****

Reducing Errors in DNA Computing by Appropriate Word Design.

Jesse M. Gray,[a] Tony G. Frutos,[a] A. Michael Berman,[b] Anne E. Condon,[c] Max G. Lagally [d] Lloyd M. Smith [a] and Robert M. Corn[a]

[a] Department of Chemistry, University of Wisconsin, Madison, WI 53706

[b] Department of Computer Science, Rowan College, Glassboro, NJ 08028

[c] Department of Computer Science, University of Wisconsin, Madison, WI 53706

[d] Department of Materials Science and Engineering, University of Wisconsin, Madison, WI 53706

Draft version date: 10/9/96

I. Introduction and Background

In a recent paper,[1] we have proposed to perform logical manipulations of large sets of data chemically by using the hybridization and enzymatic manipulation of DNA molecules attached to surfaces. In these experiments, combinatorial mixtures of DNA molecules are attached to a surface, and subsets of this mixture are identified by the hybridization adsorption of complementary DNA molecules. Enzymes are then used to destroy all unhybridized DNA, and the process is repeated until only a few DNA molecules representing the "solutions" to a mathematical problem remain on the surface. In order to store information in the attached DNA molecules, we have proposed to break up the data into "words" of 5 or 8 bits that are stored in short oligonucleotides (either 15mers or 16mers respectively). By linking sets of these words together, we can eventually form the very large combinatorial sets required in the calculations while perfecting the chemistry on a much smaller length scale. In this paper we address the problem of finding the best sequences for storing the data in these oligonucleotides.

In our first set of test experiments, we have used 15mers to store 5 bits of information. The sequence of the 15mers has the form 3'-FFFFFFvvvvvFFFF-5', where F is a fixed base (G,C,T or A) that is the same for every 15mer, and v is a variable base that can vary between either (G,C) or (A,T). In this manner, one bit of information can be stored at each variable base position. Since there are 5 variable base positions, there are a total of 2^5 or 32 possible molecules in the entire combinatorial set. The template for the sequence used in our first set of experiments is 3'-ATAACCcccaaTCCT-5' where c refers to a (G,C) variable position and "a" refers to an (A,T) variable position. The total GC content of the molecules in this set was fixed at 7/15 (47%) in order to make the hybridization thermodynamics similar for each member of the mixture.

What are the errors that can result due to the hybridization adsorption process? Our proposed computation strategy assumes that any member of this combinatorial set of molecules can be identified by hybridization to its perfect complement, i.e. the 15mer sequence that will form a DNA duplex in which all 15 bases are hydrogen bonded to a complementary base. We denote this pair of molecules as the "perfect match" or "15-base perfect match". If we generate a complete combinatorial set of 32 molecules corresponding to our template 3'-ATAACCcccaaTCCT-5', and a second complementary combinatorial set of 32 oligonucleotides 5'-TATTGGcccaaAGGA-3', then each of the 32 molecules

in the original combinatorial set will have one perfect match in the complementary set. For example, the 15mer 3'-ATAACCCCCAATCCT-5' will make a 15-base perfect match with the complement 5'-TATTGGGGGTTAGGA-3':

```
3'-ATAACCCCCAATCCT-5'   15-base perfect match
5'-TATTGGGGGTTAGGA-3'
```

However, there will also be the molecule 5'-TATTGGGGGATAGGA-3' in the complementary set that will form a "14-base partial match" or a "single base mismatch" with this molecule:

```
            *
3'-ATAACCCCCAATCCT-5'   14-base partial match (single base mismatch)
5'-TATTGGGGGATAGGA-3'
            *
```

And there will also be the molecule 5'-TATTGGGCGTAAGGA-3' in the complementary set that will form a "13-base partial match" or a "2-base mismatch" with this oligonucleotide:

```
           *  *
3'-ATAACCCCCAATCCT-5'   13-base partial match (2-base mismatch)
5'-TATTGGGCGTAAGGA-3'
           *  *
```

Will these partial matches introduce errors into the DNA calculations? The answer to this question depends upon the relative stabilities of the perfect and partial matches. We need to design sets of DNA molecules that have large free energy differences between the perfect and partial matches, and we need to test whether these differences are enough to prevent the incorrect labelling of surface-bound oligonucleotides. While in principle, we could have easily chosen a set of 32 different 15mers that had very few partial matches, the combinatorial sets of molecules that we have chosen here are easily generated on a DNA synthesizer. This ease of set generation is an advantage that becomes more important as the number of variable bits increases, and is expected to be extremely important in any the DNA computing methodology (see Ref. 1 for more details).

A second question that arises is how many of these partial matches exist in the combinatorial mixture? This question is complicated by the possibility of "slide matches", or the hybridization of two DNA 15mers not in registry:

```
3'-XXXXXXXXXXXXXXX-5'          a possible "slide match" configuration
      5'-XXXXXXXXXXXXXXX-3'
```

In the remainder of this paper we address the problem of how to select a DNA template sequence that minimizes the number of partial matches for a combinatorial set of 2^5 (32) 15mers and also for a combinatorial set of 2^8 (256) 16mers. In general, we find that some fraction of the partial matches in a combinatorial set cannot be avoided, whereas others can be minimized by the appropriate selection and arrangement of the fixed and variable bases.

II. Selection of a 5-bit combinatorial set of 15mers

The problems with distinguishing between partial matches and perfect matches for a combinatorial set of DNA molecules and its complement can be reduced by choosing a DNA word design that minimizes the number of partial matches for a given the combinatorial set. However, some of the partial matches cannot be

eliminated; these "inherent partial matches" are present in all combinatorial mixtures.

Inherent partial matches. Consider a 5-bit DNA template of the form 3'-GGGGGGaaaaaGGGG-5', and its combinatorial complementary template 5'-CCCCCCaaaaaCCCC-3'. For the moment we will ignore the possibilities of slide matches. If one member of the set is selected and compared with the 32 molecules in the complementary set, we will find one complementary molecule that forms a perfect match. We will also find 5 molecules that form single base mismatches (14-base partial matches), 10 molecules that form 2-base mismatches (13-base partial matches), 10 molecules that form 3-base mismatches (12-base partial matches), 5 molecules that form 4-base mismatches (11-base partial matches), and one molecule that forms a 5-base mismatch (10-base partial match). In general, there are 5!/(n!)(5-n)! n-base mismatches. Note that every complement forms at least a 10-base partial match since the 10 fixed base positions always line up properly. These 31 partial matches cannot be eliminated by word template design considerations, and will always be present in these combinatorial sets. We denote these partial matches as "inherent partial matches".

We have written a program (see Appendix I) that calculates the hybridization interactions of each member of a template with all of the molecules in both the complementary set as well as with the molecules in the original combinatorial set. All possible slide matches are included in the calculation. The results of the program for the template 3'-GGGGGGaaaaaGGGG-5' are shown in Table 1.

Table 1.
For DNAtemplate GGGGGGaaaaaGGGG:
Variable positions = 5; fixed positions = 10
Total strands expected = 32
Slides are taken into account.
29 matches are counted for each pair.

matches	total	ratio	inherent	s&c
0	22432	701	0	701
1	9632	301	0	301
2	7488	234	0	234
3	6208	194	0	194
4	6944	217	0	217
5	1056	33	0	33
6	768	24	0	24
7	768	24	0	24
8	896	28	0	28
9	768	24	0	24
10	800	25	1	24
11	672	21	5	16
12	448	14	10	4
13	320	10	10	0
14	160	5	5	0
15	32	1	1	0

The first column in the table is the number n of correctly matched base pairs in a given duplex, and the second column (labelled "total") is the number of times that n-base match occurs in the combinatorial set. The third column (labelled "ratio") is the second column divided by the number of perfect matches (in this case, 32), and gives the average number of occurrences of

each n-base mismatch for each molecule of the template. This number is divided into two types in the next two columns; the "inherent" column" tabulates the inherent partial mismatchs that we expect from the 5!/(n!)(5-n)! n-base mismatch pattern that we calculated above, and the "s&c" column tabulates the mismatches that arise due to slide and complement configurations of the DNA molecules. In general, slides usually reduce the number of matches for a given molecule, but in some cases they may instead increase the binding capacity of two strands by either increasing the number of mismatches or by altering the location of the mismatches.

Slide Matches. The last column in Table 1 lists the average number of n-base slide matches that will occur for a molecule in the combinatorial set. We define a quality parameter Q as the largest n-base partial match created by a slide configuration. From Table 1, the template GGGGGGaaaaaGGGG is found to have a Q=12. The lower the Q, the better the particular word template is at eliminating the contributions of slide matches. The template GGGGGGaaaaaGGGG is a particularly poor choice; the results for the template that we have used in our first set of experiments, 3'-ATAACCcccaaTCCT-5', is shown in Table 2 and has a Q=9.

Table 2.
For DNAtemplate ATAACCcccaaTCCT:
Variable positions = 5; fixed positions = 10
Total strands expected = 32
Slides are taken into account.
29 matches are counted for each pair.

matches	total	ratio	inherent	s&c
0	19434	607	0	607
1	9226	288	0	288
2	13796	431	0	431
3	7040	220	0	220
4	3668	114	0	114
5	2308	72	0	72
6	1800	56	0	56
7	848	26	0	26
8	210	6	0	6
9	34	1	0	1
10	36	1	1	0
11	160	5	5	0
12	320	10	10	0
13	320	10	10	0
14	160	5	5	0
15	32	1	1	0

How low a Q value is possible? For these 5-bit 15mers, there are a total of 4^10 x 2^5 = 33 million possible templates. Many of these can be eliminated if we require that the GC content remain fixed at approximately 50% (7/15 or 8/15). In Appendix II we describe an exhaustive search program examines all of the possible DNA templates in order to find those with the lowest Q values. For the 5-bit 15mer case, the lowest Q value that was found for a template with a GC content of approximately 50% was 6. A total of 554 such templates were identified; an example of one of these templates is 3'-AAACACaacccACCA-5'. The n-base match frequencies for this example are listed in Table 3.

Table 3.
For DNAtemplate AAACACaacccACCA:
Variable positions = 5; fixed positions = 10
Total strands expected = 32
Slides are taken into account.
29 matches are counted for each pair.

matches	total	ratio	inherent	s&c
0	14272	446	0	446
1	16384	512	0	512
2	10368	324	0	324
3	8192	256	0	256
4	5312	166	0	166
5	3072	96	0	96
6	768	24	0	24
7	0	0	0	0
8	0	0	0	0
9	0	0	0	0
10	32	1	1	0
11	160	5	5	0
12	320	10	10	0
13	320	10	10	0
14	160	5	5	0
15	32	1	1	0

In addition to the 554 templates with a Q=6, the exhaustive search identified 75,160 templates with a Q=7 and 1,272,816 templates with a Q=8. While the Q=6 templates are significantly better mathematically then those shown in Tables 1 and 2, it remains to be seen whether there are real chemical differences in the surface hybridization selectivity of these different templates. An additional step that must be taken before selecting a combinatorial set of molecules will be to identify and eliminate any oligonucleotides that are self-complementary or form hairpin loops. We are currently designing a series of surface adsorption experiments to examine both of these issues.

III. Selection of an 8-bit combinatorial set of 16mers

While the 5-bit 15mers are a fine test system for studying the effects of inherent partial matches on errors in the DNA calculations, the information content in these sets of molecules is not very large. We propose to increase the information content in the word molecules in our next set of experiments by using a 16mer word template with 8 variable base positions and 8 fixed base positions (resulting in 256 molecule combinatorial sets). To insure that the inherent partial base mismatches are unstable, we will place the variable positions in the middle of the molecule so that the template will have the form: 3'-FFFFvvvvvvvvFFFF-5'. We will also keep the GC content of the template fixed at 50% (8/16). Repeating the calculations that were performed in the last section on the 15mers, we find that the template 3'-GGGGaaaaaaaaGGGG-5' has a Q=13:

Table 4.
For DNAtemplate GGGGaaaaaaaaGGGG:
Variable positions = 8; fixed positions = 8
Total strands expected = 256
Slides are taken into account.

31 matches are counted for each pair.

matches	total	ratio	inherent	s&c
0	1432832	5597	0	5597
1	730112	2852	0	2852
2	628736	2456	0	2456
3	513024	2004	0	2004
4	295424	1154	0	1154
5	105472	412	0	412
6	68608	268	0	268
7	55296	216	0	216
8	52736	206	1	205
9	50176	196	8	188
10	45056	176	28	148
11	35840	140	56	84
12	25088	98	70	28
13	15360	60	56	4
14	7168	28	28	0
15	2048	8	8	0
16	256	1	1	0

Table 4 shows that for the 8-bit 16mer combinatorial set there are 256 perfect matches (as expected), and that the inherent n-base mismatches now range from n=1 to n=8 with an average frequency given by the formula 8!/(n!)(8-n)!. If we again use our template analysis program and search the 4^8 x 2^8 = 16 million possible 8-bit 16mer templates, we find that for templates with a 50% GC content the lowest Q possible is 7. There are only 8 such templates, and an example is 3'-ACAAccaccccaAACA-5':

Table 5.
For DNAtemplate ACAAccaccccaAACA:
Variable positions = 8; fixed positions = 8
Total strands expected = 256
Slides are taken into account.
31 matches are counted for each pair.

matches	total	ratio	inherent	s&c
0	1036288	4048	0	4048
1	697344	2724	0	2724
2	882688	3448	0	3448
3	736256	2876	0	2876
4	376832	1472	0	1472
5	195584	764	0	764
6	63488	248	0	248
7	9216	36	0	36
8	256	1	1	0
9	2048	8	8	0
10	7168	28	28	0
11	14336	56	56	0
12	17920	70	70	0
13	14336	56	56	0
14	7168	28	28	0
15	2048	8	8	0
16	256	1	1	0

Table 5 shows that although the lowest Q values obtained for the 5-bit and 8-

bit words are similar (Q=6 vs. Q=7), the inherent and slide matches are not as well separated in the 8-bit case as compared to the 5-bit case. This is due to the wider distribution of inherent partial matches in the 16mers. In addition to the 8 Q=7 templates, we find that for the 16mers there are 6,560 templates with Q=8 and 178,512 templates with Q=9. We next need to sort through a set of these templates for the selection of multiple word templates.

IV. Multiple Word Selection

The word format proposed in this paper will be extended to multiple words by varying the fixed base labels in the words. The use of multiple words will allow us to extend our combinatorial sets to the sizes required for the DNA computing applications. For example, using two linked 8-bit words allows us to create combinatorial sets of 2^16= 64K molecules, and five linked word sets will yield combinatorial sets of 2^40 = 10^12 molecules.

Selection of the appropriate word labels requires that we calculate the intra-word partial matches for a given set of word molecules. Here are some preliminary calculations comparing the two 8-bit 16mer templates 3'-AGAAccaccccaAAGA-5' (we will refer to the combinatorial set generated by this template as "w1") and the template 3'-TGTTccaccccaTTGT-5' (similarly, we will refer to the combinatorial set generated by this template as "w2"):

AGAAccaccccaAAGA compared to word template TGTTccaccccaTTGT

num	total	ratio	wc1	wr1	s1	wc2	wr2	s2
0	1907712	7452	0	1	4047	0	0	3404
1	1610752	6292	0	4	2720	0	0	3568
2	1853696	7241	0	6	3442	1	0	3792
3	1300480	5080	0	4	2872	8	0	2196
4	711680	2780	0	1	1471	28	0	1280
5	368640	1440	0	0	764	56	0	620
6	163584	639	0	0	248	70	1	320
7	60416	236	0	0	36	56	4	140
8	18176	71	1	0	0	28	6	36
9	6144	24	8	0	0	8	4	4
10	7680	30	28	0	0	1	1	0
11	14336	56	56	0	0	0	0	0
12	17920	70	70	0	0	0	0	0
13	14336	56	56	0	0	0	0	0
14	7168	28	28	0	0	0	0	0
15	2048	8	8	0	0	0	0	0
16	256	1	1	0	0	0	0	0

These two 8-bit 16mer templates were chosen from the 8 possible 16mer templates that we found with a Q=7 (the lowest possible Q for the 16mer case). Note that they have the same internal data bit structure: we have decided that for chemical reasons, we'd like to always keep the data bit structure the same for all the words in a set. Note that these two templates are also special in that the 8 fixed word bits are of the form XYXX...XXYX. This is a result of the single word calculations (all the Q=7 templates have this form). This additional symmetry makes the table a little easier to follow.

The table is similar to our previous tables in that the "ratio" column represents the average number of n-base matches that a particular template molecule will see. To generate this table, we compared the combinatorial set

w1 to four other combinatorial sets:

i) "wc1",the combinatorial set of the complementary template of w1, 5'-TCTTccaccccaTTCT-3',

ii) "wr1", the combinatorial set of the reversed template of w1,5'-AGAAaccccaccAAGA-3', (note: wr1 and w1 are actually the same set)

iii) "wc2", the combinatorial set of the complementary template of w2,5'-ACAAccaccccaAACA-3' and

iv) "wr2", the the combinatorial set of the reversed template of w2,5'-TGTTaccccaccTTGT-3' (note: wr2 and w2 are actually the same set).

The numbers in the column labeled "wc1" are the inherent partial matches with wc1 that we found previously in the single word case. The column labeled "wr1" contains the inherent (zero slide) partial matches with wr1 (the reversed template set); in the previous sections we included these with all of the intraword slides in an "s&c" column; we now list the intraword slides separately in the column labeled "s1". As we found in the previous section, this word has a intraword Q of 7 due to the slide matches in column s1.

Note that the inherent partial matches from wr1 form a combinatorial set from 0 to 4-base matches of the form 4!/n!(4-n!). This is because when we reverse the template and compare the data bits, we get four positions which can still hydrogen bond:

```
       * ** *
 w1:  ccacccca
wr1:  accccacc
       * ** *
```

Since none of the word bits match between the original template and its reversed complement (w1 and wr1), the combinatorial set of these inherent partial matches make in the best case a 4-base partial match.

When we compare our template molecules with the members of the complementary and reversed templates of the second word (wc2 and wr2, respectively), we find two more sets of inherent partial matches. The columns labeled "wc2" and "wr2" correspond to the inherent (zero slide) matches with wc2 and wr2. The numbers in the final column ("s2") are the slide matches of the original template set w1 with both wr2 and wc2. Note that the inherent partial matches of w1 with wc2 yield a 8!/n!(8-n!) combinatorial set that ranges from 2-base partial matches to 10-base partial matches. This is because two of the positions in the fixed word bits of wc2 match with w1:

```
       *       *
 w1:  AGAA...AAGA
wc2:  ACAA...AACA
       *       *
```

In column "wr2", the inherent partial matches again follow the pattern 4!/n!(4-n!) due to the matching of the regular and reversed data bits. However, these numbers now run from 6-base to 10-base partial matches due to the six matches of the fixed word bits in w1 and wr2:

```
      * **   ** *
 w1:  AGAA...AAGA
wr2:  TGTT...TTGT
```

```
      * **   ** *
```

Note that the Q for the two word set is 10, and is determined NOT from the interword slides (column s2), but from the inherent partial matches of w1 with wc2 and wr2. This Q is the lowest that we have found from the possible combinations of the various 8-bit 16mers with Q=7 -- as long as we force the internal data bit structure to be the same for both words. We expect the inherent interword partial matches to be even more important if we start comparing multiple words.

V. Conclusions

By using a simple program that counts the number of matches between each pair of oligonucleotides in a given set, we can choose our word design so as to reduce non-specific hybridization (false positives). For single word sets, a 5-bit 15mer template of the form 3'-AAACACaacccACCA-5' was found to give a Q= 6 and represents one of the best possible choices for a DNA template. For the 8-bit 16-mer, a template of the form 3'-ACAAccacccccaAACA-5' was found to be one of the best single word templates with a Q=7. A list of the 96 templates with a data bit structure of the form "ccacccca" and a Q of 7 or 8 is listed in Appendix III.

For multiple word selections, it was determined that in addition to slide matches, one must minimize the intraword inherent partial matches in order to achieve the best sets of DNA templates. For the case of two 8-bit 16mer templates, the pair: 3'-AGAAccacccccaAAGA-5' and 3'-TGTTccacccccaTTGT-5' where found to have a total Q=10.

In an upcoming paper, we will examine the best 5 word sets of 8-bit 16mers, and compare them with the best 8 word sets of 5-bit 15mers. Each of these multiple word templates can be used to generate combinatorial sets of 2^40 = 10^12 molecules, which approaches the total number of molecules bound to our planar surfaces in our DNA computing project.[1]

References

1. Qinghua Liu, Zhen Guo, Anne E. Condon, Robert M. Corn, Max. G. Lagally, Lloyd M. Smith, "A Surface-Based Approach to DNA Computation," Proceedings of the DIMACS Second Annual Meeting on DNA Based Computers, Princeton, June 1996.

Appendix I. N-base Partial Match Distribution Program

```
// DNA Word Toolkit
// Michael Berman, Rowan College of NJ
// berman@rowan.edu

// Modifications made by Jesse Gray
// University of Wisconsin at Madison
// jmgray@students.wisc.edu
// usage: dwt <template string> <options>, where template string is of the
// form:
//      capital A, C, T, G: this position has a fixed value as specified
//      lower case a: this position can be either A or T
//      lower case c: this position can be either C or G
//      Options include "off" (offsets off), "on" (offsets on), and
//      "best" (considers all possible matches but only records the best
```

```
//      match between each pair of strands).
//      By default, options are set to "on".

// A,T,G,C,a, and c are the only valid characters in a template.
// Note that the length of the template string must be <= maxDNALength
// defined below.

#include <string.h>
#include <ctype.h>
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include <fstream.h>

#define MAXWORDS 100
const int maxDNALength = 20; // longest DNA strand allowed
int allLower[MAXWORDS];  // if they're all lower, it works differently

// DNAstrand -- a class for representing and manipulating a strand of DNA

const char baseNameStr[] = "ACGT";
enum bases {A, C, G, T}; // note: the order matters!
// the following masks are used to pick out the relevant bits from the
// internal representation of a DNA strand
const unsigned long int mask[] = {0x0L, 0x3L, 0xfL, 0x3fL, 0xffL,
        0x3ffL, 0xfffL, 0x3fffL, 0xffffL,
        0x3ffffL, 0xfffffL, 0x3fffffL, 0xffffffL,
        0x3ffffffL, 0xfffffffL, 0x3fffffffL, 0xffffffffL};
class DNAstrand {
public:
    DNAstrand(); // construct an empty DNA strand
    DNAstrand(char * initString); // construct from a C string
    DNAstrand(DNAstrand &); // copy constructor
    void set(char * cString); // set to DNA string passed
    void print(); // send to cout
    void asCString(char * cString); // will fill in argument with
                                    // C String rep of DNA strand
    DNAstrand & complement(); // returns new DNAstrand complement of this
    DNAstrand & operator~(); // operator form of complement
    DNAstrand reverse(DNAstrand d); //reverses a dna strand
    int length(); // return length of a strand
    int matchCount(DNAstrand & d, int offset); // match a pair with offset
    int operator==(DNAstrand & d); // compare strands for equality

private:
    unsigned long int b;
    char len;
    int convert(char baseLetter);
};

/*int offset_toggle = 1;
int best_matches_toggle = 0;*/

DNAstrand::DNAstrand()
{
    b = 0;
    len = 0;
}
```

```
// return a new strand containing the bases in reverse order of
// the receiver
DNAstrand DNAstrand::reverse(DNAstrand d)
{
    //DNAstrand * d = new DNAstrand();
    d.len = len;
    d.b = 0;
    int pos;
    for (pos = 0; pos < len; pos++) {
        d.b = d.b << 2;
        d.b = d.b | ((b >> pos*2) & 3);
    }
    return d;
}


DNAstrand::DNAstrand(DNAstrand & d)
{
    b = d.b;
    len = d.len;
}


// given a character representing a base, find the appropriate binary
// representation (or exit(0) an exception if none)
int DNAstrand::convert(char baseLetter)
{
    int i;
    for (i = 0; i < 4; i++)
        if (baseLetter == baseNameStr[i])
            return i;
    cerr << "Bad base letter: " << baseLetter << endl;
    exit(0);
    return 0;
}

// construct from a string, using the set function
DNAstrand::DNAstrand(char * initString)
{
    set(initString);
}

// process the string a character at a time, converting to binary
// representation and adding to the word representing the strand
void DNAstrand::set(char * cString)
{
    b = 0;
    for (len = 0; len < maxDNALength && cString[len] != 0; len++) {
        b = b<<2;
        b = b | convert(cString[len]);
    }
    if (cString[len] != 0) {
        cerr << "Error: init string to DNAstrand constructor too long: "
                << cString << endl;
        exit(0);
    }
}

void DNAstrand::print()
```

```
{
    char DNAstring[maxDNALength+1];
    asCString(DNAstring);
    cout << DNAstring;
}

// return the character representation of the strand into the argument
// 'cString'. Note that the caller has the responsibility to allocate
// enough memory to hold the result, i.e. len+1 chars.
void DNAstrand::asCString(char * cString)
{
    cString[len] = 0;
    int spos;
    for (spos = 0; spos < len; spos++)
        cString[(len - spos) - 1] = baseNameStr[int(b >> spos*2) & 3];
}

// return a new strand containing the complement of the receiver.
// Because of the representation chosen, the complementary strand is
// represented by the bitwise complementary string
DNAstrand & DNAstrand::complement()
{
    DNAstrand * d = new DNAstrand();
    d->len = len;
    d->b = ~b;
    return *d;
}

DNAstrand & DNAstrand::operator~()
{
    return this->complement();
}

int DNAstrand::length()
{
    return len;
}

// matchCount: match with offset and report # of matches
// The offset is the number of positions to the right that
// the argument is offset relative to the strand receiving the
// request (this). A negative offset is an offset to the left (or
// equivalently, like offsetting "this" offset bases to the right, which
// is how it's actually computed.)
// You need to use the mask to avoid extraneous matches in portions of the
// word that are not in use.
// A match is performed by taking the exclusive or of the relavent ranges
// of the two words and then checking for '3' (i.e. 11 binary)
// You only check (len-|offset|) positions, because the others
// are outside the range actually containing data and can have
// extraneous matches that are meaningless
int DNAstrand::matchCount(DNAstrand & d, int offset) {

    unsigned long int match;
    if (offset >= 0)
        match = b ^ (d.b >> offset *2);
    else
        match = (b >> (-offset * 2)) ^ d.b;
    int count = 0;
```

```
    int pos;
    for (pos = 0; pos < len - abs(offset); pos++)
        if (((match >> pos * 2) & 3) == 3)
            count++;
    return count;
}

int DNAstrand::operator==(DNAstrand & d)
{
    return len == d.len &&
        ((b & mask[len]) == (d.b & mask[d.len]));
}

/***
rf() reads from a file of templates and feeds them to an array of characters
***/

char templateArray[100000][maxDNALength];

void rf()
{
    char tmp;
    ifstream inFile( "results", ios::in);
    int strandCounter = 0;
    int jcount = 0;

    if(!inFile) { // open failed
        cerr << "cannot open ''results'' for output\n";
        exit(-1);
    }
    inFile.get(tmp);
    while(tmp != 'Z'){
        jcount = 0;
 while(tmp != ' '){
            templateArray[strandCounter][jcount++] = tmp;
            inFile.get(tmp);
        }
        templateArray[strandCounter][jcount] = NULL;
        strandCounter++;
        inFile.get(tmp);

        while(tmp != '\n'){
            inFile.get(tmp);
        }
        inFile.get(tmp);
    }
    templateArray[strandCounter][0] = 'Z';
}




//************************************************

// dnaArray and compArray get created by the generate function,
// then recursiveGenerate fills in their values

int count = 0;
DNAstrand dnaArray[maxDNALength*maxDNALength][MAXWORDS];
```

```
// (must be large enough for 2^(number of data bits).

DNAstrand compArray[maxDNALength*maxDNALength*maxDNALength][MAXWORDS];
char *wordTemplate[MAXWORDS]; //to keep track of the templates
int len[MAXWORDS];  //the various lengths

// recursiveGenerate builds up the prefix, using the pattern
// in the DNAtemplate, until it has a complete instance of
// a DNA strand (represented as a string). At that point the
// DNAstrand is created and analyzed.

void recursiveGenerate(char * prefix, char * DNAtemplate, int dnaword) {
    int prefixLen = strlen(prefix);
    if (strlen(DNAtemplate) == 0) {
        // we've built a complete strand, so add to array
        DNAstrand d(prefix);
        dnaArray[count][dnaword] = d;
        compArray[count++][dnaword] = ~d;
        return;
    }
    if (isupper(DNAtemplate[0])) {
        // upper case elements represent fixed values
        prefix[prefixLen] = DNAtemplate[0];
        prefix[prefixLen+1] = 0;
        recursiveGenerate(prefix, DNAtemplate+1,dnaword);
    }
    else if (DNAtemplate[0] == 'a') {
        prefix[prefixLen] = 'A';
        prefix[prefixLen+1] = 0;
        recursiveGenerate(prefix, DNAtemplate+1,dnaword);
        prefix[prefixLen] = 'T';
        prefix[prefixLen+1] = 0;
        recursiveGenerate(prefix, DNAtemplate+1,dnaword);
    }
    else if (DNAtemplate[0] == 'c') {
        prefix[prefixLen] = 'C';
        prefix[prefixLen+1] = 0;
        recursiveGenerate(prefix, DNAtemplate+1,dnaword);
        prefix[prefixLen] = 'G';
        prefix[prefixLen+1] = 0;
        recursiveGenerate(prefix, DNAtemplate+1,dnaword);
    }
    else {
        cerr << "Bad char in template: " << DNAtemplate[0] << endl;
        exit(0);
    }
}

// calculate 2 to the nth power
unsigned int power2(int n)
{
    unsigned int x = 1;
    while (n--)
        x = x * 2;
    return x;
}

// generate every strand specified by the template, and its complement;
// put the generated strands in dnaArray and complements in compArray
```

```
int generate(char * DNAtemplate, unsigned int totalStrands[], int dnaword) {

    len[dnaword] = strlen(DNAtemplate);
    wordTemplate[dnaword]=DNAtemplate;
    char prefix[maxDNALength+1];
    prefix[0] = 0;
    if (len[dnaword] > maxDNALength) {
        cerr << "Template string too long." << endl;
        return 1;
    }

    cout << "For DNAtemplate " << DNAtemplate << ":" << endl;
    // determine how many distinct strands to expect
    int i;
    int counter = 0;
    allLower[dnaword] = 1;
    for (i = 0; i < len[dnaword]; i++)
        if (islower(DNAtemplate[i]))
            counter++;
        else
          allLower[dnaword] = 0;

    cout << "Variable positions = " << counter;
    cout << "; fixed positions = " << (len[dnaword] - counter) << endl;
    totalStrands[dnaword] = power2(counter);
    cout << "Total strands expected = " << totalStrands[dnaword] << endl;
    cout << (2*len[dnaword])-1 << " matches are counted for each pair." <<
endl;
    count = 0;
    recursiveGenerate(prefix, DNAtemplate, dnaword);
    return 0;
}

// crossCheck performs the test of every generated strand against every
// complementary strand, and reports the results

void crossCheck(unsigned int totalStrands[], int dnaword)
{
    int i, j, offset;
    DNAstrand tmpArray;
    int inherentBins[maxDNALength];

    // set up some empty bins to count the number of matches
    unsigned long int * countBins = new unsigned long int [len[dnaword]+1];

    for (i = 0; i <= len[dnaword]; i++){
        countBins[i] = 0;
        inherentBins[i] = 0;
    }

    // Perform the check.

    for (i = 0; i < totalStrands[dnaword]; i++) {
        for (j = 0; j < totalStrands[dnaword]; j++){
            for (offset = -(len[dnaword] - 1); offset <= len[dnaword] -
1;offset++){

countBins[dnaArray[i][dnaword].matchCount(compArray[j][dnaword],offset)]++;
```

```
countBins[dnaArray[i][dnaword].matchCount(dnaArray[j][dnaword].reverse(tmpArr
ay),offset)]++;
     }
        }
    }

    for (i = 0; i < totalStrands[dnaword]; i++) // subtract inherent matches
        for (j = 0; j < totalStrands[dnaword]; j++){

inherentBins[dnaArray[i][dnaword].matchCount(compArray[j][dnaword],0)]++;
        }
    cout << endl;
    // report the results
    cout << "matches\t         freq\tratio\tinherent\tslides\t\n";
    for (i = 0; i <= len[dnaword]; i++) {
        cout.width(0);
        cout << i << '\t';
        cout.width(13);
        cout << countBins[i] << '\t';
        cout.width(5);
        cout << countBins[i]/totalStrands[dnaword] << '\t';  //the near
complements for each strand
        cout.width(8);
 cout << inherentBins[i]/totalStrands[dnaword] << '\t';
 cout.width(6);
 cout << (countBins[i] - inherentBins[i])/totalStrands[dnaword];
        cout << endl;
    }
}

// check performs the test of every generated and complementary
// strand of one word against every g&c strand of another, and it
// reports the results.

void check(unsigned int totalStrands[], int word1, int word2)
{
    int i, j, offset,most_matches, dnaword, tmpMatch, tmp;
    DNAstrand tmpArray;

    if(len[word1] >= len[word2]) //use length of shortest strand
        dnaword = word2;
    else
        dnaword = word1;

    // set up some empty bins to count the number of matches
    unsigned long int * countBins = new unsigned long int[len[dnaword]+1];
    unsigned long int * inherentBins = new unsigned long int[len[dnaword]+1];

    for (i = 0; i <= len[dnaword]; i++){
        countBins[i] = 0;
 inherentBins[i] = 0;
    }

    // Perform the check. If "on" is specified, each strand is tested
    // against every complement strand, across all offsets.  If "best" is
    // specified, only the best match data for each pair of strands will be
    // stored.  If "off" is specified, no offsets (slides) will be
    // considered.
```

```
    if(allLower[word1] && allLower[word2])
        for (i = 0; i < totalStrands[word1]; i++) //compare dna arrays
            for (j = 0; j < totalStrands[word2]; j++)
                for (offset = -(len[dnaword] - 1); offset < len[dnaword];
offset++)

countBins[dnaArray[i][word1].matchCount(compArray[j][word2],offset)]++;

    else if(allLower[word1])
        for (i = 0; i < totalStrands[word1]; i++) //compare dna arrays
            for (j = 0; j < totalStrands[word2]; j++)
                for (offset = -(len[dnaword] - 1); offset <
len[dnaword];offset++){

countBins[dnaArray[i][word1].matchCount(dnaArray[j][word2].reverse(tmpArray),
offset)]++;

countBins[dnaArray[i][word1].matchCount(compArray[j][word2],offset)]++;
            }

    else if(allLower[word2])
        for (i = 0; i < totalStrands[word1]; i++) //compare dna arrays
            for (j = 0; j < totalStrands[word2]; j++)
                for (offset = -(len[dnaword] - 1); offset < len[dnaword];
offset++){

countBins[dnaArray[i][word1].matchCount(dnaArray[j][word2].reverse(tmpArray),
offset)]++;

countBins[compArray[i][word1].matchCount(dnaArray[j][word2],offset)]++;
                }
    else
        for (i = 0; i < totalStrands[word1]; i++) //compare dna arrays
            for (j = 0; j < totalStrands[word2]; j++)
                for (offset = -(len[dnaword] - 1); offset < len[dnaword];
offset++){

countBins[dnaArray[i][word1].matchCount(dnaArray[j][word2].reverse(tmpArray),
offset)]++;

countBins[compArray[i][word1].matchCount(dnaArray[j][word2],offset)]++;

countBins[dnaArray[i][word1].matchCount(compArray[j][word2],offset)]++;

countBins[compArray[i][word1].matchCount(compArray[j][word2].reverse(tmpArray
),offset)]++;
                }

        for (i = 0; i < totalStrands[word1]; i++) //compare dna arrays
            for (j = 0; j < totalStrands[word2]; j++){

inherentBins[dnaArray[i][word1].matchCount(dnaArray[j][word2].reverse(tmpArra
y),0)]++;
 // inherentBins[compArray[i][word1].matchCount(dnaArray[j][word2],0)]++;
  inherentBins[dnaArray[i][word1].matchCount(compArray[j][word2],0)]++;
//
inherentBins[compArray[i][word1].matchCount(compArray[j][word2].reverse(tmpAr
ray),0)]++;
            }
```

```
    // report the results
    cout << endl;
    cout << "For word template " << wordTemplate[word1] << " compared to word
template " << wordTemplate[word2] << ":" << endl;
    cout << "matches\t         freq\tinherent\tslides\t\n";
    for (i = 0; i <= len[dnaword]; i++) {
        cout.width(0);
        cout << i << "." << '\t';
        cout.width(13);
        cout << countBins[i] << '\t';
        cout.width(8);
 cout << inherentBins[i] << '\t';
 cout.width(6);
 cout << (countBins[i] - inherentBins[i]);
        cout << endl;
    }

}


int main(int argc, char **argv)
{
    int i, j, k,l;
    unsigned int totalStrands[MAXWORDS];

    if(argc == 1)
        rf();  // get the templates from a file
    else{
 for(i=0;i < (argc -1);i++){  // read in the templates
         // from the command line
            strcpy(templateArray[i],argv[i+1]);
        }
            templateArray[i][0] = 'Z';
    }

    l = 0;
    while(templateArray[l][0] != 'Z'){  // check each individual template
        generate(templateArray[l], totalStrands, l);
        crossCheck(totalStrands, l++);

    }

    for(i=0;i<l;i++)
        for(j=0;j<l;j++)
     if(i>j)
         check(totalStrands, j, i);
    return 0;
}
```

Appendix II. Template Q Evaluation and Exhaustive Search Program

```
// DNA Exhaustive Search Program
// Jesse Gray, UW-Madison: gracchus@corninfo.chem.wisc.edu
// Recursive Generate by Michael Berman, Rowan College: berman@rowan.edu

// To compile, do something like "CC exhaust.c -O -o exhaust"
// You may need to add a -lm flag, depending on the compiler is set up.
```

```
// This program takes a DNA "design" of a specified length and with
// specified positions for data bits and word bits (data bits in the middle)
// and returns the templates of a specified quality.

// usage: exhaust <w> <x> <y> <z>
// where w is the total number of base pairs
// x is the first bp to be a "data" bit (the first base is #1)
// y is the last bp to be a "data" bit
// z is the highest "Q" value to be printed.  Q is a measure of "quality"
// where a Q of 4 means that there are no two non-complementary
// strands in the solution set for which 5 base pairs are matched
// (except for inherent mismatches).

#include <string.h>
#include <ctype.h>
#include <iostream.h>
#include <stdlib.h>
#include <math.h>

unsigned long int count, totalStrands, examinedStrands, counter;
const int maxDNALength = 16;
int maxQ, len, firstvar, lastvar, found;

// calculate 2 to the nth power
unsigned int power2(int n)
{
    unsigned int x = 1;
    while (n--)
        x = x * 2;
    return x;
}

void reverse(char *reversedTemplate, char *prefixtmp)
{
    int i;
    for(i=0;i < len; i++){
        reversedTemplate[i] = prefixtmp[(len-1) - i]; // oops -- prefix is
modified
    }
    reversedTemplate[i] = NULL;
}

//*************************************************
// crossCheck performs the test of every generated template against it's
// complement template and returns the Q value, the g/c content, and the
// strand.

void crossCheck(char * prefix,int gc)
{
    int i, j, offset,maxMatch;
    char tmpStrand[2*len];
    int matchCount = 0;
    char prefixtmp[len];
    int n;

    // perform the check. Each strand is tested against half the possible
    // offsets, since checking across all offsets would be redundant.
```

```
    strcpy(tmpStrand,prefix);
    for(j=len;j<(2*len);j++){
        tmpStrand[j] = 'Z';
    }
    tmpStrand[j] = NULL;

    maxMatch = 0;
    for (offset = 1; offset <= len - 1; offset++){
        matchCount = 0;
        for (j=0;j<len;j++){

            if(tmpStrand[j] == tmpStrand[j+offset])
  matchCount++;

            else if((tmpStrand[j] == 'c' && (tmpStrand[j+offset] ==
            'C' || tmpStrand[j+offset] == 'G')) || ((tmpStrand[j] ==
            'C' || tmpStrand[j] == 'G') && tmpStrand[j+offset] == 'c'))
                matchCount++;

            else if((tmpStrand[j] == 'a' && (tmpStrand[j+offset] ==
            'A' || tmpStrand[j+offset] == 'T')) || ((tmpStrand[j] ==
            'T' || tmpStrand[j] == 'A') && tmpStrand[j+offset] == 'a'))
                matchCount++;

        }
        if(matchCount > maxMatch)
            maxMatch = matchCount;
    }

    examinedStrands++; // record that a strand was examined

    matchCount = 0;
    strcpy(prefixtmp,prefix); // 'cuz prefix would be modifed otherwise...

    if(maxMatch < maxQ+1){
  //then check reversed one too
        reverse(tmpStrand, prefixtmp);
        for(j=len;j<(2*len);j++){
            tmpStrand[j] = 'Z';
            prefix[j] = 'Z';
 }
        prefix[j] = NULL;
 tmpStrand[j] = NULL;
/*
cout << "prefix = " << prefix << endl;
cout << "tmpstrand = " << tmpStrand << endl;
*/
       for (offset = -(len-1); offset <= len - 1; offset++){
            matchCount = 0;
            for (j=0;j<len;j++){

  if((prefix[j] == 'c' && (tmpStrand[j+offset] == 'C' ||
  tmpStrand[j+offset] == 'G')) || ((prefix[j] == 'C' ||
  prefix[j] == 'G') && tmpStrand[j+offset] == 'c'))
                    matchCount++;

               else if((prefix[j] == 'a' && (tmpStrand[j+offset] == 'A'
  || tmpStrand[j+offset] == 'T')) || ((prefix[j] == 'T' ||
  prefix[j] == 'A') && tmpStrand[j+offset] == 'a'))
```

```
                        matchCount++;

  else if((prefix[j] == 'G' && tmpStrand[j+offset] == 'C')
  || (prefix[j] == 'A' && tmpStrand[j+offset] == 'T')
  || (prefix[j] == 'C' && tmpStrand[j+offset] == 'G')
  || (prefix[j] == 'T' && tmpStrand[j+offset] == 'A'))
   matchCount++;

  else if((prefix[j] == 'a' && tmpStrand[j+offset] =='a')
  || (prefix[j] == 'c' && tmpStrand[j+offset] == 'c'))
   matchCount++;
            }

            if(matchCount > maxMatch)
                maxMatch = matchCount;
     matchCount = 0;

            for (j=0;j<len;j++){

  if((tmpStrand[j] == 'c' && (prefix[j-offset] == 'C'
  || prefix[j-offset] == 'G')) || ((tmpStrand[j] == 'C' ||
  tmpStrand[j] == 'G') && prefix[j-offset] == 'c'))
                        matchCount++;

                else if((tmpStrand[j] == 'a' && (prefix[j-offset] == 'A'
  || prefix[j-offset] == 'T')) || ((tmpStrand[j] == 'T' ||
  tmpStrand[j] == 'A') && prefix[j-offset] == 'a'))
                        matchCount++;

  else if((tmpStrand[j] == 'G' && prefix[j+offset] == 'C')
  || (tmpStrand[j] == 'A' && prefix[j+offset] == 'T')
  || (tmpStrand[j] == 'C' && prefix[j+offset] == 'G')
  || (tmpStrand[j] == 'T' && prefix[j+offset] == 'A'))
   matchCount++;

  else if((tmpStrand[j] == 'a' && prefix[j+offset] == 'a')
  || (tmpStrand[j] == 'c' && prefix[j+offset] == 'c'))
   matchCount++;

            }
            if(matchCount > maxMatch)
                maxMatch = matchCount;
        }
    }
    prefix[len] = '\0'; // filter out the dummy characters.
    if(maxMatch < maxQ+1){
        cout << prefix << " Q=" << maxMatch << " gc=" << gc << endl;
        found++;
    }
}

// recursiveGenerate builds up the prefix until it has a complete instance of
// a DNA strand (represented as a string). At that point the
// DNA strand is passed to crossCheck to be examined.

void recursiveGenerate(char * prefix) {
    int gc = 0,i;
    if(counter > totalStrands)
        return;
```

```
    int prefixLen = strlen(prefix);
    if (strlen(prefix) == len) {
        // we've built a complete strand, so add to array
        counter++;
        for(i=0;i<len;i++){ // check gc content
            if(prefix[i] == 'G' || prefix[i] == 'c' || prefix[i] == 'C')
                gc++;
        }

        if(gc != len/2) // if not near 50%, don't analyze
         if(gc != ((len/2) +1))
                return;
     else
         if(!len%2)
             return;
        crossCheck(prefix,gc);
        return;
    }
    if((prefixLen < firstvar-1) || (prefixLen >= lastvar)){
        prefix[prefixLen] = 'A';
        prefix[prefixLen+1] = 0;
        recursiveGenerate(prefix);
        prefix[prefixLen] = 'T';
        prefix[prefixLen+1] = 0;
        recursiveGenerate(prefix);
        prefix[prefixLen] = 'C';
        prefix[prefixLen+1] = 0;
        recursiveGenerate(prefix);
        prefix[prefixLen] = 'G';
        prefix[prefixLen+1] = 0;
        recursiveGenerate(prefix);
    }
    else if(prefixLen >= firstvar-1 || prefixLen <= lastvar){
        prefix[prefixLen] = 'c';
        prefix[prefixLen+1] = 0;
        recursiveGenerate(prefix);
        prefix[prefixLen] = 'a';
        prefix[prefixLen+1] = 0;
        recursiveGenerate(prefix);
    }
}

int main(int argc, char **argv)
{

    if(argc > 4){
        len = atoi(argv[1]);
        firstvar = atoi(argv[2]);
        lastvar = atoi(argv[3]);
        maxQ = atoi(argv[4]);
    }
    else{
        cerr << endl;
        cerr << "Usage: " << argv[0] << " length_of_strand first_variable_bit
last_variable_bit maximum_Q_value_to_print" << endl;
        cerr << endl;
        return 1;
    }
```

```
    char prefix[maxDNALength*2];
    prefix[0] = 0;
    found = 0;
    count = lastvar - firstvar + 1;
    cerr << "Variable positions = " << count;
    cerr << "; fixed positions = " << (len - count) << endl;
    totalStrands = power2(len - count) * power2(len-count) * power2(count);
    cerr << "Total strands expected = " << totalStrands << endl;
    counter = 0;
    examinedStrands = 0;
    recursiveGenerate(prefix);

    cout << "Z" << endl;
    cout << "Program Done. " << examinedStrands << " strands examined." <<
endl;
    cout << "Found " << found << " strands with a g/c value of " << len/2;
    cout << " and a Q value of " << maxQ << " or lower." << endl;
    cout << endl;
    exit(0); // return here gives seg faults for 11-mers and above -- dunno
why yet...
}
```

Appendix III. 96 8-bit templates with Q <= 8 and the same data structure

```
AACCaacaaaacCGCC
AACGaacaaaacCCGC
AACGaacaaaacCGGC
AACGaacaaaacCGGG
AAGCaacaaaacGCCC
AAGCaacaaaacGCCG
AAGCaacaaaacGGCG
AAGGaacaaaacGCGG
ATCCaacaaaacCGCC
ATCGaacaaaacCGGG
ATGCaacaaaacGCCC
ATGGaacaaaacGCGG
ACCCaacaaaacACCC
ACCCaacaaaacCACC
ACCCaacaaaacCTCC
ACCCaacaaaacCCAC
ACCGaacaaaacCCGA
ACGCaacaaaacGCAC
ACGGaacaaaacGCAG
AGCCaacaaaacCGAC
AGCGaacaaaacCGAG
AGGCaacaaaacGGCA
AGGGaacaaaacAGGG
AGGGaacaaaacGAGG
AGGGaacaaaacGTGG
AGGGaacaaaacGGAG
TACCaacaaaacCGCC
TACGaacaaaacCGGG
TAGCaacaaaacGCCC
TAGGaacaaaacGCGG
TTCCaacaaaacCGCC
TTCGaacaaaacCCGC
TTCGaacaaaacCGGC
TTCGaacaaaacCGGG
TTGCaacaaaacGCCC
```

TTGCaacaaaacGCCG
TTGCaacaaaacGGCG
TTGGaacaaaacGCGG
TCCCaacaaaacTCCC
TCCCaacaaaacCACC
TCCCaacaaaacCTCC
TCCCaacaaaacCCTC
TCCGaacaaaacCCGT
TCGCaacaaaacGCTC
TCGGaacaaaacGCTG
TGCCaacaaaacCGTC
TGCGaacaaaacCGTG
TGGCaacaaaacGGCT
TGGGaacaaaacTGGG
TGGGaacaaaacGAGG
TGGGaacaaaacGTGG
TGGGaacaaaacGGTG
CATGaacaaaacCCCC
CACAaacaaaacCCGC
CACCaacaaaacACCC
CACCaacaaaacTCCC
CACCaacaaaacCCAC
CACCaacaaaacCCTC
CACCaacaaaacCGAC
CAGCaacaaaacGCAC
CAGGaacaaaacGCAG
CAGGaacaaaacGGAG
CTAGaacaaaacCCCC
CTCTaacaaaacCCGC
CTCCaacaaaacACCC
CTCCaacaaaacTCCC
CTCCaacaaaacCCAC
CTCCaacaaaacCCTC
CTCCaacaaaacCGTC
CTGCaacaaaacGCTC
CTGGaacaaaacGCTG
CTGGaacaaaacGGTG
CGGGaacaaaacAGAG
CGGGaacaaaacTGTG
GATCaacaaaacGGGG
GACCaacaaaacCCAC
GACCaacaaaacCGAC
GACGaacaaaacCGAG
GAGAaacaaaacGGCG
GAGGaacaaaacAGGG
GAGGaacaaaacTGGG
GAGGaacaaaacGCAG
GAGGaacaaaacGGAG
GAGGaacaaaacGGTG
GTACaacaaaacGGGG
GTCCaacaaaacCCTC
GTCCaacaaaacCGTC
GTCGaacaaaacCGTG
GTGTaacaaaacGGCG
GTGGaacaaaacAGGG
GTGGaacaaaacTGGG
GTGGaacaaaacGCTG
GTGGaacaaaacGGAG
GTGGaacaaaacGGTG

GCCCaacaaaacACAC
GCCCaacaaaacTCTC