Optimization of Exact Algorithms for Planted Motif Problems

Andrew Petersen\textsuperscript{1}, Edward Felekey\textsuperscript{2}

\textsuperscript{1}Loyola Marymount University
\textsuperscript{2}University of Connecticut

BioGrid Research Experience for Undergraduates, Summer 2013
Department of Computer Science and Engineering, University of Connecticut, Storrs, CT
apeter22@lion.lmu.edu, edward.felekey@uconn.edu

Abstract. Planted Motif Search is used to identify novel motifs in DNA and protein sequences. These motifs, relatively short sequences of DNA or amino acids that occur many times in a data set, are some of the building blocks for understanding protein structure and function, gene regulatory networks, and other cellular processes. Given the immense amount of biological data available today due to the many genome projects being pursued throughout the world, efficient processing algorithms and computational techniques are required to extract meaningful information. To ascertain the effectiveness of existing Planted Motif Search algorithms and improvements, the first algorithm (PMS1) presented in Rajasekaran et al. 2005, as well as the improvements they introduced for DNA sequence processing, were incrementally implemented and tested in run-time and memory usage.

Keywords: Planted Motif Search, Exact Algorithms, Computational Biology, Motifs

1 Introduction

The detection of similarities between different DNA sequences can provide insight into the function and significance of those sequences. One category of such similarities is the motif, a pattern that recurs across many DNA sequences. Motifs can represent meaningful structures such as transcription factor binding sites\textsuperscript{1}. Therefore, it is of interest to biologists and researchers to be able to quickly and accurately detect motifs in large collections of genetic data. The Planted Motif Search problem has been formalized to abstract the process of finding motifs.

Planted-\textit{(l,d)} Motif Search Problem Definition. The required inputs are: a list of \textit{t} sequences, each of length \textit{n}, to be examined for motifs; an integer \textit{l}, which specifies the length of the desired motif; and an integer \textit{d} specifying the Hamming distance permitted for any instance of the motif in an input sequence.
Hamming distance is defined as the number of positions at which two sequences differ. For example, the strings AAAT and CGTA are both at a Hamming distance of 2 from the string CGAT. The task is to output all motifs of length $l$ that have at least one variant in each of the $t$ sequences. A variant of a motif is any sequence that is a Hamming distance of exactly $d$ away from the motif. This problem can be modified to include all variants of motifs with Hamming distance $d$ or less.

This problem poses many challenges. Certain motifs are extremely subtle— that is, they appear with a frequency that is difficult to distinguish from random noise [2]. Even if all motifs can be distinguished, doing so exactly (that is, every time) requires greater than polynomial time and memory [3].

Numerous algorithms have been proposed and implemented to detect motifs in DNA sequences using a variety of approaches [4]. Algorithms such as MEME [5] and ProfileBranching [6] take a profile-based approach. They predict the positions at which the motif appears in each sequence. Other algorithms, like PairMotif [7], MITRA [8], and PROJECTION [9], predict the motif itself and are known as pattern-based algorithms.

Since the Planted Motif Search problem can be handled in so many ways, it is of interest to find techniques that will improve the performance of multiple motif search algorithms. For example, Rajasekaran and Dinh [10] presented a technique to speed up any Planted Motif Search algorithm by limiting the number of input sequences on which the algorithm actually needs to run.

This paper will present a number of proposed optimizations that can be applied in many approaches to the Planted Motif Search problem. While these modifications will not alter the asymptotic runtime of a given algorithm, they can provide considerable practical improvement over naïve approaches. The PMS1 algorithm and proposed improvements developed by Rajasekaran et al. are the framework on which these implementations are based. The optimizations are applied incrementally, so the improvements can be analyzed separately. These improvements do not only apply to this specific algorithm, but can be generalized to other computationally intensive algorithms that process sequences.

### 1.1 Methodology

All code was written in C++, due to its portability and speed when compared to other programming languages. The machines used to compile and run the code were running Ubuntu 10.04 LTS. The machines used to test the run-time of the different implementations had 7.7GiB of RAM and a 2.83GHz processor. To accurately record the execution time of the program, the cross-platform API OpenMP was used. To test the memory usage, the free, open-source software Valgrind was used. Valgrind’s purpose is to analyze and debug the memory usage of a program. It requires much more time and memory to run a program in Valgrind than it would to simply execute the compiled file. Therefore, Valgrind could not be run for large test cases.

All code is included in the appendix.

The biological data used to test the performance of the different implementations came from randomly selected pieces of DNA found through the LASAGNA-Search, an online tool used to find transcription factor binding sites that can be found at
http://biogrid-head.engr.uconn.edu/lasagna_search. The LASAGNA-Search algorithm was not used, but rather allowed access to real biological data. The data used is found in the appendix in FASTA format.

2 Naïve Implementation of the PMS1 Algorithm

To establish a baseline from which performance improvements can be measured, a naïve implementation of the PMS1 algorithm was written. This program represents each input sequence as a string of characters. It takes each string, extracts its $l$-mers, and generates a neighborhood around the $l$-mers by altering individual characters in each $l$-mer until all possible variants have been generated. Then each string’s neighborhood is sorted and duplicate $l$-mers are removed. Finally, all of the neighborhoods are combined into a single list of $l$-mers. If an $l$-mer occurs $t$ times in this final merged list, it must be found at least once in each input sequence. Such an $l$-mer would then be accepted as a motif.

3 Proposed Optimizations to PMS1

The naïve implementation, while functional, can be easily optimized in a number of ways. Simple, generalizable principles can be applied to reduce the practical memory requirements and run time of PMS1.

3.1 Integer Representation

Rather than using a character string representation, the integer data type can be used to encode the sequences of DNA as they are processed. Integers, and every other data type, are represented in binary code inside the processor of a computer. Efficiently using binary to encode only the required data greatly reduces the amount of space needed by the program. This encoding also allows powerful and fast low-level instructions to be used to isolate and manipulate characters. Because the sequences belong to a 40-character alphabet (ACTG), it is natural to have the $i^{th}$ position in the character string be represented by the $(2i)^{th}$ and $(2i+1)^{th}$ bits in the binary integer. As each character is represented by two bits this way, $l$-mers of length $w/2$ or less can be processed, where $w$ is the word length of the computer on which the program is run. In these experiments, both 32-bit and 64-bit word lengths were used.

The benefits of using low-level operations and optimizations are clear. Instructions that are hard-coded into the structure of a processor, such as bit-shift and exclusive-or, are very fast. If these instructions can be meaningfully used in a C++ program, the program will run much more quickly. For example, instead of multiplying a number by a factor of 2, it would be simpler to shift the bits that represent that number to the left once, provided that overflow is accounted for. Taking advantage of these sorts of inner workings of a processor has large benefits in the run-time and memory usage of a program.
Another benefit to accessing lower-level operations is that logical operations that would normally use an if-else statement, which uses expensive branch predictions, can be more easily replaced with logical operations done directly to the bits in the data.

3.2 Intersection

The naïve algorithm generates a list for each input string and then merges all of the lists into a master list of all possible motifs across all $l$-mers. This list necessarily contains duplicated motif candidates, as a motif is considered correct if it appears $t$ times in the master list. This requires a great deal of space in memory, much of which is used for candidate motifs that will simply be discarded, since many of them are incorrect. It is desirable to find a way that fewer incorrect motifs can be stored to reduce the memory footprint of the motif search program.

Rather than merge lists together, it is proposed that the lists instead be intersected. In this way, rather than set aside enough memory to hold all of the sublists combined, it is only necessary to hold two sublists in memory at once, and then store only those motifs that occur on both lists. As each new list is intersected, the collection of candidate motifs gets smaller and smaller. Only one instance of any particular motif needs to be stored in the intersection list. Therefore, this method will use much less memory than the naïve approach, and will in fact require smaller and smaller allocations of memory as the program runs. This method ought to also decrease the run-time, because instead of repeatedly merging lists with a growing list, lists are repeatedly intersected with a shrinking list. Both merging and intersecting are linear-time algorithms, so using the intersection method should decrease the run-time.

3.3 Customized Sorting Algorithm

In the non-optimized implementation, the sorting steps are performed using the C++ standard library sort. However, this algorithm is a general implementation that is intended to be used in a variety of programming contexts and is therefore not optimized for any one task. The utilization of a more efficient or specialized sorting algorithm will therefore improve the performance of any motif search algorithm for which sorting is required.

In the proposed optimization, a bucket sort was written to order the motifs. This gives an improved runtime compared to the naïve implementation of PMS1.

3.4 k-Split

The naïve algorithm creates a neighborhood from all of the $l$-mers present in each input sequence. These neighborhoods are very large and require considerable memory allocation to store. Reducing the size of each neighborhood would allow for a substantial reduction in memory space usage.
In this optimization, each input sequence is split into $k$ substrings of length \(((k-1)(l-1)+n)/k\). If there is a remainder for this number, those indices are distributed among the last substrings. The PMS1 algorithm is then run on each subdivision of the input; that is, PMS1 is run on the first substring of each input sequence, then on the second substring of each input sequence, and so on. PMS1 then collects any motif that appears $t/k$ times in any substring iteration and adds it to the merged list. This list is then checked against the original complete input to verify that each candidate motif is actually in all input sequences.

This improvement allows for a reduction in the space used by the algorithm.

### 3.5 Combination of Improvements

Many of the above improvements can be combined in a single program, giving a much greater practical improvement in runtime and memory usage. However, not all of these improvements are compatible.

The integer representation improvement is independent of any other technique used to speed up runtime or reduce memory usage. Therefore, its benefits can be compounded with any of the other techniques discussed. Likewise, any improvement in sorting can be applied without need to consider other improvements. However, $k$-split cannot be combined with intersection because the split lists would miss certain motifs if they were intersected.

### 4 Experimental Details

This section will detail the programs written and discuss the results obtained. All programs were run on real biological data obtained from the LASAGNA-Search website. For each test case, $t = 10$ and $n = 1000$. Except for the PMS1Split program, each program was run on each test case 3 times, and the average running time was reported. The PMS1Split program was only run twice for each case due to its extremely slow running time.

In Table 1, the abbreviation OOM stand for “Out Of Memory” and indicates that the program did not complete the challenge case due to having exceeded the memory capacity of the machine on which it was run.

<table>
<thead>
<tr>
<th>Program</th>
<th>(9, 2)</th>
<th>(11, 3)</th>
<th>(13, 4)</th>
<th>(15, 5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMS1String</td>
<td>11.937</td>
<td>203.088</td>
<td>OOM</td>
<td>OOM</td>
</tr>
<tr>
<td>PMS1Integer</td>
<td>1.84891</td>
<td>29.3325</td>
<td>432.068</td>
<td>OOM</td>
</tr>
<tr>
<td>PMS1Intersection</td>
<td>1.43107</td>
<td>23.0912</td>
<td>343.29</td>
<td>4860.03</td>
</tr>
<tr>
<td>PMS1CustomSort</td>
<td>1.39682</td>
<td>22.7992</td>
<td>341.834</td>
<td>OOM</td>
</tr>
<tr>
<td>64-bit PMS1CustomSort</td>
<td>1.36052</td>
<td>22.5084</td>
<td>336.82</td>
<td>OOM</td>
</tr>
</tbody>
</table>
4.1 PMS1String

This program is the naïve implementation of PMS1, using string representations, merge operations, and the C++ standard library sort. The data for this program in Table 1 and Table 2 are used as a baseline with which to compare the other programs.

4.2 PMS1Integer

The encoding of each input string as an unsigned binary integer was the single most significant improvement observed. The PMS1Integer program ran more than 6 times faster than the naïve implementation for the (9, 2) and (11, 3) challenge cases. In addition, PMS1Integer completed the (13, 4) challenge case, which the naïve implementation could not accomplish. Integer representation alone reduces the memory required for the PMS1 algorithm by more than a factor of 7.

4.3 PMS1Intersect

The PMS1Intersect program uses integer representations for the input sequences, and replaces the inefficient merge operation with intersection. This version of PMS1 used less memory than any other, requiring approximately 1/40 of the space of PMS1String for the (9, 2) case, as shown in Table 2. This allowed PMS1Intersect to complete the (15, 5) challenge case without exceeding the available 7.7GiB of RAM, which no other variant did. PMS1Intersect also improved the running time, finishing more than 8 times faster than PMS1String.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMS1String</td>
<td>127,354,880</td>
</tr>
<tr>
<td>PMS1Integer</td>
<td>17,732,744</td>
</tr>
<tr>
<td>PMS1Intersect</td>
<td>3,257,704</td>
</tr>
</tbody>
</table>

Table 2. Memory Usage (in bytes) by Specific Modifications in the (9, 2) Challenge Case
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PMS1CustomSort</td>
<td>4,543,344</td>
</tr>
<tr>
<td>64-bit PMS1CustomSort</td>
<td>9,067,200</td>
</tr>
</tbody>
</table>

### 4.4 PMS1CustomSort

The implementation of the custom bucket sort did not demonstrate a significant improvement to PMS1Intersect. PMS1CustomSort shows a slight improvement in running time that decreases as \( l \) and \( d \) increase. There is approximately a 3% improvement for the (9, 2) case when compared with PMS1Intersect, but this decreases to only about 1.3% for (11, 3) and only about 0.4% for (13, 4). This slight increase in speed is accompanied by a much larger increase in memory usage. PMS1CustomSort uses 39% more space than PMS1Intersect.

### 4.5 64-bit PMS1CustomSort

The 64-bit version of PMS1CustomSort behaved as might be expected, using twice as much memory as its 32-bit counterpart and running at a comparable speed, as shown in tables 1 and 2. Interestingly, the 64-bit version had a slightly shorter running time than the 32-bit version, taking between 0.5% and 3% less time, depending on the challenge case. It was expected that the 64-bit implementation would actually take slightly longer to run. This unexpected result is attributed to differences in the 32-bit and 64-bit compiling.

### 4.6 PMS1Split

PMS1Split used integer representations, merge operations, and the \( k \)-split technique intended to reduce memory requirements. This program was much slower than other programs studied, taking more than 60 times longer than PMS1String in all cases that it could complete. However, PMS1Split was unable to complete the (15, 5) case due to exceeding available memory, indicating that it did not bring with it a memory reduction factor commensurate to its slowdown.

### 5 Conclusions

This paper has presented a number of possible optimizations for Planted Motif Search algorithms. The most effective of these, integer representation of input sequences and intersection of candidate motif lists, greatly improve both running time and memory usage and can be easily applied to any PMS algorithm. The implementation of a bucket sort algorithm resulted in a minor increase in speed but a considerable increase in memory usage, and is not recommended for most cases. The \( k \)-split implementation
used in this research performs far worse than the naïve implementation of PMS1. It is possible that a different implementation of $k$-split might have better results, but the implementation used here is not recommended as an optimization.

Acknowledgments

This research was supported in part by the NSF grant OCI-1156837.

References

Appendix

/*
Andrew Petersen
Ted Felekey

motifFunctions.cpp
*/

#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <omp.h>
#include <cstring>
#include <stdint.h>
using namespace std;
const int W = 32;

//Accepts a character c and converts it to the corresponding integer (0, 1, 2, or 3)
inline int char_to_int(char c){
    char x = ~c;
    char y = x;
    x >>= 2;
    y = y & 2;
    x = x & 1;
    x += y;
    return x;
}

//Accepts a string (o_string) and converts it to an uint32_teger
uint32_t string_to_int(string o_string, int l){
    uint32_t o_int = 0;
    for(int i = 0; i < l; i++)
    {
        o_int += (char_to_int(o_string[i]) << (2*(l-i-1)));
    }
    return o_int;
}
/Accepts an int (o_int) and converts it to a string of length l
string int_to_string(uint32_t o_int, short int l){
    string sequence = "";
    for (int i = 0; i < l; i++){
        sequence += 'G';
    }
    for (int i = l-1; i >=0; i--){
        if(o_int !=0){
            if (o_int % 2 == 0){
                if (((o_int = (int)o_int/2) % 2) != 0){
                    sequence[i] = 'T';
                }
            }else{
                if (((o_int = (int)o_int/2) % 2) == 0){
                    sequence[i] = 'C';
                }else{
                    sequence[i] = 'A';
                }
            }
        }else{
            sequence[i] = 'A';
        }
    }
    o_int = (int)o_int/2;
    return sequence;
}

//Accepts a sequence represented as an integer (o_int) and converts the character at pos to the next character. That is G->C->T->A.
void increment_char(uint32_t &o_int, short int pos){
    uint32_t temp = o_int;
    temp <<= (W - (2*(pos+1)));
    temp >>= (W - 2);
    o_int += (((uint32_t)(((~4)*(temp==3) + 1)) << (2*pos));
}

//Generates all the possible motif candidates from a given sequence, represented as an uint32_t
void generate_motif_candidates(uint32_t c, vector<uint32_t> &n, int d, int pos, int length, int &count){
    if (d==0){
        n[count++] = c;
    }
    return;
for (int i=pos+1; i <= (length-d); i++){
    for (int j=0; j < 4-1; j++){
        uint32_t temp = c;
        for (int k = 0; k<=j;k++){
            increment_char(temp,i);
        }
        if(d>1) generate_motif_candidates(temp,n,d-1,i,length,count);
        else n[count++] = temp;
    }
    //call gen here for d<=hammingdistance
    //generate_motif_candidates(c,n,d-1,length,count);
    return;
}

//Eliminates any duplicate entries in an uint32_t vector
void eliminate_duplicates(vector<uint32_t> &n){
    n.erase(unique(n.begin(),n.end()),n.end());
}

//Merges two uint32_t vectors with each other, assumes both are already sorted
void merge_neighborhoods(vector<uint32_t> &n, vector<uint32_t> &destination, int l){
    vector<uint32_t> temp;
    temp.resize(destination.size()+n.size());
    merge(n.begin(),n.end(),destination.begin(),destination.end(),temp.begin());
    destination.resize(temp.size());
    for(int i = 0; i < temp.size(); i++)
        destination[i] = temp[i];
}

//Copies any uint32_t found at least x times in a vector into a new vector, motifs. Assumes first vector is sorted
void check_for_x_copies(vector<uint32_t> &merged, vector<uint32_t> &motifs, int x){
    motifs.resize(merged.size());
    int num_motifs = 0;
    int i = 0;
    while(i<merged.size()-1){
int count = 0;
while( (i<merged.size()-1) && (merged[i] == merged[i+1]) ){
    count++;
    i++;
}
if(count >= x-1){
    motifs[num_motifs++] = merged[i];
}
i++;
motifs.resize(num_motifs);
}

//Accepts two vectors of integers and stores their intersection in the second one (destination)
void intersect(vector<uint32_t> &n, vector<uint32_t> &destination){
    vector<uint32_t> temp;
    vector<uint32_t>::iterator it;
    temp.resize(destination.size());
    it=set_intersection(n.begin(),n.end(),destination.begin(),destination.end(),temp.begin());
    temp.resize(it-temp.begin());
    destination.resize(temp.size());
    for(int i = 0; i < temp.size(); i++)
        destination[i] = temp[i];
}

// NCR.cpp
// n choose r
long long NCR(int n, int r) {
    if(r > n / 2) r = n - r;
    long long ans = 1;
    int i;
    for(i = 1; i <= r; i++) {
        ans *= n - r + i;
        ans /= i;
    }
    return ans;
}

/*
 pms1string.cpp
 */
```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <cstring>
#include <omp.h>
#include <stdint.h>
#include "NCR.cpp"
using namespace std;

// Accepts a sequence as a string, converts the character at pos to the next character. That is G->C->T->A->G.
void increment_char(string &s, short int pos){
    // ACGT
    switch (s[pos]) {
        case 'A': s[pos] = 'C'; break;
        case 'C': s[pos] = 'G'; break;
        case 'G': s[pos] = 'T'; break;
        case 'T': s[pos] = 'A';
    }
}

// Generates all the possible motif candidates from a given string
void generate_motif_candidates(string c, vector<string> &n, int d, int pos, int length, int &count){
    if (d==0){
        n[count] = c;
        // cout << count << ": " << c << endl;
        count++; return;
    }
    for (int i=pos+1; i <= length-d; i++){
        for (int j=0; j < 4-1; j++){// 4 is the size of the alphabet
            string temp = c;
            for (int k = 0; k<=j;k++){
                increment_char(temp,i);
            }
            if(d>=1)
                generate_motif_candidates(temp,n,d-1,i,length,count);
        }
    }
    // call gen here for d<=hammingdistance, don't forget to modify nSize
```
void eliminate_duplicates(vector<string> &n) {
    n.erase(unique(n.begin(), n.end()), n.end());
}

void merge_neighborhoods(vector<string> &n, vector<string> &destination, int l) {
    string s;
    s.reserve(l);
    vector<string> temp;
    temp.resize(destination.size() + n.size(), s);
    merge(n.begin(), n.end(), destination.begin(), destination.end(), temp.begin());
    destination.resize(temp.size());
    for(int i = 0; i < temp.size(); i++)
        destination[i] = temp[i];
}

void check_for_x_copies(vector<string> &merged, vector<string> &motifs, int x) {
    motifs.resize(merged.size());
    int num_motifs = 0;
    int i = 0;
    while(i < merged.size() - 1) {
        int count = 0;
        while((i < merged.size() - 1) && (merged[i] == merged[i+1])) {
            count++;
            i++;
        }
        if(count >= x-1) {
            motifs[num_motifs++] = merged[i];
        }
        i++;
    }
    motifs.resize(num_motifs);
}
int main()
{
    // Init config file reading
    ifstream inf("config.dat");
    ifstream infileFile;
    string infile, temp;

    // Init timing variables
    double start, stop, sub_start, sub_stop;
    double t = 0.0;
    double sT = 0.0, eT = 0.0, mT = 0.0, gT = 0.0, cT = 0.0;

    // Declare/Init variables
    int l, d, n, count = 0;
    vector<string> sequences;
    vector<string> candidate_neighborhood,
        secondary_neighborhood, sorting_neighborhood;
    uint32_t n_size = 0;
    inf >> l >> d >> infile;

    // Read input sequences
    infileFile.open(infile.c_str());
    while(!infileFile.eof() && (infileFile >> temp))
    {
        sequences.push_back(temp);
    }
    infileFile.close();

    // Init neighborhood memory allocation info
    n = sequences[0].length();
    n_size = (n - l + 1)*(NCR(l,d))*(pow((double)3,d));

    // Starts timing.
    start = omp_get_wtime();

    sorting_neighborhood.resize(n_size);

    for (int i = 0; i < sequences.size(); i++)
    {
        sub_start = omp_get_wtime();
        secondary_neighborhood.resize(n_size);
        // Generate neighborhoods
        for (int j = 0; j <= n-l; j++)
        {
            generate_motif_candidates(sequences[i].substr(j,l),
                                      secondary_neighborhood, d, -l, l, count);
        }
    }
}
count = 0;
sub_stop = omp_get_wtime();
gT += (double) (sub_stop - sub_start);
//Sort
sub_start = omp_get_wtime();
sort(secondary_neighborhood.begin(), secondary_neighborhood.end());
sub_stop = omp_get_wtime();
sT += (double) (sub_stop-sub_start);
//Eliminate duplicates
sub_start = omp_get_wtime();
eliminate_duplicates(secondary_neighborhood);
sub_stop = omp_get_wtime();
eT += (double) (sub_stop-sub_start);
if(i==0)
{

candidate_neighborhood.resize(secondary_neighborhood.size ());
    for(int j = 0; j < secondary_neighborhood.size(); j++)
        candidate_neighborhood[j]=secondary_neighborhood[j];
    continue;
}
//Merge
sub_start = omp_get_wtime();
merge_neighborhoods(secondary_neighborhood, candidate_neighborhood, l);
sub_stop = omp_get_wtime();
mT += (double) (sub_stop-sub_start);
}
sub_start = omp_get_wtime();
vector<string> motifs;
check_for_x_copies(candidate_neighborhood, motifs, sequences.size());
sub_stop = omp_get_wtime();
cT += (double) (sub_stop-sub_start);
stop = omp_get_wtime();

ofstream outf("motifs.txt");
for (int i = 0; i < motifs.size(); i++)
{
    outf << motifs[i] << endl;
    //cout << motifs[i] << endl;
cout << "Motifs found: " << motifs.size() << endl;
cout << "Time Taken: " << t << " seconds" << endl;
cout.setf(ios::fixed,ios::floatfield);
cout << "Gen Time: " << gT << endl << "Sort time: "
<< sT << endl;
cout << "Elimination time: " << eT << endl << "Merge
time: " << mT << endl;
cout << "Check Time: " << cT << endl;
return 0;
}

#include "NCR.cpp"
#include "motifFunctions.cpp"

int main(){
    //Init config file reading
    ifstream inf("config.dat");
    ifstream infileFile;
    string infile, temp;

    //Init timing variables
    double start, stop, sub_start,sub_stop;
    double t = 0.0;
    double sT = 0.0, eT = 0.0, mT = 0.0, gT = 0.0, cT = 0.0;

    //Declare/Init variables
    int l, d, n, count = 0;
    vector<string> sequences;
    vector<uint32_t> candidate_neighborhood,
    secondary_neighborhood, sorting_neighborhood;
    uint32_t n_size = 0;

    inf >> l >> d >> infile;
    if(l>16)
    {
    }
cout << "'l'(ength) exceeds the word-length of this implementation";
    return 5;
}

//Read input sequences
infileFile.open(infile.c_str());
while(!infileFile.eof() && (infileFile >> temp))
{
    sequences.push_back(temp);
}
infileFile.close();
//Init neighborhood memory allocation info
n = sequences[0].length();
//for (int i=d; i>=0; i--)
n_size += (n - l + 1)*(NCR(l,d))*(pow(3.0,d));

//Starts timing.
start = omp_get_wtime();

sorting_neighborhood.resize(n_size);
for (int i = 0; i < sequences.size(); i++)
{
    sub_start = omp_get_wtime();
    secondary_neighborhood.resize(n_size);
    //Generate neighborhoods
    count = 0;
    for (int j = 0; j <= n - l; j++)
    {
        // for (int k = d; k >=0; k--)
        generate_motif_candidates(string_to_int(sequences[i].substr(j,l),l),secondary_neighborhood,d,-1,l,count);
    }
    sub_stop = omp_get_wtime();
    gT += (double) (sub_stop - sub_start); //Sort
    sub_start = omp_get_wtime();
    sort(secondary_neighborhood.begin(),
        secondary_neighborhood.end());
    sub_stop = omp_get_wtime();
    sT += (double) (sub_stop - sub_start);
    //Eliminate duplicates
    sub_start = omp_get_wtime();
    eliminate_duplicates(secondary_neighborhood);
    sub_stop = omp_get_wtime();
    eT += (double) (sub_stop - sub_start);
if(i==0)
{

candidate_neighborhood.resize(secondary_neighborhood.size());
    for(int j = 0; j < secondary_neighborhood.size(); j++)
    candidate_neighborhood[j]=secondary_neighborhood[j];
    continue;
}

//Merge
sub_start = omp_get_wtime();
merge_neighborhoods(secondary_neighborhood,candidate_neighborhood, l);
    sub_stop = omp_get_wtime();
    mT += (double) (sub_stop-sub_start);
}

//Check Merged List
sub_start = omp_get_wtime();
vector<uint32_t> motifs;
check_for_x_copies(candidate_neighborhood, motifs, sequences.size());
    sub_stop = omp_get_wtime();
    cT += (double) (sub_stop-sub_start);
stop = omp_get_wtime();

ofstream outf("motifs.txt");
for (int i = 0; i < motifs.size(); i++)
{
    outf << int_to_string(motifs[i],l) << endl;
}
outf.close();

    t = (double) (stop-start);
    cout << "Motifs found: " << motifs.size() << endl;
    cout << "Time Taken: " << t << " seconds" << endl;
    cout.setf(ios::fixed,ios::floatfield);
    cout << "Gen Time: " << gT << endl << "Sort time: " << sT << endl;
    cout << "Elimination time: " << eT << endl << "Merge time: " << mT << endl;
    cout << "Check Time: " << cT << endl;
return 0;
#include "NCR.cpp"
#include "motifFunctions.cpp"

int main()
{
    ifstream inf("config.dat");
infileFile.open(infile.c_str());
    string infile, temp;
    ifstream inf("config.dat");
infileFile.open(infile.c_str());
    string infile, temp;
    ifstream inf("config.dat");
infileFile.open(infile.c_str());
    string infile, temp;
    ifstream inf("config.dat");
infileFile.open(infile.c_str());
    string infile, temp;
    ifstream inf("config.dat");
infileFile.open(infile.c_str());
    string infile, temp;
    ifstream inf("config.dat");
infileFile.open(infile.c_str());
    string infile, temp;
    ifstream inf("config.dat");
infileFile.open(infile.c_str());
    string infile, temp;
    ifstream inf("config.dat");
infileFile.open(infile.c_str());
    string infile, temp;
    ifstream inf("config.dat");
infileFile.open(infile.c_str());
    string infile, temp;
    ifstream inf("config.dat");
infileFile.open(infile.c_str());
    string infile, temp;
    ifstream inf("config.dat");
infileFile.open(infile.c_str());
    string infile, temp;
    ifstream inf("config.dat");
infileFile.open(infile.c_str());
    string infile, temp;
    ifstream inf("config.dat");
infileFile.open(infile.c_str());
    string infile, temp;
    ifstream inf("config.dat");
infileFile.open(infile.c_str());
    string infile, temp;
}
n_size += (n - l + 1)*(NCR(l,d))*(pow(3.0,d));

//Starts timing.
start = omp_get_wtime();
candidate_neighborhood.resize(n_size);

for (int i = 0; i < sequences.size(); i++)
{
    sub_start = omp_get_wtime();
    secondary_neighborhood.resize(n_size);
    //Generate neighborhoods
    count = 0;
    for (int j = 0; j <= n-l; j++)
    {
        // for (int k = d; k >=0; k--)
        generate_motif_candidates(string_to_int(sequences[i].substring(j,l),l),secondary_neighborhood,d,-l,l, count);
    }
    sub_stop = omp_get_wtime();
    gT += (double) (sub_stop - sub_start);
    //Sort
    sub_start = omp_get_wtime();
    sort(secondary_neighborhood.begin(),
    secondary_neighborhood.end());
    sub_stop = omp_get_wtime();
    sT += (double) (sub_stop-sub_start);
    //Eliminate duplicates
    sub_start = omp_get_wtime();
    eliminate_duplicates(secondary_neighborhood);
    sub_stop = omp_get_wtime();
    eT += (double) (sub_stop-sub_start);

    if(i==0)
    {
        candidate_neighborhood.resize(secondary_neighborhood.size());
        for(int  j = 0;  j < secondary_neighborhood.size(); j++)
            candidate_neighborhood[j]=secondary_neighborhood[j];
        continue;
    }
    //Intersect
    sub_start = omp_get_wtime();
intersect(secondary_neighborhood,candidate_neighborhood);
    sub_stop = omp_get_wtime();
    iT += (double) (sub_stop-sub_start);
}
stop = omp_get_wtime();
ofstream outf("motifs.txt");
for (int i = 0; i < candidate_neighborhood.size(); i++)
{
    outf << int_to_string(candidate_neighborhood[i],l) << endl;
}
outf.close();

t = (double) (stop-start);
coutvero Motifs found: " << candidate_neighborhood.size() << endl;
cout "Time Taken: " << t " seconds" << endl;
cout "Gen Time: " << gT << endl "Sort time: " << sT << endl "Elimination time: " << eT << endl "Intersect time: " << iT << endl;
return 0;

/*
pmslcustomSort.cpp
*/
#include "NCR.cpp"
#include "motifFunctions.cpp"

//Sets the character at position pos in o_int to character ch
void set_char(uint32_t &o_int, short int pos, short int ch)
{
    uint32_t temp = o_int;
    temp <<= (W-(2*(pos+1)));
    temp >>= (W-2);
    o_int -= (temp << (2*pos));
    o_int += ((uint32_t)ch << (2*pos));
}
// Accepts a sequence represented as an integer (o_int) and returns the value at pos
inline uint32_t get_char(uint32_t o_int, short int pos){
  o_int <<= (W -(2*(pos+1)));
  o_int >>= (W - 2);
  return o_int;
}

// Sorts a vector of integers
void counting_sort(vector<uint32_t> &temp_n, vector<uint32_t> &n, int l){
  int s = n.size();
  int bucket[4];
  for (int i = 0; i < l; i++){
    memset(bucket,0, 4* sizeof(int));
    for (int j = 0; j< s; j++)
      bucket[get_char(n[j],i)]++;
    for(int j = 1; j < 4; j++)
      bucket[j]+= bucket[j-1];
    for (int j = s - 1; j >=0; j--)
      temp_n[--bucket[get_char(n[j],i)]] = n[j];
  }
}

int main()
{
  // Init config file reading
  ifstream inf("config.dat");
  ifstream infileFile;
  string infile, temp;

  // Init timing variables
  double start, stop, sub_start, sub_stop;
  double t = 0.0;
  double sT = 0.0, eT = 0.0, iT = 0.0, gT = 0.0;

  // Declare/Init variables
  int l, d, n, count = 0;
  vector<string> sequences;
  vector<uint32_t> candidate_neighborhood, secondary_neighborhood, sorting_neighborhood;
  uint32_t n_size = 0;
inf >> l >> d >> infile;

if(l>16)
{
    cout << "'l'(ength) exceeds the word-length of this implementation";
    return 5;
}

//Read input sequences
infileFile.open(infile.c_str());
while(!infileFile.eof() && (infileFile >> temp))
{
    sequences.push_back(temp);
}
infileFile.close();

//Init neighborhood memory allocation info
n = sequences[0].length();
//for (int i=d; i>=0;i--)
n_size += (n - l + 1)*(NCR(l,d))*(pow(3.0,d));

//Starts timing.
start = omp_get_wtime();
candidate_neighborhood.resize(n_size);
sorting_neighborhood.resize(n_size);

for (int i = 0; i < sequences.size(); i++)
{
    sub_start = omp_get_wtime();
    secondary_neighborhood.resize(n_size);
    //Generate neighborhoods
count = 0;
    for (int j = 0; j <= n - l; j++)
    {
        // for (int k = d; k >=0; k--)

        generate_motif_candidates(string_to_int(sequences[i].substr(j,l),l),secondary_neighborhood,d,-1,l, count);
    }
    sub_stop = omp_get_wtime();
gT += (double) (sub_stop - sub_start);
    //Sort
    sub_start = omp_get_wtime();
    counting_sort(sorting_neighborhood,secondary_neighborhood ,l);
    sub_stop = omp_get_wtime();
sT += (double) (sub_stop-sub_start);
//Eliminate duplicates
sub_start = omp_get_wtime();
eliminate_duplicates(secondary_neighborhood);
sub_stop = omp_get_wtime();
eT += (double) (sub_stop-sub_start);

if(i==0)
{

candidate_neighborhood.resize(secondary_neighborhood.size());
    for(int j = 0; j < secondary_neighborhood.size(); j++)
        candidate_neighborhood[j]=secondary_neighborhood[j];
    continue;
}
//Intersect
sub_start = omp_get_wtime();
intersect(secondary_neighborhood,candidate_neighborhood);
sub_stop = omp_get_wtime();
iT += (double) (sub_stop-sub_start);
}
stop = omp_get_wtime();
ofstream outf("motifs.txt");
for (int i = 0; i < candidate_neighborhood.size(); i++)
{
    outf << int_to_string(candidate_neighborhood[i],l) << endl;
}
outf.close();

t = (double) (stop-start);
cout << "Motifs found: " << candidate_neighborhood.size() << endl;
cout << "Time Taken: " << t << " seconds" << endl;
cout.setf(ios::fixed,ios::floatfield);
cout << "Gen Time: " << gT << endl << "Sort time: " << sT << endl << "Elimination time: " << eT << endl << "Intersect time: " << iT << endl;
return 0;
/**
 pms1_64.cpp
 */

#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <cstring>
#include <omp.h>
#include <stdint.h>
#include "NCR.cpp"

using namespace std;

const int W = sizeof(unsigned long)*8;

//Accepts a sequence represented as an integer (o_int) and converts the character at pos to the next character. That is G->C->T->A->G.
void increment_char(unsigned long &o_int, short int pos){
    unsigned long temp = o_int;
    temp <<= (W - (2*(pos+1)));
    temp >>= (W - 2);
    o_int += ((unsigned long)((((-4)*(temp==3) + 1)) << (2*pos));
}

//Sets the character at position pos in o_int to character ch
void set_char(unsigned long &o_int, short int pos, short int ch){
    unsigned long temp = o_int;
    temp <<= (W - (2*(pos+1)));
    temp >>= (W - 2);
    o_int -= (temp << (2*pos));
    o_int += ((unsigned long)ch << (2*pos));
}

//Accepts a sequence represented as an integer (o_int) and returns the value at pos
inline unsigned long get_char(unsigned long o_int, short int pos){
    o_int <<= (W - (2*(pos+1)));
    o_int >>= (W - 2);
    return o_int;
// Accepts a character c and converts it to the corresponding integer (0, 1, 2, or 3)
inline int char_to_int(char c){
    char x = ~c;
    char y = x;
    x >>= 2;
    y = y & 2;
    x = x & 1;
    x += y;
    return x;
}

// Accepts a string (o_string) and converts it to an unsigned long
unsigned long string_to_int(string o_string, int l){
    unsigned long o_int = 0;
    for(int i = 0; i < l; i++)
    {
        o_int += (char_to_int(o_string[i]) << (2*(l-i-1)));
    }
    return o_int;
}

// Accepts an int (o_int) and converts it to a string of length l
string int_to_string(unsigned long o_int, short int l){
    string sequence = "";
    for (int i = 0; i < l; i++)
    {
        sequence += 'G';
    }
    for (int i = l-1; i >=0; i--)
    {
        if(o_int !=0)
        {
            if (o_int % 2 == 0)
            {
                if (((o_int = (int)o_int/2) % 2) != 0)
                {
                    sequence[i] = 'T';
                }
            }
            else
            {
                if (((o_int = (int)o_int/2) % 2) == 0)
{  
  sequence[i] = 'C';
}  
else  
{  
  sequence[i] = 'A';
}  
}

o_int = (int)o_int/2;

return sequence;

//Generates all the possible motif candidates from a
given sequence, represented as an unsigned long
void generate_motif_candidates(unsigned long c,
vector<unsigned long> &n, int d, int pos, int length, int &count){  
  if (d==0){ n[count++] = c; return;}
  for (int i=pos+1; i <= (length-d); i++)  
  {  
    for (int j=0; j < 4-1; j++)//4 is the size of the
alphabet  
      {  
        unsigned long temp = c;
        for (int k = 0; k<=j;k++)
          {  
            increment_char(temp,i);
          }
        if(d>1) generate_motif_candidates(temp,n,d-1,i,length,count);
        else n[count++] = temp;
      }
  }  
  //call gen here for d<=hammingdistance
  //generate_motif_candidates(c,n,d-1,-1,length,count);
  return;
}

//Sorts a vector of unsigned long
void counting_sort(vector<unsigned long> &temp_n,
vector<unsigned long> &n, int l){  
  int s = n.size();
  int bucket[4];
  for (int i = 0; i < l; i++)  
  {  
    memset(bucket,0, 4* sizeof(int));
    }
for (int j = 0; j < s; j++)
    bucket[get_char(n[j], i)]++;
for (int j = 1; j < 4; j++)
    bucket[j] += bucket[j - 1];
for (int j = s - 1; j >= 0; j--)
    temp_n[--bucket[get_char(n[j], i)]] = n[j];
for (int j = 0; j < s; j++)
    n[j] = temp_n[j];
}

// Eliminates any duplicate entries in an unsigned long vector
void eliminate_duplicates(vector<unsigned long> &n)
{
    n.erase(unique(n.begin(), n.end()), n.end());
}

// Accepts two vectors of unsigned long and stores their intersection in the second one (destination)
void intersect(vector<unsigned long> &n, vector<unsigned long> &destination)
{
    vector<unsigned long> temp;
    vector<unsigned long>::iterator it;
    temp.resize(destination.size());
    it = set_intersection(n.begin(), n.end(), destination.begin(), destination.end(), temp.begin());
    temp.resize(it - temp.begin());
    destination.resize(temp.size());
    for (int i = 0; i < temp.size(); i++)
        destination[i] = temp[i];
}

int main()
{
    // Init config file reading
    ifstream inf("config.dat");
    ifstream infileFile;
    string infile, temp;
    // Init timing variables
    double start, stop, sub_start, sub_stop;
    double t = 0.0;
    double sT = 0.0, eT = 0.0, iT = 0.0, gT = 0.0;
    // Declare/Init variables
    int l, d, n, count = 0;
vector<string> sequences;
vector<unsigned long> candidate_neighborhood,
secondary_neighborhood, sorting_neighborhood;
unsigned long n_size = 0;
inf >> l >> d >> infile;
if(l>(W/2))
{
    cout << '"' << l << "' (length) exceeds the word-length of this implementation" << endl;
    return 5;
}
//Read input sequences
ifstream.open(infile.c_str());
while(!ifstream.eof() && (ifstream >> temp))
{
    sequences.push_back(temp);
}
ifstream.close();
//Init neighborhood memory allocation info
n = sequences[0].length();
//for (int i=d; i>=0;i--)
n_size += (n - l + 1)*(NCR(l,d))*(pow(3.0,d));
//Starts timing.
start = omp_get_wtime();
candidate_neighborhood.resize(n_size);
sorting_neighborhood.resize(n_size);
for (int i = 0; i < sequences.size(); i++)
{
    sub_start = omp_get_wtime();
    secondary_neighborhood.resize(n_size);
    //Generate neighborhoods
    count = 0;
    for (int j = 0; j <= n - l; j++)
    {
        // for (int k = d; k >=0; k--)
        generate_motif_candidates(string_to_int(sequences[i].substr(j,l),1),secondary_neighborhood,d, -l, l, count);
    }
    sub_stop = omp_get_wtime();
gT += (double) (sub_stop - sub_start);
//Sort
sub_start = omp_get_wtime();
counting_sort(sorting_neighborhood,secondary_neighborhood,1);
sub_stop = omp_get_wtime();
sT += (double) (sub_stop-sub_start);
//Eliminate duplicates
sub_start = omp_get_wtime();
eliminate_duplicates(secondary_neighborhood);
sub_stop = omp_get_wtime();
eT += (double) (sub_stop-sub_start);
if(i==0)
{

candidate_neighborhood.resize(secondary_neighborhood.size());
for(int j = 0; j < secondary_neighborhood.size(); j++)
candidate_neighborhood[j]=secondary_neighborhood[j];
continue;
}
//Intersect
sub_start = omp_get_wtime();
intersect(secondary_neighborhood,candidate_neighborhood);
sub_stop = omp_get_wtime();
iT += (double) (sub_stop-sub_start);
}
stop = omp_get_wtime();
ofstream outf("motifs.txt");
for (int i = 0; i < candidate_neighborhood.size(); i++)
{
    outf << int_to_string(candidate_neighborhood[i],l) << endl;
}
outf.close();

t = (double) (stop-start);
cout "Motifs found: " << candidate_neighborhood.size() << endl;
cout "Time Taken: " << t << " seconds" << endl;
cout.setf(ios::fixed,ios::floatfield);
cout << "Gen Time: " << gT << endl << "Sort time: " << sT << endl << "Elimination time: " << eT << endl << "Intersect time: " << iT << endl;
return 0;
}

#include "NCR.cpp"
#include "motifFunctions.cpp"

//returns the hamming distance between two l-mers, represented by two uint32_t
int hamming_distance(uint32_t a, uint32_t b, int l){
    int h = 0;
a^=b;
    for(int i = 0; i < l; i++)
    {
        uint32_t tempa = a;
        tempa <<= (W -(2*(i+1)));
        tempa >>= (W - 2);
        h+= (tempa>0);
    }
    return h;
}

//returns true if the given sequence is found in all the input sequences, otherwise returns false
bool naive_check(uint32_t s, vector<string> &sequence, int l, int d){
    for(int k=0; k< sequence.size();k++){
        bool found = false;
        for(int j= 0; j < sequence[k].length()-l+1; j++)
        {
            int h = hamming_distance(s,string_to_int(sequence[k].substr(j,l),l),l);
            if (h==d){
                found = true;
                break;
            }
        }
        if(!found){
            return false;
        }
    }
    return false;
}
int main(int argc, char* argv[]) {
    // Init config file reading
    ifstream inf("config.dat");
    ifstream inFileFile;
    string inFile, temp;

    // Init timing variables
    double start, stop, sub_start, sub_stop;
    double t = 0.0;
    double sT = 0.0, eT = 0.0, mT = 0.0, gT = 0.0, cT = 0.0, nT = 0.0;

    // Declare/Init variables
    int l, d, n, count = 0;
    vector<string> sequences;
    vector<uint32_t> candidate_neighborhood, secondary_neighborhood, temp_neighborhood, to_check, motifs;
    uint32_t n_size = 0;
    int k = 3; // default value for k is 3
    if(argc == 2) { k = atoi(argv[1]); }

    inf >> l >> d >> inFile;
    if(l > 16)
    {
        cout << "'l'(ength) exceeds the word-length of
        this implementation" << endl;
        return 5;
    }

    // Read input sequences
    inFileFile.open(inFile.c_str());
    while(!inFileFile.eof() && (inFileFile >> temp))
    {
        sequences.push_back(temp);
    }
    inFileFile.close();
    // Init neighborhood memory allocation info
    n = sequences[0].length();

    // Starts timing.
start = omp_get_wtime();

int vSize = (k-1.0)*(l-1.0)+n;
double pieceSize = (double)(vSize)/(k*1.0);
int rem = vSize % k;

for(int p=0; p<k; p++){
    int start = p*(int)(pieceSize-(l-1)) + (p>=k-rem)*(p+(rem-k));
    int len;
    if(p>=(k-(rem))){
        len = pieceSize + 1;
    }else{
        len = pieceSize;
    }
    n_size = (len - l + 1)*(NCR(l,d))*(pow(3.0,d));
    for (int i = 0; i < sequences.size(); i++)
    {
        sub_start = omp_get_wtime();
        secondary_neighborhood.resize(n_size);
        //Generate neighborhoods
        count = 0;
        for (int j = 0; j < len-l+1; j++)
        {
            generate_motif_candidates(string_to_int(sequences[i].substr(start,len).substr(j,l)),secondary_neighborhood,d, -1,l, count);
        }
        sub_stop = omp_get_wtime();
        gT += (double) (sub_stop - sub_start);
        //Sort
        sub_start = omp_get_wtime();
        sort(secondary_neighborhood.begin(),
        secondary_neighborhood.end());
        sub_stop = omp_get_wtime();
        sT += (double) (sub_stop-sub_start);
        //Eliminate duplicates
        sub_start = omp_get_wtime();
        eliminate_duplicates(secondary_neighborhood);
        sub_stop = omp_get_wtime();
        eT += (double) (sub_stop-sub_start);
    }
}

if(i==0)
{
  candidate_neighborhood.resize(secondary_neighborhood.size ());
}
for(int j = 0; j < secondary_neighborhood.size(); j++)
candidate_neighborhood[j]=secondary_neighborhood[j];
continue;
}
//Merge
sub_start = omp_get_wtime();
merge_neighborhoods(secondary_neighborhood, candidate_neighborhood, l);

sub_stop = omp_get_wtime();
mT += (double) (sub_stop-sub_start);
}
sub_start = omp_get_wtime();
check_for_x_copies(candidate_neighborhood, temp_neighborhood, ceil(sequences.size() / k));
sub_stop = omp_get_wtime();
cT += (double) (sub_stop-sub_start);
if(p==0){
    to_check.resize(temp_neighborhood.size());
    for(int j = 0; j < temp_neighborhood.size(); j++)
        to_check[j]=temp_neighborhood[j];
}else{
    merge_neighborhoods(temp_neighborhood, to_check, l);
    eliminate_duplicates(to_check);
}
sub_start = omp_get_wtime();
//Naive check
motifs.resize(to_check.size());
int motif_count = 0;
for(int i=0; i<to_check.size(); i++){
    if(naive_check(to_check[i], sequences, l, d)){
        motifs[motif_count] = to_check[i];
        motif_count++;
    }
}
motifs.resize(motif_count);
sub_stop = omp_get_wtime();
nT += (double) (sub_stop-sub_start);
stop = omp_get_wtime();
ofstream outf("motifs.txt");
for (int i = 0; i < motifs.size(); i++)
{
    outf << int_to_string(motifs[i],l) << endl;
}
outf.close();

t = (double) (stop - start);
cout << "Motifs found: " << motifs.size() << endl;
cout << "Time Taken: " << t << " seconds" << endl;
cout.setf(ios::fixed,ios::floatfield);
cout << "Gen Time: " << gT << endl << "Sort time: " << sT << " seconds" << endl;
cout << "Elimination time: " << eT << " seconds" << endl;
cout << "Merge time: " << mT << " seconds" << endl;
cout << "Check Time: " << cT << " seconds" << endl;
cout << "Naive Check Time: " << nT << " seconds" << endl;
return 0;

Biological Data Used, from the LASAGNA-Search Tool:

>gi|224589811:159824146-159825145 Homo sapiens chromosome 2, GRCh37.p10 Primary Assembly
GTCAACTAGATCAACGAAGGCTGTCTTTAGAGAGCTTCGGGTATTAAGTGAGCAGA
GTCCCTTAGGCCAATTTGGCTGGGAGGATCTTGGAGAACCCAG
ACACCCGAAATCCCTAGTCCCCATCTCTGTTTTTCGCTCTGCTGAGATAAACAACA
TTTTGGGAACTCAGGAGTTTAATTTACTGCCACCAACCGACAAAAATTAATAGCATTCT
TGCTGTGTTTCTCTCCCTCCTCGCCCTCCCTCCTACTCTCCTCCTCAGACAGGCTCGT
CCCTCTGTGAATGTAGCGCAGACAGCACGGGCAGCCCGGAGCGGTCGCAGGAGGCT
CCAGACCCGCGGCCCGGAGGCCCCGGGAATTACCAGCCGCCCGCGCTCCCATCCTT
CCCTTTTTGGGAAATAAAAAGACGAGCGCGGGTGAATGCTTCTCTAGCTTCCCTCAGCT
CATTGACTGTGATCTGTGACCAGAAAGTGTTTTGTGGTTGGTTGGAGTTAGTGAGAG
GAGGAGAAAAAATCAGAAAGCGCAACACCACAAACCGAAAAACTCCA
GCTGGTAAATTGCTCCTCCTTTATGTGCTCTCTCCTCCTCGTGTAAGAAAGACGCAAGGAG
GGAAAGCCGGGGTGGCTTTATATAGGCTGTTAAAAAAAGACTATGGGCCCAT
CTATTAGGGCCAGAGGCCCCAGGGAGCAGCCCCGGAAGGGGGGAGGCTCCCGG
GAGCTTCGCGGCAAAGAGTGTCGCGCTCCGCTCCGTGCTGCTGCTGCTGCTGCTGCTGCT
CCAGGGGGAAGGCAGCAGGAGCAGGAGCAGGAGCAGGAGCAGGAGCAGGAGCAGGAGCAGGAGCAG
AAGGACGCTTCGGGAACACGCCGAACACCACCAAGCGGGAATCCCGGCTCCCGG

>gi|224589800:c11163838-111682839 Homo sapiens chromosome 1, GRCh37.p10 Primary Assembly