International Journal of Computational Intelligence in Control
Vol. 12 No. 2 (July-December, 2020)

A DNA Number System and its Applications

P. V. Gopalacharyulu[1] and G. P. Raja Sekhar[2]

[1]Research Scientist, Biosystems Modelling, VTT Technical Research Center of Finland, Tietotie 2
Espoo, P. O. Box – 1000, 02044VTT, Finland, *E-mail: cugopal@gmail.com*

[2]Associate Profesor, Department of Mathematics, Indian Institute of Technology Kharagpur, Kharagpur 721302, India
E-mail: rajas@maths.iitkgp.ernet.in

Abstract: *In this paper, we present a DNA number system to represent numbers in terms of DNA strands instead of numerical digits. Triplets of nucleotides, called codons, serve as digits in our DNA number system. In this number system, we distinguish integers from real numbers based on the length of the corresponding DNA sequence representation. Further in this paper, we describe a search method that performs the task in a unit time, irrespective of the input size. This method uses the novel idea of the DNA annealing property. We also propose a classical cryptosystem in terms of a DNA algorithm.*

Key words : *DNA sequence, Number system, Searching, Cryptosystem.*

1. INTRODUCTION

DNA computing is a fast growing area of research concerned with the use of DNA molecules for the implementation of computational processes. It is a multidisciplinary area of research involving interplay between molecular biology and information science. Since the first practical example by Adleman [1], there has been intensive research into the use of DNA molecules as a tool for calculations, simulating the digital information processing procedures in conventional computers [2-11]. It has been shown by many research accomplishments that any process that could naturally be described as an algorithm can be realized by DNA computation. Computing with DNA offers the advantages of massive degree of miniaturization and parallelism. However, the main application of DNA in computing technology will be rather to perform complex molecular constructions, diagnostics and evolutionary tasks [8, 12, 13].

Adleman described in detail a library of operations, which are useful when working with a molecular computer and estimated that given an arbitrary pair of plain-text and cipher-text, one can recover the Data Encryption Standard (DES) key in about four months of work. Furthermore, it was shown that under chosen plain-text attack, it is possible to recover the DES key in one day using suitable preprocessing. This method can be generalized to break any cryptosystem, which uses keys of length less than 64 bits [14, 15]. Another hard solvable problem is the factorization of a large number into its prime factors. This may be used to break a public key cryptosystem. There are a number of hard solvable problems from graph theory and operations research which may be solvable by DNA computing [8, 13]. While there are advantages of DNA computation like parallelism, high information density, and the DNA reaction speed etc, some disadvantages also exist, most important one being the lack of a unique generalization. The DNA encoding and the possible methodology in handling a problem may not be applicable to other problem.

In this paper, we propose a new *DNA number system* and its applications to *searching* and *cryptosystems*. DNA strands are sequences of 4 nucleotides A, T, G and C. In the present work, we describe a method to represent numbers in terms of these DNA strands instead of numerical digits and call this system, the *DNA number system (DNA – NS)*. In order to distinguish integers from real numbers, every integer is represented as a DNA sequence whose length is a multiple of 3 and every real number is represented as a DNA sequence of length one more than a multiple of 3. We use the concept of *codons* to define a DNA number function (DNF) to represent negative numbers. Further, we shall describe a search method that performs the task in a unit time, once the input is given, irrespective of the input size. This method uses the novel idea of DNA annealing property. We also propose a classical cryptosystem in terms of a DNA algorithm.

2. REVIEW OF POSITIONAL NUMBER SYSTEMS

Usually, numbers are represented in the decimal form. In the decimal number system, any number is a sequence of 10 digits or bases {0, 1, 2, … 9}. Since the number of base symbols is 10, this number system is called base 10 number system or simply the decimal number system. The digital computers make use of binary numbers i.e., numbers to the base 2. In the binary system any number is a sequence of 0's and 1's.

In general, any positive integer can be used as the base of a number system. Let *b* be a positive integer. With *b* as the base, any real number can be represented using symbols from the set {0, 1, 2… b-1} as digits. When we write base b numbers we keep b as a suffix to the number to indicate that the representation is to the base b. For example: $(1101)_2$ stands for a binary number with digits 1101.

Since the decimal number system is the most commonly used system, realizing the decimal equivalents of base b numbers and vice versa would be an important task. A base b number $(b_k b_{k-1} \dots b_1 b_0 . b_{-1} b_{-2} b_{-3} \dots b_{-r})_b$ – where "." is the decimal separator, the digits b_j for j in {0, 1, 2, 3… k} represent the integer part and digits b_{-j} for j in {1, 2, …, r} represent the decimal or precision part – has its decimal equivalent

$$b_k b^k + b_{k-1} b^{k-1} + \dots + b_1 b + b_0 + b_{-1} b^{-1} + b_{-2} b^{-2} + \dots + b_{-r} b^{-r}.$$

Conversely, base b equivalent of a given decimal number $(N.F)_{10}$ (N is the integer part of the decimal number and F is the precision) can be obtained as follows: the base b equivalent of the decimal integer N_{10} is given by $(N)_b = [N/b]_b \# (N\%b)$, where $[N/b]$ represents the integer part of N/b (i.e. the largest integer $\leq N/b$), $N\%b$ represents remainder when N is divided by b and the operator # is the *concatenation* of *strings* over the alphabet {0, 1, 2 … b–1}. The base b equivalent of the precision F is given by $(F)_b = [F*b] \# (F*b - [F*b])_b$ where $[f]$ represents the integer part of any real number f. Then, base b equivalent of the real number $(N.F)_{10}$ is: $(N)_b.(F)_b$.

Example 2.1

1. The number $(101101.011)_2$ is equivalent to the decimal number

   $1*2^5+0*2^4+1*2^3+1*2^2+0*2^1+1*2^0+0*2^{-1}+1*2^{-2}+1*2^{-3}$

   $= 32 + 8 + 4 + 1 + 0.25 + 0.125$

   $= 45.375$

2. $(10.1125)_{10} = (1010.0001110011001100\dots)_2$

2|10	0.1125	* 2 = 0.225
2| 5 – 0	0.225	* 2 = 0.45
2| 2 – 1	0.45	* 2 = 0.9
1 – 0	0.9	* 2 = 1.8
	0.8	* 2 = 1.6
	0.6	* 2 = 1.2
	0.2	* 2 = 0.4
	0.4	* 2 = 0.8

Table 2.1 *A worked example of computing the binary equivalent of a decimal number.*

With this background, we introduce the DNA number system below.

3. DNA NUMBER SYSTEM

DNA strands are sequences of 4 nucleotides A, T, G and C. In what follows, we shall describe a method to represent numbers in terms of these DNA strands instead of numerical digits and call this system, the *DNA number system (DNA – NS)*. To be able to distinguish integers from real numbers, every integer is represented as a DNA sequence whose length is a multiple of 3 and every real number is represented as a DNA sequence of length one more than a multiple of 3.

Before describing the representation of integers under the new number system, let us define a *codon*. A *codon* is a triplet of the nucleotides A, T, G or C. *Example*: ATG, GCT, AAC etc. are *codons*.

3.1. Representing Integers from 0 to 63

We shall define the number system using the codons as the digits. Since the number of nucleotides is 4, there exist $4^3 = 64$ distinct codons. A number system which uses the codons as digits would therefore be numerically equivalent to the base 64 number system. The problem of defining the number system thus reduces to defining a mapping between the codons and the numbers 0, 1, 2… 63.

A natural way to do this is to associate the 64 distinct codons with all three digit numbers to the base 4 (which are exactly 64 in number). Note that, the smallest and largest three digit numbers to the base 4 are 0 and 63 $(=4^3-1)$ respectively. Also note that every number between 0 and 63 has a unique base 4 representation.

Therefore, our idea is to assign a one – one correspondence between the sets I = {0, 1, 2, 3} and D = {A, T, G, C} and hence represent the corresponding numbers from 0 to 63 in terms of the codons.

Let us define **f**: I $\to$ D as follows:

$$\mathbf{f}(0) = A,\ \mathbf{f}(1) = T,\ \mathbf{f}(2) = G,\ \mathbf{f}(3) = C \qquad (1)$$

Now, using the correspondence given in (1) we have,

AAA = $(000)_4$ = 0	AGA = (020)4 = 8	
AAT = $(001)_4$ = 1	CCC = $(333)_4$ = 63	
AAG = $(002)_4$ = 2	GCT = $(231)_4$ = 45	etc.,

It may be noted that **f** is bijective. In fact, in a similar way, one can define a total number of 4! (=24) such bijective functions from I to D and correspondingly in each case, one can obtain a unique number associated with each codon.

Definition 3.1: DNA Number Function (DNF)

Since each of the 4! functions such as the one defined in (1) gives rise to a unique number system, it is very important to know which function is in use for computing the numerical equivalent of a DNA strand or vice versa. Let us call each of these functions as a *DNA number function (DNF)*

3.2. Representing Numbers Beyond 63

Now let us proceed with the numbers beyond 63. Let c_0, c_1, c_2, … , c_r be any numbers from {0, 1, 2, … , 63} (where

$r \geq 0$). Then the number (crcr-1 … c1c0) to the base 64 has the decimal equivalent

$$c_r*64^r + c_{r-1}*64^{r-1} + \ldots + c_1*64^1 + c_0*64^0$$
$$= c_r*64^r + c_{r-1}*64^{r-1} + \ldots + c_1*64 + c_0$$

In the DNA number system, the role of each c_i is played by a codon. Thus any number greater than or equal to 64 is a sequence of two or more codons whose value is computed to the base 64 using the above interpretation of codons as numbers from 0 to 63.

Example 3.2.1

With the DNF defined in equation (1), we have:

1.	AGT TGA	AGT= $(021)_4$ = 9
	= 9*64+24	TGA= $(120)_4$ = 24
	= 576+24	
	= 600	
2.	GCC TAG AGA	GCC= $(233)_4$ = 31
	= 31*64^2 + 18*64+8	TAG= $(102)_4$ = 18
	= 31*4096+18*64+8	AGA= $(020)_4$ = 8
	= 126976+1152+8	
	= 128136.	

3. 757739 = AAG CGA CCC GGC

$$\begin{array}{r|l}
4 & 757739 \\
4 & 189434 \cdots 3 = C \\
4 & 47358 \cdots 2 = G \\
4 & 11839 \cdots 2 = G \\
4 & 2959 \cdots 3 = C \\
4 & 739 \cdots 3 = C \\
4 & 184 \cdots 3 = C \\
4 & 46 \cdots 0 = A \\
4 & 11 \cdots 2 = G \\
4 & 2 \cdots 3 = C \\
4 & 0 \cdots 2 = G
\end{array}$$

Note that, the approach of using codons as digits constrains the length of a DNA strand representing a number to be a multiple of 3. This constraint on the length will be exploited to distinguish the floating point numbers from integers, as described later.

Example 3.2.2

1. AGT GTA	2.AGT GTA
= $(020120)_4$	= $(9Y)_{64}$ where Y=24
= $0*4^5+2*4^4+0*4^3+1*4^2+2*4^1+0$	= $9*64^1$+Y
= 512+64+16+8	= 576+24
= 600	= 600

So far, we have described a way to represent non-negative integers. We can extend our system to represent negative integers as well as real numbers.

3.3. Representing Negative Integers

One way to represent the negative numbers could be to make it a convention that every negative number starts with a codon whose numerical equivalent is "0" with respect to the *DNF* in use. If we use any other codon whose numerical equivalent is non-zero, it would create ambiguity as to whether the first codon represents the sign or the first digit of the number. In contrast, if we use the codon whose numerical equivalent is 0, the net value remains the same, because appending 0 prior to a number does not alter the net value. In the example number system that we have described above using the *DNF* given in equation (1) (see section 3.1), the codon **AAA** equals "0". Hence, in this number system, the negative numbers can be represented with DNA strands starting with **AAA.**

Example 3.3.1

1. AAA TGC CGA
   = - $(123320)_4$
   = - (1024+512+192+48+8+0)
   = -1784.
2. –51 = AAA CAC.

The following example illustrates the ambiguity caused by using a randomly chosen codon whose value is non-zero to represent the minus sign.

Example 3.3.2: The DNA equivalent of 75 is AAT AGC.

Choosing **AAT** to represent the negative sign, we have

75 = AAT AGC = - $(023)_4$ = -11.

Hence the codon **AAT** fails to unambiguously represent the negative sign. The same holds true for any codon whose numerical equivalent is non-zero. Therefore, one has to choose the codon whose numerical equivalent is 0 to represent the negative sign.

3.4. Representing Real Numbers

To represent the real numbers, let us fix the first nucleotide to represent the sign and reserve some fixed number of codons to represent the integral part (unsigned) of the real number followed by some fixed number of codons to represent the decimal part.

To convert a real number into a DNA sequence, we will first set the nucleotide representing the sign accordingly, find the base 64 equivalent of the number and replace each base 64 digit in the so-obtained base 64 representation of the real number by its equivalent codon. In doing this, one particular *DNF* is chosen and is fixed.

Similarly to find the numerical equivalent of a DNA strand, we will first check whether the strand is of length

$3p+1$ for some $p = 0, 1, 2, 3 \ldots$ If not, the corresponding DNA strand does not represent a real number. If the length condition is satisfied, then first we will set the sign by looking at the first nucleotide. Next, by knowing the *DNF* in use, each codon is recognized with its equivalent number. These numbers are the digits of the base 64 representation of the real number we are looking for. Hence by knowing the number of codons used to represent integral part and the decimal part, one can compute the required real number.

The number of codons that have to be reserved depends upon the magnitudes of numbers present in the problem under consideration. Suppose in a particular case, the highest real number has its integer part equal to an integer "*a*". One has to find the first natural number *n* such that $a \leq 64n$. Then *n* number of codons can be used to represent the integer part. And the number of codons to represent the precision can be chosen depending on the maximum number of significant digits in the precision to be used.

Definition 3.4.1: DNA Sign Function

The sign of the real number is determined by a 1 - 1 function from the set {+, -} to a subset of {A, T, G, C}. There are exactly $4C_2 * 2! = 12$ such functions. We call each of these 12 functions a DNA sign function (*DSF*).

Suppose that 5 codons are used to represent integral part and 5 codons are used for decimal part. Then using the *DNF* defined in equation (1) (see section 3.1) to determine codons as numerical digits, we have:

G AGT TAA CGT GAT ATA ATA TCA CTT CCT AAA

$= + (9*64^4+16*64^3+57*64^2+33*64+4+4/64+28/64^2+53/64^3+61/64^4+0/64^5)$

$= 150994944 + 4194304 + 233472 + 2112 + 4 + 0.0625 + 0.0068359375 + 0.000202178955078125 + 0.0000036358833312988 28125$

$= 155424836.069541752338409423828125.$

3.4.1. A Better Representation for Real Numbers

An improvement for the representation of real numbers presented in previous section could be to add two more codons as follows: the two codons are added after the first nucleotide that represents the sign of the real number. These two codons are used to represent the number of codons in the integral part and the precision part respectively. This modified representation can be used to represent the above example as follows:

[Number of codons in the precision]

G ATT ATT AGT TAA CGT GAT ATA ATA TCA CTT CCT AAA

[Number of codons in integer part]

We prefer this modified representation of real numbers to the previous representation. In the later sections, whenever we refer to the DNA strand representation of a real number, unless otherwise mentioned, it is this improved method of representation that we refer to and not the one that is described in the *example 3.4.1* or prior to it. But we have come across the previous method for better understanding. However, it should be noted that, this representation puts a theoretical limit on the magnitude of numbers that could be represented. For instance, the fact that one codon is used to indicate the number of codons constituting the integer part implies that the integer part is at most 64^{63}-1. While representing higher magnitude numbers can be naturally achieved by increasing the number of codons to indicate how many codons will be used in integer part and precision, we believe that one codon as described above is sufficient for most practical purposes.

Example 3.4.2: Using the functions defined in (1) and (2) (see section 3.1 and example 3.4.1) and considering five significant digits in precision we have

150.96 = GAAGATTAAGTTGCCTTCAAGGACCTTC

Because

0.96*64=61.44	$61 \rightarrow$ CCT
0.44*64=28.16	$28 \rightarrow$ TCA
0.16*64=10.24	$10 \rightarrow$ AGG
0.24*64=15.36	$15 \rightarrow$ ACC
0.36*64=23.04	$23 \rightarrow$ TTC

3.4.2. Computer Algorithm

We summarize the number system described above into two computer algorithms. The first procedure *DNA-NS_DNA_to_NUMBER(DNA strand)* converts the *DNA strand* into its equivalent number.

The second procedure *DNA-NS_NUMBER_to_DNA(Number)* does the opposite; it converts the *Number* to its equivalent DNA strand representation.

3.4.2.1. Algorithm to convert a DNA strand into a Number

(a) DNA-NS_DNA_to_NUMBER (DNA strand)

```
L ← length of DNA strand;
If (L % 3 = 0) then
        Number ← Convert_to _integer (DNA strand);
else if (L % 3 = 1 and L > 1) then
        Number ← Convert_to_real (DNA strand)
else DNA strand is invalid
Exit
```

(b) Convert_to_integer (DNA strand)

```
    i ← 1
    num ← 0
    While (DNA stand [i] ≠ EOF)
        num = num * 4 + atoi (DNA strand[i])
        i ← i + 1
    sign(DNA strand, num)
    return (num)
```

(c) atoi (ch)

Input: A *DNF* / This procedure is given for *DNF* in (1). The return statements that appear in the subsequent lines in the procedure will change according to the *DNF* that is inputted /

```
    if (ch = 'A')
        return 0
    if (ch = 'T')
        return 1
    if (ch = 'G')
        return 2
    if (ch = 'C')
        return 3
```

(d) sign (DNA strand, num)

```
if (the first three characters in DNA strand are AAA)
then
    return( - num )
else
    return( num )
```

(e) Convert_to_real (DNA strand)

S ← *DNA strand* [1]

Cod1 ← the string formed by 2nd, 3rd and 4th characters in the *DNA strand*

Cod2 ← the string formed by 5th, 6th and 7th characters in the *DNA strand* For j ← 8 to (Cod1+7)

```
        Str1 [j – 7] ← DNA strand [j]
For k ← (Cod1+8) to (Cod1+Cod2+7)
        Str2 [k – Cod1 – 7] ← DNA strand [k]
A ← absolute value of (Convert_to_integer (Str1))
B ← Convert_to_precision (Str2)
Num ← Set_sign(A.B, S)
return (Num)
```

(f) Convert_to_precision (str)

```
i ← 1
den ← 4
sum ← 0
while (str [i] ≠ EOF)
        sum ← sum + atoi(str[i])/den
        den ← den * 4
return (sum)
```

(g) Set_sign (num, S)

Input: A *DSF* / This procedure is given for the *DSF* defined in Example 3.4.1/

```
If (S = 'G')
then return (num)
else return (- num)
```

3.4.2.2 Algorithm to convert a number into a DNA strand

(a) DNA-NS_NUMBER_to_DNA (Number)

```
    If Number is real
    then
        DNA strand ← Real_to_DNA (Number)
    If Number is integer
        then
    DNA strand ← Integer_to_DNA (Number)
```

(b) Integer_to_DNA (Number)

```
DNA strand ← Conversion (Number)
If (Number < 0) then
    DNA strand ← concatenation of ('AAA', DNA strand)
return (DNA strand)
```

(c) Conversion (Number)

```
i ← 1
j ← 1
num ← absolute value of Number
while (num > 0)
    r ← num % 4
    strand [j] ← itoc (r)
    j ← j + 1;
    num ← num/10
L ← (j – 1)   /the length of "strand" /
If (L%3 ≠ 0)
    For k ← 1 to L%3
        strand ← concatenation of ('A', strand)
return (strand)
```

(d) itoc(r)

/this procedure changes according to the *DNF* /

```
if (r = 0)
    return ('A')
if (r = 1)
    return ('T')
if (r = 2)
    return ('G')
if (r = 3)
    return ('C')
```

(e) Real_to_DNA (Number)

```
n ← the least possible number such that 64n ≥ Number
m ← the number of codons representing the precision digits
    required to be displayed (depends on the user's requirement)
DNA strand ← conversion (n)
DNA strand ← concatenation of (DNA strand, conversion (m))
Num ← integer part of Number
DNA strand ← concatenation of (DNA strand, conversion (Num))
F ← Number – Num
DNA strand ← concatenation of (DNA strand, findflo (F,m))
if (Number < 0)
then
        DNA strand ← concatenation of ('C', DNA strand)
else
        DNA strand ← concatenation of ('G', DNA strand)
```

(f) findflo (F, m)

```
for i ← 1 to m
        F ← F * 64
        R ← integral part of F
        Str ← concatenation of (Str, conversion (R))
        F ← F – R
return (Str)
```

4. POLYNUCLEOTIDE NUMBER SYSTEM

Another straight forward method for representing numbers as DNA strands is to find the base 4 equivalent of the number and to replace its digits by nucleotides using a *DNF*. Note that negative numbers can be represented as a DNA strand with its first nucleotide equivalent to zero in the *DNF* in use.

But with this representation, there is no way to distinguish between integers and real numbers, because such sequences could be of arbitrary length. In other words, the function $\boldsymbol{\mu}$ from the set of all DNA strands to the set of all real numbers defined by $\boldsymbol{\mu}$ (x) = N, where x is the DNA representation of N under the present representation scheme, is not one – one.

However, we shall call this straight forward number system as *polynucleotide number system (PN-NS)*. This may be useful in some situations where it is known that only integers or only real numbers are dealt with (for example see section 3.1 or 3.4).

4.1. Computer Algorithm for Polynucleotide Number System

Since we have already observed that the function $\boldsymbol{\mu}$ defined above is not one – one, we present only one algorithm that converts a given number into its polynucleotide representation.

(a) PN-NS_NUMBER_to_DNA (Number)

```
If Number is real then
    DNA strand ← Real_to_DNA (Number)
If Number is integer then
    DNA strand ← Integer_to_DNA (Number)
```

(b) Integer_to_DNA (Number)

```
DNA strand ← Conversion (Number)
If (Number < 0) then
    DNA strand ← concatenation of ('A', DNA strand)
return (DNA strand)
```

(c) Conversion (Number)

```
i ← 1
j ← 1
num ← absolute value of Number
while (num > 0)
    r ← num % 4
    strand [j] ← itoc (r)
    j ← j + 1;
    num ← num/10
return (strand)
```

(d) itoc(r)

```
/this procedure changes according to the DNF /
if (r = 0)
    return ('A')
if (r = 1)
    return ('T')
if (r = 2)
    return ('G')
if (r = 3)
    return ('C')
```

(e) Real_to_DNA (Number)

```
n ← the least possible number such that $4^n \geq$ Number
m ← the number of nucleotides (representing the precision) to be displayed (depends on the user's requirement )
DNA strand ← conversion (n)
DNA strand ← concatenation of (DNA strand, conversion (m))
Num ← integer part of Number
DNA strand ← concatenation of (DNA strand, conversion (Num))
F ← Number – Num
DNA strand ← concatenation of (DNA strand, findflo (F,m))
If (Number < 0)
then
    DNA strand ← concatenation of ('C', DNA strand)
else
    DNA strand ← concatenation of ('G', DNA strand)
```

(f) findflo (F, m)

```
for i ← 1 to m
    F ← F * 64
    R ← integral part of F
    Str ← concatenation of (Str, conversion (R))
    F ← F – R
return (Str)
```

5. AN EFFICIENT SEARCHING ALGORITHM USING DNA COMPUTATION

Several computational problems require a particular item to be searched from a given list of items. For example in a database, one has to search a number of times, particular record that could be identified uniquely by a key. There are several algorithms for searching [16].

The most basic method that works all the time is the *linear search.* When there is a list of "*n*" items, this algorithm performs the task in O(*n*) time. Another method is the *binary search,* which works when the given list is sorted either in ascending order or in descending order. This algorithm requires O(log *n*) time to search a particular item from a list of "*n*" items. Another method uses *binary search tree*. In this method the list of "*n*" items is organized into a binary search tree and searching takes O(log *n*) time.

Now, we present a method that performs the task in a unit time, once the input is given, irrespective of the input size. This method uses the novel idea of DNA annealing property.

5.1. DNA Algorithm for Searching

Suppose we have a set of numbers $S = \{a_1, a_2, a_3 \ldots a_k\}$ (where $k \geq 1$) and a target number *N*. Let *N* be the item to be searched in the set *S*. We present the new DNA algorithm to achieve the search below as a stepwise procedure.

Step 1: Encode the given numbers $\{a_1, a_2, a_3 \ldots a_k\}$ as follows:

The DNA strand corresponding to a_j ($1 \leq j \leq k$) consists of three (logical) parts, namely:

(1) The *Polynucleotide* representation of the number *j* (refer to the definition of a *Polynucleotide number* explained in *section 4*), which specifies the position of the number a_j in the list

(2) A special codon, "**GGG**" to recognize that the number is from the list (this is essential to prevent 'unnecessary' annealing of DNA strands corresponding to different numbers in the set *S*)

(3) The *DNA representation* of the number a_j.

The *Polynucleotide number* representing *j* is realized with the function

f(0) = C, f(1) = G, f(2) = A, f(3) = T.

The *DNA number* corresponding to *aj* is realized with the function

f(0) = A, f(1) = T, f(2) = G, f(3) = C.

And the strand corresponding to *aj* is the concatenation of all the three strands described above.

Example 5.1: Suppose that the task is to search the number 23 from the list of numbers 8, 2, 16, 23, 61, and 55. Table 5.1 shows encryption of each *aj*

Table 5.1
Encryptions of Each Number a_j

Given Number	*Position*	*Value*	a_j
$8 = a_1$	G	AGA	GGGGAGA
$2 = a_2$	A	AAG	AGGGAAG
$16 = a_3$	T	TAA	TGGGTAA
$23 = a_4$	GC	TTC	GCGGGTTC
$61 = a_5$	GG	CCT	GGGGGCCT
$55 = a_6$	GA	CTC	GAGGGCTC

Step 2: Synthesize DNA strand equivalent to the number *N* which is to be searched, with the function **f(0) = T, f(1) = A, f(2) = C, f(3) = G** and pad special codon **CCC** before the strand equivalent to the number *N*. This special codon works as an identification mark for the number to be searched. For the above example, *N* = 23 and hence the encoding of N is: *CCCAAG*.

Step 3: Pour all these strands into a 0.5 *ml* microfuge tube. Mix the components well, spin briefly, and incubate the mixture at 4 degrees centigrade to allow the annealing reaction take place. If the target strand finds a match, it binds to its complement to form a double strand.

For the above example, the following double strand would be formed.

GCGGGTTC
CCCAAG

Step 4: The so-formed double strand is separated by using hydroxyapatite column chromatography [17]. If double strand is not formed, then column chromatography will not give any result meaning that the item is not found. Success of the column chromatography experiment for the above example indicates the presence of the following double strand:

GCGGGTTC
CCCAAG

Step 5: To read the position of the item found, an enzyme called Taq DNA polymerase is added to the double strand. As a result, the full-length double strand will be formed. For the above example, the result of this step would be the formation of the following full-length double strand:

GCGGGTTC
CGCCCAAG

Step 6: Finally, the full-length double strand is sequenced and the position is read. In the above example, the position is read as GC which is equal to 4, which means the item is found at location 4 in the list.

Complexity: Step 3 is the place where searching is performed. The parallelism offered by DNA is highly exploited at this step. The DNA annealing is so powerful that the complementary strands bind together just in nanoseconds irrespective of the number of other strands present in the centrifuge. Each of the subsequent steps also require just a constant time irrespective of the number of items in the given set. Thus the above algorithm performs searching in a constant time.

6. A CLASSICAL CRYPTOSYSTEM USING DNA

6.1. Introduction to Cryptography

Cryptography is the study of coding messages [18] into a hidden form such that only some people who share a particular secret information with the sender will be able to decode the message and retrieve the actual information.

The actual message is called the *plain text* and the coded message is called the *cipher text*. The set of alphabets that constitute the plain text messages is denoted by ***P*** and that of cipher text messages is denoted by ***C***.

An *enciphering transformation* or a *cryptosystem* is a 1-1 function f: $\boldsymbol{P} \rightarrow \boldsymbol{C}$ that converts the plain text message into a cipher text. Note that, for every enciphering transformation the inverse function is well defined. There are many cryptosystems designed using conventional mathematics [18]. Now, we shall describe a new cryptosystem which is constructed on an entirely different background, namely DNA.

6.2. DNA Cryptosystem

Let us describe the cryptosystem below in multiple steps.

Step 1: Let the plain text be the English language with some punctuation marks namely ‘space’, ‘comma’ and ‘full stop’ etc.:

i.e., $\boldsymbol{P}$ = {A, B, C,…, Z, , , , . } (3)

Let the cipher text alphabet be the following set of 64 characters:

$\boldsymbol{C}$ = {A, B, C,…,Z, a, b, c,…,z,+,-,*,/,%,@,#,$,!,?,>,< }(4)

Associate the values 0, 1, 2… 63 with each of the characters in ***C*** respectively. This correspondence between the elements of ***C*** and numbers from 0 to 63 is shown in Table 6.2.1**.** In fact, the cryptosystem that we propose works well with the same set ***C*** (described in (4)) independent of the set ***P*** i.e. for any ***P*** we can follow the same method that is being described to convert the plain text message units to the cipher text over the alphabet ***C***. We have fixed ***P*** as in (3) just for the sake of convenience.

Table 6.2.1
The Correspondence between the Cipher Text Characters (Elements of C) and Numbers from 0 to 63

Element of C	*Number assigned*	*Element of C*	*Number assigned*
A	0	g	32
B	1	h	33
C	2	i	34
D	3	j	35
E	4	k	36
F	5	l	37
G	6	m	38
H	7	n	39
I	8	o	40
J	9	p	41
K	10	q	42
L	11	r	43
M	12	s	44
N	13	t	45
O	14	u	46
P	15	v	47
Q	16	w	48
R	17	x	49
S	18	y	50
T	19	z	51
U	20	+	52
V	21	-	53
W	22	*	54
X	23	/	55
Y	24	%	56
Z	25	@	57
a	26	#	58
b	27	$	59
c	28	!	60
d	29	?	61
e	30	>	62
f	31	<	63

Step 2: Assign DNA sequences to each element of ***P*** such that the length of each strand is a multiple of 3. Let us fix the length of our strands to 6 for the current discussion. At this step, it is mandatory that no two distinct elements of ***P*** are assigned the same DNA strand i.e., there should be a one – one correspondence between the elements of the set ***P*** and their corresponding DNA codes. The choice of DNA sequences that are assigned to elements of ***P*** is purely on the random basis. One such correspondence given in Table 6.2.2 is chosen for our discussion.

Table 6.2.2
Assignment of DNA Strands to the Plain Text Characters

Alphabet	*DNAcoding*	*Alphabet*	*DNAcoding*
A	GCGATA	P	GAGCGA
B	CGATTC	Q	CCGCAG
C	ATTAGC	R	TTTGGG
D	AGTACA	S	AGCTAG
E	ACGAAT	T	GGAATT
F	GATGCA	U	CCTTGG
G	GGCAAC	V	ACGTTT
H	CATCAT	W	GGTTAA
I	CTGGTT	X	TTCCGG
J	ATAGCT	Y	TGCATG
K	TATTCG	Z	CCCCCC
L	TCGCGA	,	TTTTTT
M	CAATAT	.	AGTAGT
N	CTTGAA		
O	AACACA	(SPACE)	CGCGCG

Encryption (Step 3 through Step 6)

Step 3: Now, collect all DNA strands corresponding to each character in the plain text message into a test tube to make a union. For instance, if the plain text message is "**MATHS**", then the test tube contains the DNA strands which code for M, A, T, H and S respectively. In our example this turns out to be the union of the strands, CAATAT, GCGATA, GGAATT, CATCAT, and AGCTAG.

Step 4: Now, run PCR (polymerase chain reaction) to get the complementary sequence [19] for each single strand present in the test tube. For the above example, the PCR results in obtaining the following double strands.

CAATAT GCGATA GGAATT CATCAT AGCTAG
GTTATA CGCTAT CCTTAA GTAGTA TCGATC

Step 5: Identify the codons in the complementary DNA strands (Table 6.2.3) from the double strands formed in step 4, and compute the numerical equivalent of each codon using the following one – one function:

$$\mathbf{f(A) = 0, f(T) = 1, f(G) = 2, f(C) = 3} \qquad (5)$$

Table 6.2.3
The Codons in the Complementary Strands Obtained in Step 4, their Numerical Equivalents, and the Equivalent Cipher Text Characters

codons	*Numericale quivalent*	*Corresponding Character*
GTT	37	l
ATA	4	E
CGC	59	$
TAT	17	R
CCT	61	?
TAA	16	Q
GTA	36	k
GTA	36	k
TCG	30	e
ATC	7	H

Step 6: The Cipher text is the sequence of characters of the set ***C*** numerically equivalent to the numbers obtained in the step 5. From Table 6.2.3 we get the cipher text as "***l*E$R?QkkeH**" corresponding to the above example.

6.3. Decryption

The person on the receiving end, on receiving the cipher text, follows the sequence of steps described below in order to decode the cipher text. This consists of reading the actual message and recognizing the numbers equivalent to each character in the cipher text. Then synthesizes the corresponding DNA codons, ligates them in that order, runs PCR, logically divides the complementary sequence into 6-base units (the length of the encoding strands) and retrieves the actual message by looking up the association between 6-base length DNA sequences and plain text characters. Table 6.3.1 shows the deciphering process. The result obtained is "**MATHS**" as required.

6.4. The Secret Key of the Cryptosystem

Table 6.2.1, Table 6.2.2 and (4) together constitute the secret key for the cryptosystem described above.

6.5. Security

The cryptosystem described above is not a public key cryptosystem. But one can increase the security of the cryptosystem by increasing the length of the DNA strands assigned to each character in ***P*** (That is, by increasing the length of the DNA strands in the Table 6.2.2). Because, if the length of the DNA strands is n, then the number of trials needed to correctly guess the DNA strand corresponding to a particular character is 4^n (in the worst case). So when n is increased, it would be very difficult for the intruders to guess the key, thus increasing the level of security of the cryptosystem.

Table 6.3.1
The Table Explaining the Deciphering Process

Cipher text	*l*	*E*	*$*	*R*	*?*	*Q*	*k*	*k*	*e*	*H*
Numerical Equivalents	37	4	59	17	61	16	36	36	30	7
Corresponding codons	**GTT**	**ATA**	**CGC**	**TAT**	**CCT**	**TAA**	**GTA**	**GTA**	**TCG**	**ATC**
After PCR:	**CAA**	**TAT**	**GCG**	**ATA**	**GGA**	**ATT**	**CAT**	**CAT**	**AGC**	**TAG**
After dividing into 6- base units	**CAATAT**		**GCGATA**		**GGAATT**		**CATCAT**		**AGCTAG**	
Plain text:	**M**		**A**		**T**		**H**		**S**	

6.6. Complexity

PCR generates the complementary strands of each DNA strand in parallel. This massive parallelism offered by DNA makes this cryptosystem more efficient over the general cryptosystems, which encode or decode messages letter by letter using some mathematical function (enciphering/ deciphering transformations). Especially this cryptosystem will be more appropriate when the message to be encoded or decoded is very lengthy.

7. CONCLUSION

In this paper, we presented a DNA number system to represent numbers in terms of DNA strands instead of numerical digits. We have described a search method that performs the task in a unit time, once the input is given, irrespective of the input size. Further, we proposed a classical cryptosystem in terms of DNA algorithm.

REFERENCES

[1] Adleman L. M: Molecular Computation of Solutions to Combinatorial Problems. *Science* 1994, **266**(11): 1021-1024.

[2] Lipton R. J.: DNA Solution of Hard Computational Problems. *Science* 1995, **268**, (5210): 542-545.

[3] Winfree E., Yang X., Seeman N. C.: Universal Computation via Self-assembly of DNA: Some Theory and Experiments. In: *Proceedings of DIMACS workshop: 1999 1996*: American Mathematical Society; 1996.

[4] Roweis S., Winfree E., Burgoyne R., Chelyapov N. V., Goodman M. F., Rothemund P. W. K., Adleman. L. M.: A Sticker-based model for DNA Computation. *Journal of Computational Biology* 1998, **5**(4): 615-629.

[5] Pãun G: DNA Computing Based on Splicing: Universality Results. *Theoretical Computer Science* 2000, **231**(2): 275-296.

[6] Okamoto A., Tanaka K., Saito I.: DNA Logic Gates. *Journal of American Chemical Society* 2004, **126**(30): 9458-9463.

[7] Wang X., Bao Z., Hu J., Wang S., Zhan A.: Solving the SAT Problem using a DNA Computing Algorithm based on Ligase Chain Reaction. *Biosystems* 2008, **91**(1): 117-125.

[8] Ignatova Z., Zimmermann K. H., Martínez-Pérez I: DNA Computing Models: Springer-Verlag New York Inc; 2008.

[9] Qian L., Winfree E.: A Simple DNA Gate Motif for Synthesizing Large-scale Circuits. *DNA Computing* 2009, 5347/2009: 70-89.

[10] Qian L., Soloveichik D., Winfree E.: Efficient Turing-Universal Computation with DNA Polymers. In: *The 16th International Conference on DNA Computing and Molecular Programming: June, 14-17 2010; Hong Kong University of School of Science and Technology, Hong Kong, China*; 2010.

[11] Liu X, Wang S: Development of an in Vivo Computer for SAT Problem. *Mathematical and Computer Modelling* 2010, In press.

[12] Krasnogor N.: Systems Self-assembly: Elsevier Science Ltd; 2008.

[13] Condon A., Harel D., Kok J. N., Salomaa A., Winfree E.: Algorithmic Bioprocesses; 2009.

[14] Wang X., Zhang Q.: DNA Computing-based Cryptography. In*: 2009*: IEEE; 2009: 1-3.

[15] Singh H., Chugh K., Dhaka H., Verma A. K.: DNA based Cryptography: an Approach to Secure Mobile Networks. *International Journal of Computer Applications IJCA* 2010, **1**(19): 82-85.

[16] Cormen T. H., Leiserson C. E., Rivest R. L., Stein C.: Introduction to Algorithms, Third Edition edn: The MIT Press; 2009.

[17] Lodish H. F., Berk A: Molecular Cell Biology: WH Freeman; 2008.

[18] Katz J., Lindell Y.: Introduction to Modern Cryptography: Chapman & Hall / CRC; 2008.

[19] Mullis K. B., Ferré F., Gibbs R. A.: The Polymerase Chain Reaction: Springer Science & Business; 1994.