free $3'$ deoxyribose hydroxyl group, therefore, any single strand has a natural orientation [17, 35, 36].

The classical double helix of DNA is formed when two separate single strands bond. Bonding occurs by the pairwise attraction of bases; A bonds with T and G bonds with C. The pairs (A, T) and (G, C) are therefore known as complementary base pairs [17, 35, 36]. Double-stranded DNA may be denatured into single strands by heating the solution to a temperature determined by the composition of the strand [17, 37]. Heating breaks the hydrogen bonds between complementary strands (Fig. 1) [17]. Because a G–C pair is joined by three hydrogen bonds, the temperature required to break it is slightly higher than that for an A–T pair, joined by only two hydrogen bonds [17]. This factor must be taken into account when designing sequences to represent computational elements. Annealing is the reverse of melting, whereby a solution of single strands is cooled, and allowing complementary strands to bind together (Fig. 1) [17]. In double-stranded DNA, if one of the single strands contains a discontinuity (i.e., one nucleotide is not bonded to its neighbor) then this may be repaired by DNA ligase [38]. This allows us to create a unified strand from several bound together by their respective complements.

From [17], we now introduce in detail how the biological operations introduced in Sect. 3.1 might be implemented in the laboratory. Each implementation illustrates only one possible way to perform the computational behavior of the biological operation. Future improvements in laboratory techniques may well yield more efficient and error resistant implementations of the basic steps of our algorithm, but this does not diminish the theoretical power of the model. From [17], we simply offer descriptions of implementation in order to show the in principle feasibility of executing our algorithm in vitro (that is to say, every step of our algorithm is completely feasible using existing laboratory techniques). From a biological standpoint, all sequences generated to represent bits must be checked to ensure that the DNA strands that they encode do not form unwanted secondary structures with one another (i.e., strands remain separate at all times, and only bind together when this is required by the algorithm). The problem of strand design for DNA-based computing has been addressed at length, and we use the methods described in [17, 30, 31] to minimize the possibility of unwanted binding.

8.2 Extract

Affinity purification is applied to extract any strands from T containing s. This process applies a probe sequence, which is complementary to the target sequence being searched for. Probes are fixed to a surface, and capture through annealing any strands containing the target sequence. Captured strands may then be separated from the rest of the population by placing them in a separate solution, which is heated to break the bonds between the probes and the target sequence. The probe used is therefore the complementary sequence of s. Retained strands are placed in one new tube, $U = +(T, s)$, and the remainder are placed in another new tube, $V = -(T, s)$.

8.3 Merge

The contents of tubes $\{T_i\}$ are simply merged by pouring. Because the number of tubes will generally be low, this is considered to be a constant-time operation.

8.4 Discard

The contents of T are discarded, and T is replaced by a new, empty tube.

8.5 Amplify

The polymerase chain reaction (PCR) is used, with its initial input being tube T. This reaction is used to massively amplify (possibly small) amounts of DNA that begin and end with specific primer sequences. As every strand in tube T is delimited by these sequences, they are all copied by the reaction. The result of the PCR is then divided equally between the specified number of tubes (the number of PCR cycles may therefore be adjusted to ensure a constant DNA volume per tube, regardless of the number of tubes).

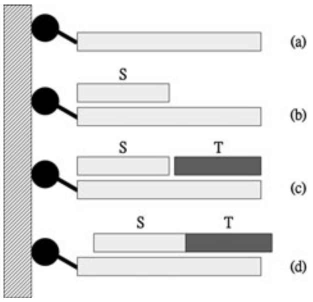

Fig. 2 Concatenation process: (**a**) Linker strand affixed to surface. (**b**) S anneals to linker strand. (**c**) T anneals to linker strand, adjacent to S. (**d**) S and T ligated to form a single strand, which is then freed by heating the solution

8.6 Concatenate (s_1, s_2)

Two strands (labeled S and T in Fig. 2) may be concatenated by the following process: create a *linker* strand, which has a sequence that is the complement of S followed by the complement of T. This linker strand is affixed to a surface with a magnetic bead (Fig. 2(a)). Strand S is then added to the solution, and anneals with the linker strand at the appropriate position (Fig. 2(b)). Strand T is then added to the solution, and this also anneals with the linker strand, at a position immediately adjacent to strand S (Fig. 2(c)). The ligase enzyme is then added to the solution to seal the "nick" between S and T, forming a single strand which may be freed by heating the solution to break its bonds with the linker strand (Fig. 2(d)).

8.7 Append-head(T, s)

The implementation of the *concatenate*() operation defined above may easily be used to append a specific sequence, s, to the head of each strand in a tube T. The sequence s corresponds, in this case, to the strand S defined in Fig. 2, and strand T in Fig. 2 corresponds to the beginning sequence of every strand in the tube. In this case, only the beginning sequence of every strand anneals to the linker strand. Clearly, then, after a series of *append-head*() operations has been performed on a strand, its sequence will be made up of a number of sequences representing bit-strings.

8.8 Detect

The tube T is run through a gel electrophoresis process, which is generally used to sort DNA strands on length. Any DNA present in T shows up as a visible band in the gel; if DNA strands of the appropriate length are present, the operation returns *true*. If there are no visible bands corresponding to DNA of the correct length, then the operation returns *false*. The length criterion is used to ensure that DNA fragments present do not cause a false positive result. If the DNA in the band corresponding to the contents of T is required in a subsequent processing step, the band may be excised from the gel by cutting and then soaked to remove the strands for further use.

9 Conclusions

The number of steps any classical computer requires in order to find discrete logarithm of a k-bit increases exponentially with k, at least by means of using algorithms [3] known at present. Shor's *quantum* factoring and discrete logarithm algorithm [39] includes that the two main components, modular exponentiation (computation of $a^x \bmod n$) and the inverse quantum Fourier transform (QFT) take only $O(k^3)$ operations. In this article, Our *molecular* discrete logarithm algorithm demonstrates theoretically how basic biological operations can be used to solve the problem of discrete logarithm with $O(k^3)$ biological operations. Both of Shor's factoring and discrete logarithm algorithm and our discrete logarithm algorithm need to simultaneously deal with 2^{1024} bit information to find the discrete logarithm of 1024 bits used in the current Diffie–Hellman public-key cryptosystem. However, due to current many technical difficulties, therefore, the two algorithms currently do not in fact find the discrete logarithm of 1024 bits. This implies that if a quantum computer and a molecular computer are *really* constructed in the future (perhaps after many years), then Shor's factoring and discrete logarithm algorithm and our discrete logarithm algorithm have very high feasibility for solving the problem of discrete logarithm.

In [42], it is demonstrated that the difficult problem of elliptic curve discrete logarithms can be solved on a DNA-based computer, and the application of DNA computing is proposed in another popular cryptosystem, ECC, which is more complex and has more challenge in cryptoanalysis. In [42], solving elliptic curve discrete logarithm takes a series of steps that is polynomial in the input size, and it has also been shown that humans' complex mathematical operations can be performed directly with basic biological operations.

Adleman [1] indicated that at a time unit 2^n combination states can be simultaneously processed by means of biological operations, but just one state can be processed in a digital computer. Therefore, Adleman [1] also pointed out that a digital computer will take exponential time to complete the digital-computer simulation of biological algorithms. This implies that the digital-computer simulation of the proposed biological algorithms for breaking public-key cryptosystems is perhaps inefficient.

Acknowledgements The authors would like to give many thanks to Dr. Amos who is the author of the 17th reference for offering valuable information on Sect. 8 entitled "Biological Implementation".

References

1. Feynman RP (1961) In: Gilbert DH (ed) Minaturization. Reinhold, New York, pp 282–296
2. Adleman L (1994) Molecular computation of solutions to combinatorial problems. Science 266(11):1021–1024
3. Diffie W, Hellman M (1976) New directions in cryptography. IEEE Trans Inf Theory 22(6):644–654
4. Adleman L, Rothemund PWK, Roweis S, Winfree E (1999) On applying molecular computation to the data encryption standard. In: The 2nd annual workshop on DNA computing, Princeton University. DIMACS: series in discrete mathematics and theoretical computer science. Am Math Soc, Providence, pp 31–44
5. Guo M, Chang W-L, Ho M, Lu J, Cao J (2005) Is optimal solution of every NP-complete or NP-hard problem determined from its characteristic for DNA-based computing. BioSystems 80(1):71–82
6. Muskulus M, Besozzi D, Brijder R, Cazzaniga P, Houweling S, Pescini D, Rozenberg G (2006) Cycles and communicating classes in membrane systems and molecular dynamics. Theor Comput Sci 372(2–3):242–266
7. Reif JH, LaBean TH (2007) Autonomous programmable biomolecular devices using self-assembled DNA nanostructures. Commun ACM 50(9):46–53
8. Wu G, Seeman NC (2006) Multiplying with DNA. Nat Comput 5(4):427–441

9. Macdonald J, Li Y, Sutovic M, Lederman H, Pendri K, Lu W, Andrews BL, Stefanovic D, Stojanovic MN (2006) Medium scale integration of molecular logic gates in an automaton. Nano Lett 6(11):2598–2603
10. Ekani-Nkodo A, Kumar A, Fygenson DK (2004) Joining and scission in the self assembly of nanotubes from DNA tiles. Phys Rev Lett 93:268301
11. Dehnert M, Helm WE, Hütt M-Th (2006) Informational structure of two closely related eukaryotic genomes. Phys Rev E 74:021913
12. Müller BK, Reuter A, Simmel FC, Lamb DC (2006) Single-pair FRET characterization of DNA tweezers. Nano Lett 6:2814–2820
13. Dirks RM, Bois JS, Schaeffer JM, Winfree E, Pierce NA (2007) Thermodynamic analysis of interacting nucleic acid strands. SIAM Rev 49(1):65–88
14. Lipton R (1995) DNA solution of hard computational problems. Science 268:542–545
15. Yeh C-W, Chu C-P, Wu K-R (2006) Molecular solutions to the binary integer programming problem based on DNA computation. Biosystems 83(1):56–66
16. Guo M, Ho M, Chang W-L (2004) Fast parallel molecular solution to the dominating-set problem on massively parallel bio-computing. Parallel Comput 30(9–10):1109–1125
17. Amos M (2005) Theoretical and experimental DNA computation. Springer, Berlin
18. Ho M, Chang W-L, Guo M, Yang LT (2004) Fast parallel solution for set-packing and clique problems by DNA-based computing. IEICE Trans Inf Syst E-87D(7):1782–1788
19. Chang W-L, Guo M, Ho M (2004) Towards solution of the set-splitting problem on gel-based DNA computing. Future Gener Comput Syst 20(5):875–885
20. Chang W-L, Guo M (2003) Solving the set-cover problem and the problem of exact cover by 3-sets in the Adleman-Lipton's model. BioSystems 72(3):263–275
21. Ho M (2005) Fast parallel molecular solutions for DNA-based supercomputing: the subset-product problem. BioSystems 80:233–250
22. Henkel CV, Bäck T, Kok JN, Rozenberg G, Spaink HP (2007) DNA computing of solutions to knapsack problems. Biosystems 88(1–2):156–162
23. Chang W-L (2007) Fast parallel DNA-based algorithms for molecular computation: the set-partition problem. IEEE Trans Nanobiosci 6(1):346–353
24. Chang W-L, Ho M, Guo M (2005) Fast parallel molecular algorithms for DNA-based computation: factoring integers. IEEE Trans Nanobiosci 4(2):149–163
25. Boneh D, Dunworth C, Lipton RJ (1996) Breaking DES using a molecular computer. In: Proceedings of the 1st DIMACS workshop on DNA based computers, 1995. DIMACS series in discrete mathematics and theoretical computer science, vol 27. Am Math Soc, Providence, pp 37–66
26. Zhang DY, Winfree E (2008) Dynamic allosteric control of noncovalent DNA catalysis reactions. J Am Chem Soc 130:13921–13926
27. Chang W-L, Ho M, Guo M (2004) Molecular solutions for the subset-sum problem on DNA-based supercomputing. BioSystems 73(2):117–130
28. Seelig G, Soloveichik D, Zhang D-Y, Winfree E (2006) Enzyme-free nucleic acid logic circuits. Science 314(5805):1585–1588
29. Kari L, Konstantinidis S, Sosík P (2005) On properties of bond-free DNA languages. Theor Comput Sci 334(1–3):131–159
30. Braich RS, Johnson C, Rothemund PWK, Hwang D, Chelyapov N, Adleman LM (2001) Solution of a satisfiability problem on a gel-based DNA computer. In: Proceedings of the 6th international conference on DNA computation. Lecture notes in computer science series, vol 2054. Springer, Berlin, pp 27–42
31. Adleman LM, Braich RS, Johnson C, Rothemund PWK, Hwang D, Chelyapov N (2002) Solution of a 20-variable 3-SAT problem on a DNA computer. Science 296(5567):499–502
32. Koblitz N (1987) A course in number theory and cryptography. Springer, Berlin. ISBN:0387942939
33. Rivest RL, Shamir A, Adleman L (1978) A method for obtaining digital signatures and public-key cryptosystem. Commun ACM 21:120–126
34. Blakley GR A computer algorithm for calculating product AB modulo M. IEEE Trans Comput c-32(5):497–500
35. Adams RL, Knowler JT, Leader DP (1986) The biochemistry of the nucleic acids, 10th edn. Chapman & Hall, London
36. Watson J, Gilman M, Witkowski J, Zoller M (1992) Recombinant DNA, 2nd edn. Scientific American Books
37. Breslauer K, Frank R, Blocker H, Marky L (1986) Predicting DNA duplex stability from the base sequence. Proc Natl Acad Sci 3746–3750

38. Brown T (1993) Genetics: a molecular approach. Chapman & Hall, London
39. Shor PW (1997) Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Comput 26(5):1484–1509
40. Kershner RJ, Bozano LD, Micheel CM, Hung AH, Fornof AR, Cha JN, Rettner CT, Bersani M, Frommer J, Rothemund PWK, Wallraff GM (2009) Placement and orientation of individual DNA shapes on lithographically patterned surfaces. Nat Nanotechnol 16:557–561
41. Barish RD, Schulman R, Rothemund PWK, Winfree E (2009) An information-bearing seed for nucleating algorithmic self-assembly. PNAS 106:6054–6059
42. Li K, Zou S, Xv J (2008) Fast parallel molecular algorithms for DNA-based computation: solving the elliptic curve discrete logarithm problem over GF(2^n). J Biomed Biotechnol 2008:518093. doi:10.1155/2008/518093

Weng-Long Chang received the Ph.D. degree in Computer Science and Information Engineering from National Cheng Kung University, Taiwan, Republic of China, in 1999. He is currently a full Professor at the Department of Computer Science and Information Engineering in National Kaohsiung University of Applied Sciences. His researching interests include quantum algorithms, adiabatic quantum algorithms, DNA-based algorithms, and languages and compilers for parallel computing.

Shu-Chien Huang received the Ph.D. degree in Computer Science and Information Engineering from National Cheng Kung University, Taiwan, Republic of China, in 1999. He is currently an assistant Professor at the Department of Computer Science in Pingtung University of Education. His researching interests include image processing, quantum algorithms, and DNA-based algorithms.

Kawuu Weicheng Lin received the B.Sc. from the Department of Computer Science and Information Engineering, National Taiwan University (NTU), Taiwan, 1999, and received his Ph.D. form the Department of Computer Science and Information Engineering, National Cheng-Kung University (NCKU), Taiwan, 2006. Since August 2007, he has been an assistant Professor at the Department of Computer Science and Information Engineering, National Kaohsiung University of Applied Sciences (KUAS), Taiwan. His research interests include data mining and its applications, sensor technologies, and parallel and distributed computing. He is a member of Phi Tau Phi honorary society, and has won the Phi Tau Phi Scholastic Honor in 2006.

Michael (Shan-Hui) Ho received his Ph.D. degree in Information Systems and Management Science from University of Texas at Austin, USA, in 1988. He is a full Professor of both Computer Center and Institute of Electric Engineering in National Taipei University. His research interests are Bioinformatics Computing, Parallel Process and Computing, Software Engineering, and Computation.