Student: Yu Cheng (Jade)

ICS 675

Assignment #1

October 25, 2009

Exercise 1 Algorithm time complexity analysis

Question: What is the time complexity of the algorithm described below. Detail the answer using a diagram, or any other useful representation.

```
Generator(n, symbols, s) returns array {
# Pre-conditions:
#       n ≥ 1
#       s = size of the array symbols
#       All(1 ≤ i ≤ s, i ≤ j ≤ s, symbols[i] != symbols[j])
#
# Post-condition
#       All(i < j where j < size of outputs, outputs[i] != outputs[j])

        outputs = symbols;
        for i ← 1 ··· n − 1 {
                j = 0;
                for each elem in outputs {
                        for each symbol in symbols {
                                # concatenates elem any symbol.
                                temp[j] = concatenate(elem, symbol);
                                j++;
                        }
                }
                outputs = temp;
        }
        return outputs;
}
```

Answer: This is a brute force way to generate all possible l-mers with a length of n. The outer loop goes from 1 to $n-1$. It give $n-1$ rounds. The inner most loop goes from 1 to the size of the array $symbols$, in the context of DNA analysis, it would be 4. The middle loop execution rounds vary as the function executes. It starts at the size of $symbols$, which is 4, and increases to 16, 64, 256, and so on. So the time complexity of this function can be expressed as:

$$s \cdot s + s^2 \cdot s + s^3 \cdot s + \cdots + s^{n-1} \cdot s = s^2 + s^3 + s^4 + \cdots + s^n$$

$$= \frac{s^2 \cdot (s^{n-1} - 1)}{s - 1}.$$

The time complexity of this given function is $O\left(\frac{s^2\cdot(s^{n-1}-1)}{s-1}\right) = O(s^{n+1}) = O(s^n)$.

Exercise 2 Greedy approach to motif finding

Question: Find the two closest sequences in a t-size input. For the selected sequences find the positions s_1 and s_2 that optimize the $Score(s, DNA)$. Use a greedy approach, search of the motif in the other sequences $t-2$ sequences.

Answer: The $greedymotifsearch$ function firstly takes two rows and compute the best motifs for them in a brute force way. Then, it loops through the rest of the rows. There are $t-2$ of them. Assuming that the motifs on the previous rows are already selected, algorithm finds out the best motif on the current row and appends it to the list of the best motifs.

The starting two sequences are selected applying hamming distance in the $hammingswap$ function. The two sequences that have the least hamming distance are closer to each other than any other pairs of sequences. Computing the closest sequences using hamming distance was done in a brute force manner, which is pretty time-consuming.

```
/* ------------------------------------------------------------------------- */
/**
 * Function searches for the starting positions of the motifs that produce the
 * best score on the first two rows of the DNA matrix in a brute force manner.
 * Then updates the starting positions on the rest of the rows assuming that
 * the previous motifs have been already chosen.
 *
 * Select the two rows that have the least hamming distance.
 *
 * @param bestMotif The array of indexes to be filled in with the starting
 *        positions that give the best score according this greedy algorithm.
 *
 */
static void greedymotifsearch(int * bestMotif)
{
    int * startpos;
    int   end, i;
    end = cols - length;

    /* Select the two rows that have the least hamming distance. */
    hammingswap();

    /* Allocate space for the aray storing the temporary starting positions. */
    startpos = (int *)calloc(rows, sizeof(int));

    /* Search for the best motifs on the first two rows in a Brute force way. */
    for (startpos[0] = 0; startpos[0] < end; startpos[0]++) {
```

```
        for (startpos[1] = 0; startpos[1] < end; startpos[1]++) {
            if (score(startpos,  2) >
                score(bestMotif, 2)) {
                bestMotif[0] = startpos[0];
                bestMotif[1] = startpos[1];
            }
        }
    }

    /* Record the starting positions of the best motifs found on the first two rows. */
    startpos[0] = bestMotif[0];
    startpos[1] = bestMotif[1];

    /* Search the rest t-2 rows assuming the previous motifs are all selected. */
    for (i = 2; i < rows; i++) {
        for (startpos[i] = 0; startpos[i] < end; startpos[i]++) {
            if (score(startpos,  i + 1) >
                score(bestMotif, i + 1))
                bestMotif[i] = startpos[i];
        }

        /* Record the starting position giving the best score on the current row. */
        startpos[i] = bestMotif[i];
    }

    free(startpos);
}

/* ------------------------------------------------------------------------- */
/**
 * Function conputes the hamming distances between all pairs of rows in the DNA
 * matrix and records the pair that has the least hamming distance.  Then
 * function swaps the selected rows, minrow1 and minrow2, with the first and the
 * second row in the DNA matrix respectively.  The updated DNA matrix's first
 * two rows have the least hamming distance of all rows.
 *
 */
static void hammingswap()
{
    int i, j, k;
    int min, minrow1, minrow2;
    min = cols;

    /* Search for the two rows having the least hamming distance brute forcely. */
    for (i = 0; i < rows; i++) {
        int distance;
        distance = 0;
        for (j = i + 1; j < rows; j++) {
            for (k = 0; k < cols; k++) {
                if (dna[i * cols + k] != dna[j * cols + k])
                    distance++;
            }
            if (distance < min) {
                min = distance;
                minrow1 = i;
                minrow2 = j;
            }
```

```
        }
    }

    /* Swap the selected rows with the first two rows in the DNA matrix. */
    swaprows(minrow1, minrow2);
}

/* ------------------------------------------------------------------------ */
/**
 * Function computes the score for a paticular set of motifs.  It sums up the
 * max character counts on each column that starts from the specified positions
 * and ends at positions that are motif-length away from the starting positions.
 *
 * @param startpos The array of starting indexes in the DNA matrix.
 * @param rowcount The number of rows to scan to compute the score.
 *
 */
static int score(const int * startpos, int rowcount)
{
    int j;
    int total;

    /* Loop lenght many columns and sum up the scores. */
    total = 0;
    for (j = 0; j < length; j++) {
        int i;
        int max;
        int sum[4] = { 0, 0, 0, 0 };

        /* Only scan specified number of sequences. */
        for (i = 0; i < rowcount; i++) {
            char ch;
            ch = dna[i * cols + startpos[i] + j];
            sum[(int)ch]++;
        }

        /* Add the one with the greatest count in this column to total. */
        max = sum[0];
        for (i = 1; i < 4; i++)
            max = sum[i] > max ? sum[i] : max;
        total += max;
    }

    return total;
}
```

Question: Analyze the complexity of your new greedy motif finding algorithm 1.

Answer: In *hammingswap* function, we scan through the rows two times and the columns once in a nested loop. It would take $O(t^2 \cdot n)$ time. The *swaprows* function (helper functions are not copied here, please refer the source code submitted.) just loop once through the length of the sequences. It would take $O(n)$ time. Clearly $O(t^2 \cdot n)$ is a higher order term that defines the time complexity of this function.

$score$ function is used in $greedymotifsearch$ function. In the $score$ function, we have a nested loop that takes $O(l \cdot i)$, where l is the length of the l-mer motif, and i is the specified row count to scan. i is upper bounded by t.

In the rest of $greedymotifsearch$ function, we have two parts. The first part searches for the best motif in two rows in a brute force way. It takes $O[(n-l+1)^2 \times 2l \cdot t] = O[lt(n-l)^2] = O(n^2 \cdot l \cdot t)$. The second part loops through the rest of the sequences. It takes $O[(t-2) \times (n-l+1) \times 2l \cdot t] = O(t^2 \cdot n \cdot l)$. Apparently the first part is the higher order term that defines the time complexity. Because n should be substantially larger than l. Normally, we would have a lot more nucleotides on each of the sequence we are trying to analyze than the number of samples we are dealing with. So, after $hammingswap$ function, it takes $O(n^2 \cdot l \cdot t)$.

Sum them together, the original greedy algorithm portion and the hamming distance searching portion, this algorithm is in the order of $O(t^2 \cdot n + n^2 \cdot l \cdot t) = O(n^2 \cdot l \cdot t)$.

Question: Instead of choosing the closest 2 sequences from the set, select 2 sequences randomly. Repeat the process x number of times defining x tuples (s_1, s_2) of starting positions in sequence 1 and sequence 2. Chose the most reoccurring tuple of starting positions and find the remaining starting positions in your $t-2$ sequences.

Answer: The $greedymotifsearch$ function is the same as the previous greedy algorithm, only the selection of the first two rows to execute is different. Instead of randomly selecting, This algorithm implemented hamming distance to select the closest sequences of the rows.

```
The omitted sections are identical with the function copied above.
/* ------------------------------------------------------------------------- */
/**
 *  ...
 *
 * Randomly select two rows to be the first two executed by the greedy method.
 *
 *  ...
 *
 */
static void greedymotifsearch(int * bestMotif)
{
    ...

    /* Randomly select two rows and swap them into the front of the DNA matrix. */
    randomswap();

    ...
}

/* ------------------------------------------------------------------------- */
/**
```

```
 * Randomly choose two rows from the DNA matrix, and swap the chosen rows, row1
 * and row2, with the first and the second row in the DNA matrix respectively.
 * The updated DNA matrix's first two rows are a random selection of all rows.
 *
 */
static void randomswap()
{
    int row1, row2;

    /* Randomly select row1 and row2 from all rows. */
    row1 = rand() % rows;
    row2 = (1 + row1 + (rand() % (rows - 1))) % rows;

    /* Swap the randomly selected rows with the first two rows in the DNA matrix. */
    swaprows(row1, row2);
}
/* ------------------------------------------------------------------------- */
/**
 *  ...
 */
static int score(const int * startpos, int rowcount)
{
    ...
}
```

Question: Analyze the complexity of your new greedy motif finding algorithm 2.

Answer: The $randomswap$ function doesn't take any time in comparison. The entire time consumption comes from the main routine, which is the original $greedymotifsearch$ function. As we decided it takes $O(n^2 \cdot l \cdot t)$, this algorithm as a whole, therefore, takes $O(n^2 \cdot l \cdot t)$ time.

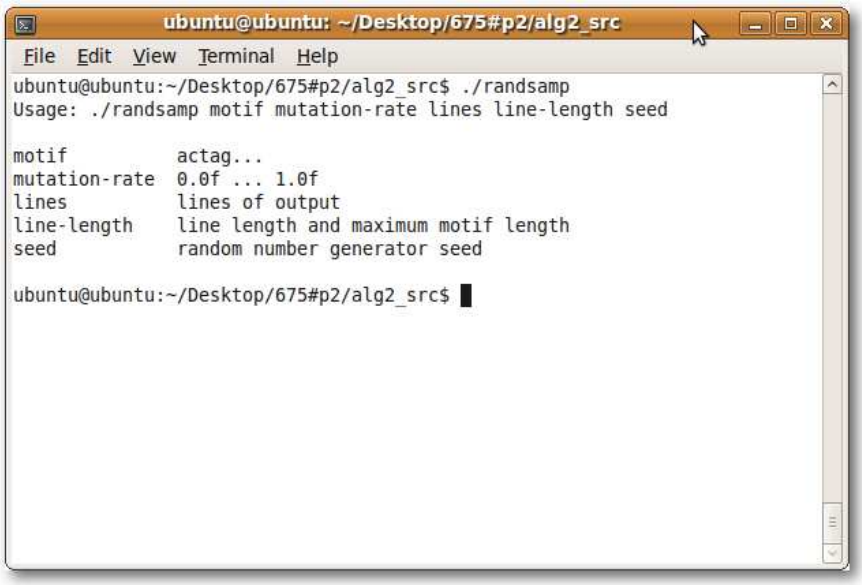
The two algorithms are in the same time complexity order. Algorithm 2 should be faster in a small scale inputs than algorithm 1 though, as it does not spend time searching for the first pair of sequences to scan.

Question: What can you say about the algorithms 1 and 2? (Compare both approaches in algorithm 1 and 2 from a time complexity and qualitative standpoints) Which algorithm do you believe will do a better job at finding the optimal motif? Do we have a guarantee for that?

Support your analysis using example of your choice. The sample sequences are generated.

a. Random DNA sample generating

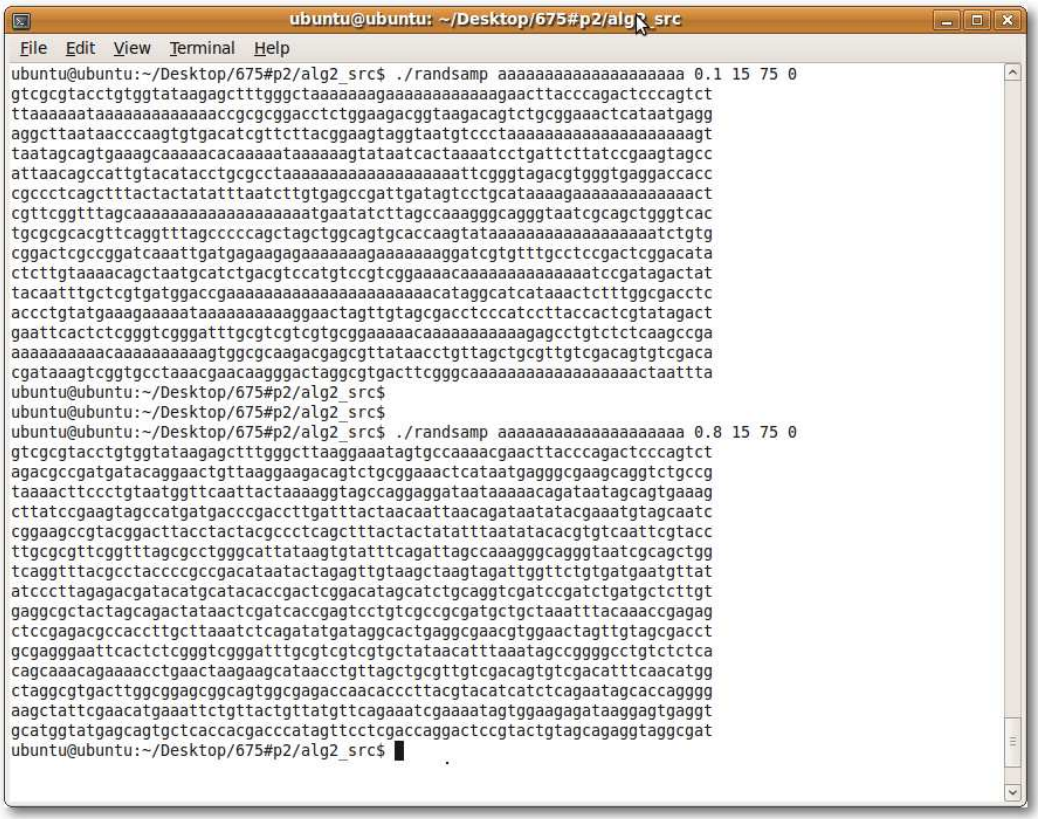
Answer: In my implementation, I tested these two algorithms with randomly generated DNA matrices. I insert a command line specified motif into all sequences in random positions. The command line also takes a rate of mutation, from 0 to 1. This allows the motifs to be different from each other by a certain ratio.



The following example demonstrates this feature. I choose a sequence of 20 a's to be the original motif, and a motif mutation ratio to be 0.1 and 0.9 for the first and second execution respectively. The DNA matrix has 15 sequences and the length of the sequences is 75 nucleotides long. In this case, I look for l-mers with size 20, of course because I'm "cheating". I already know the motifs. The random seed is given as 0. With the seed, I will be able to reproduce these sequences later if I need to.

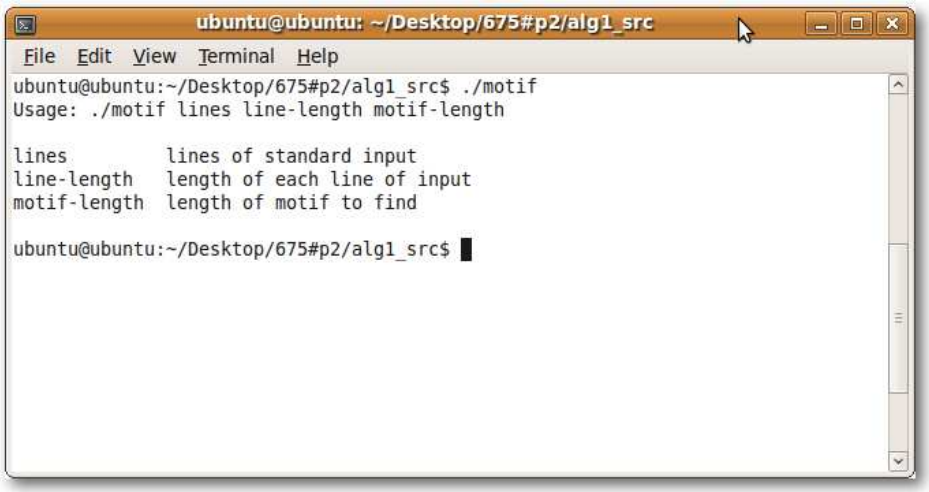
If we specify the mutation ratio to be 0.1, we can still "see" the sequences of a's in there. While if I specify the mutation ratio to be 0.9, the sequences of a's are basically "disappeared". If the mutation ratio is as high as 0.9, the specified motifs don't ready exist anymore. They might as well be random nucleotide sequences. The analysis output in this case would have too low of a score and not trustworthy anymore.

In the following screenshot, each row represents a DNA sequences and the entire standard output represent the DNA matrix generated with the specified parameters. This output format would be later used as the input for the greedy motif searching algorithms.

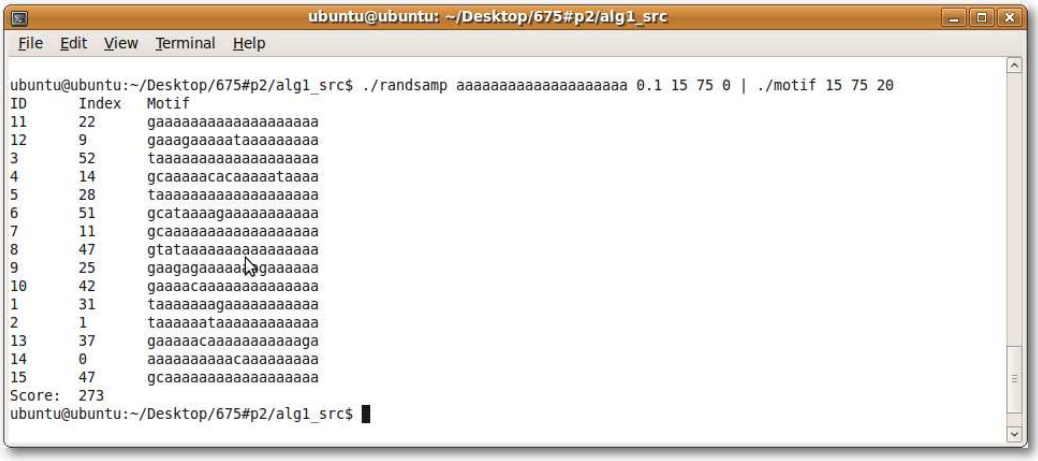


b. Greedy Motif Searching starting with the closest sequences

Answer: Then I implemented the two greedy motif finding algorithms. In the first algorithm, hamming distance was used to select the closest sequences of the DNA matrix. The closest two sequences are used as the initial sequences to compute the initial best motif.



In the following example, the randomly generated DNA matrix was piped into the motif finding function as the input. We also specified the size of the DNA matrix and the length of the motif we are looking for. Here the length of the l-mers are the same as what I've inserted into my generated DNA sequences. Again, I'm "cheating". I already know from the DNA generating function parameters that what my motifs should look like.



```
ubuntu@ubuntu: ~/Desktop/675#p2/alg1_src
File  Edit  View  Terminal  Help

ubuntu@ubuntu:~/Desktop/675#p2/alg1_src$ ./randsamp aaaaaaaaaaaaaaaaaaaa 0.1 15 75 0 | ./motif 15 75 20
ID      Index   Motif
11      22      gaaaaaaaaaaaaaaaaaaa
12      9       gaaagaaaaataaaaaaaaa
3       52      taaaaaaaaaaaaaaaaaaa
4       14      gcaaaaacacaaaaataaaa
5       28      taaaaaaaaaaaaaaaaaaa
6       51      gcataaaagaaaaaaaaaaa
7       11      gcaaaaaaaaaaaaaaaaaa
8       47      gtataaaaaaaaaaaaaaaa
9       25      gaagagaaaaaagaaaaaaa
10      42      gaaaacaaaaaaaaaaaaaa
1       31      taaaaaaagaaaaaaaaaaa
2       1       taaaaaataaaaaaaaaaaa
13      37      gaaaaacaaaaaaaaaaaga
14      0       aaaaaaaaaacaaaaaaaaa
15      47      gcaaaaaaaaaaaaaaaaaa
Score:  273
ubuntu@ubuntu:~/Desktop/675#p2/alg1_src$
```

From the program output, we saw that DNA sequences were rearranged since the 11th and 12th sequences were decided by the hamming distance algorithm to be the closest sequences of the 15 input sequences. Then the best motif array was generated indicating the starting indexes of the motifs found on each sequence. Also, by knowing the starting indexes and the input DNA matrix, we were able to retrieve the exact l-mers found on each sequence. They are shown as the third column of the output.

Apparently, by taking a low mutation ratio, 0.1, we still have most of the a's in our l-mers. The maximum score for the l-mers, with lengths of 20 and sample size 15, is $15 \times 20 = 300$. We had 273, which should be pretty good.

In the following example we have a comparison of low and high mutation ratios. We can clearly see the difference. First, we can really see our sequences of a's anymore as the l-mers found on each sequence. Then, notice that we had a bad score as the sequences found don't really close to each other. Because the way our scoring schema works, for a completely random comparison, we would still get $1/4$ amount of scores. In this example is would be 75. We got 150 because the greedy algorithm is helping us to make relatively good choices rather than completely random outputs, in which case, we would expect around 75.

```
ubuntu@ubuntu: ~/Desktop/675#p2/alg2_src
File  Edit  View  Terminal  Help

ubuntu@ubuntu:~/Desktop/675#p2/alg2_src$ ./randsamp aaaaaaaaaaaaaaaaaaaa 0.1 15 75 0 | ./motif 15 75 20 10
ID      Index   Motif
11      22      gaaaaaaaaaaaaaaaaaaa
12      9       gaaagaaaaataaaaaaaaa
3       52      taaaaaaaaaaaaaaaaaaa
4       14      gcaaaaacacaaaaataaaa
5       28      taaaaaaaaaaaaaaaaaaa
6       51      gcataaaagaaaaaaaaaaa
7       11      gcaaaaaaaaaaaaaaaaaa
8       47      gtataaaaaaaaaaaaaaaa
9       25      gaagagaaaaaaagaaaaaa
10      42      gaaaacaaaaaaaaaaaaaa
1       31      taaaaaaagaaaaaaaaaaa
2       1       taaaaaataaaaaaaaaaaa
13      37      gaaaaacaaaaaaaaaaaga
14      0       aaaaaaaaaacaaaaaaaaa
15      47      gcaaaaaaaaaaaaaaaaaa
Score:  273
ubuntu@ubuntu:~/Desktop/675#p2/alg2_src$ ./randsamp aaaaaaaaaaaaaaaaaaaa 0.9 15 75 0 | ./motif 15 75 20 10
ID      Index   Motif
11      6       atttgcgtcgtcgtgcggtt
12      16      agctgcgttgtcgacagtgt
3       2       atgtcccttcaaattcaatt
4       3       agtaccgatgatgacccgta
5       54      atctcctaggtctgcacgat
6       16      atttccaatggaatatctta
7       2       gtgacggtcgacaccaagtt
8       37      ttctgcattatggcttcgta
9       6       agctcaatatcagtcctgtc
10      7       gcctgtatgatcatctcagt
1       7       acgtctggtataagagcttt
2       6       aggtctccaggcgccggtaa
13      35      atagcaccaggggtctcttg
14      51      acgaggggtggcgcaccggc
15      9       cagtccgtactgtagcagag
Score:  158
ubuntu@ubuntu:~/Desktop/675#p2/alg2_src$
```

c. Greedy Motif Searching starting with a randomly selected two sequences

Answer: The second greedy algorithm implemented used a randomly selected pair of sequences to be the first two rows algorithm execution. Since randomization is used, we need to take one extra parameter to seed the random number generator. Also we will be able to reproduce the output.

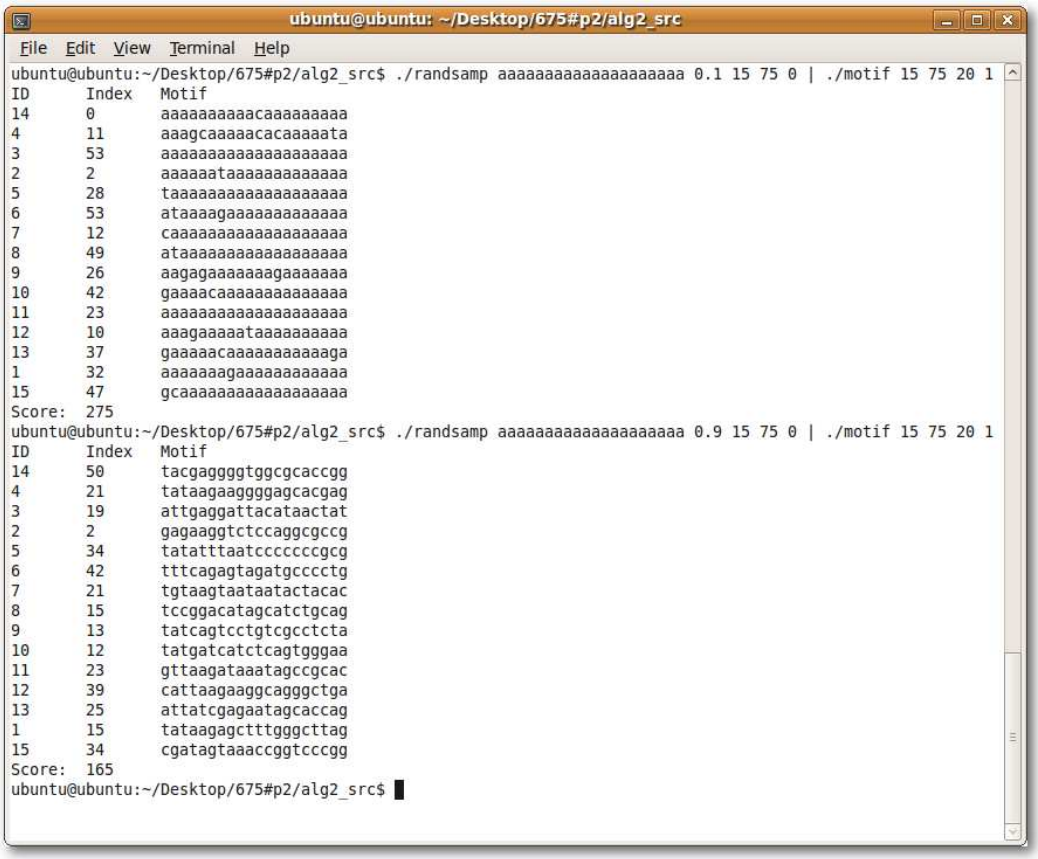
```
ubuntu@ubuntu: ~/Desktop/675#p2/alg2_src
File  Edit  View  Terminal  Help
ubuntu@ubuntu:~/Desktop/675#p2/alg2_src$ ./motif
Usage: ./motif lines line-length motif-length

lines          lines of standard input
line-length    length of each line of input
motif-length   length of motif to find
seed           random number generator seed

ubuntu@ubuntu:~/Desktop/675#p2/alg2_src$
```

The same DNA matrix was used as the input data in the following example. We will be able to compare the outputs better.

```
ubuntu@ubuntu: ~/Desktop/675#p2/alg2_src
File  Edit  View  Terminal  Help
ubuntu@ubuntu:~/Desktop/675#p2/alg2_src$ ./randsamp aaaaaaaaaaaaaaaaaaaa 0.1 15 75 0 | ./motif 15 75 20 1
ID      Index   Motif
14      0       aaaaaaaaaacaaaaaaaaa
4       11      aaagcaaaaacacaaaaata
3       53      aaaaaaaaaaaaaaaaaaaa
2       2       aaaaaataaaaaaaaaaaa
5       28      taaaaaaaaaaaaaaaaaaa
6       53      ataaaagaaaaaaaaaaaaa
7       12      caaaaaaaaaaaaaaaaaaa
8       49      ataaaaaaaaaaaaaaaaaa
9       26      aagagaaaaaaagaaaaaaa
10      42      gaaaacaaaaaaaaaaaaaa
11      23      aaaaaaaaaaaaaaaaaaaa
12      10      aaagaaaaataaaaaaaaaa
13      37      gaaaaacaaaaaaaaaaaga
1       32      aaaaaaagaaaaaaaaaaaa
15      47      gcaaaaaaaaaaaaaaaaaa
Score:  275
ubuntu@ubuntu:~/Desktop/675#p2/alg2_src$ ./randsamp aaaaaaaaaaaaaaaaaaaa 0.1 15 75 0 | ./motif 15 75 20 2
ID      Index   Motif
1       33      aaaaaagaaaaaaaaaaaag
13      38      aaaaacaaaaaaaaaaagag
3       54      aaaaaaaaaaaaaaaaaaag
4       17      aaaacacaaaaataaaaag
5       29      aaaaaaaaaaaaaaaaaaat
6       53      ataaaagaaaaaaaaaaaaa
7       13      aaaaaaaaaaaaaaaaaaat
8       49      ataaaaaaaaaaaaaaaaaa
9       27      agagaaaaaaagaaaaaaag
10      43      aaaacaaaaaaaaaaaaaat
11      23      aaaaaaaaaaaaaaaaaaaa
12      11      aagaaaaataaaaaaaaag
2       2       aaaaaataaaaaaaaaaaa
14      2       aaaaaaaacaaaaaaaaag
15      48      caaaaaaaaaaaaaaaaaac
Score:  274
ubuntu@ubuntu:~/Desktop/675#p2/alg2_src$ ./randsamp aaaaaaaaaaaaaaaaaaaa 0.1 15 75 0 | ./motif 15 75 20 3
ID      Index   Motif
7       10      agcaaaaaaaaaaaaaaaaa
13      35      cggaaaaacaaaaaaaaaaa
3       49      ccctaaaaaaaaaaaaaaaa
4       13      agcaaaaacacaaaaataaa
5       24      cgcctaaaaaaaaaaaaaaa
6       50      tgcataaaagaaaaaaaaa
1       28      ggctaaaaaaagaaaaaaaa
8       46      agtataaaaaaaaaaaaaaa
9       24      agaagagaaaaaaagaaaaa
10      40      cggaaaacaaaaaaaaaaaa
11      21      cgaaaaaaaaaaaaaaaaaa
12      8       tgaaagaaaaataaaaaaaa
2       0       ttaaaaaataaaaaaaaaaa
14      0       aaaaaaaaaacaaaaaaaaa
15      46      ggcaaaaaaaaaaaaaaaaa
Score:  259
```

As the random seed changes the first pair of sequences selected changes. The first execution took sequences 14 and 4 as the brute force portion of the greedy algorithm. The second execution took sequences 1 and 13. The third execution took sequences 7 and 13. In the example above the value of $x = 3$, and the best score provided was 275, which is about the same as algorithm 1.

Also I tested the effect of different mutations ratios. The result is, of course, the same as algorithm 1. A greater mutation ratio gives a lower score and the analysis is not trustworthy.



```
ubuntu@ubuntu: ~/Desktop/675#p2/alg2_src
File  Edit  View  Terminal  Help
ubuntu@ubuntu:~/Desktop/675#p2/alg2_src$ ./randsamp aaaaaaaaaaaaaaaaaaaa 0.1 15 75 0 | ./motif 15 75 20 1
ID      Index   Motif
14      0       aaaaaaaaaacaaaaaaaaa
4       11      aaagcaaaaacacaaaaata
3       53      aaaaaaaaaaaaaaaaaaaa
2       2       aaaaaataaaaaaaaaaaaa
5       28      taaaaaaaaaaaaaaaaaaa
6       53      ataaaagaaaaaaaaaaaaa
7       12      caaaaaaaaaaaaaaaaaaa
8       49      ataaaaaaaaaaaaaaaaaa
9       26      aagagaaaaaaagaaaaaaa
10      42      gaaaacaaaaaaaaaaaaaa
11      23      aaaaaaaaaaaaaaaaaaaa
12      10      aaagaaaaataaaaaaaaaa
13      37      gaaaaacaaaaaaaaaaaga
1       32      aaaaaaagaaaaaaaaaaaa
15      47      gcaaaaaaaaaaaaaaaaaa
Score:  275
ubuntu@ubuntu:~/Desktop/675#p2/alg2_src$ ./randsamp aaaaaaaaaaaaaaaaaaaa 0.9 15 75 0 | ./motif 15 75 20 1
ID      Index   Motif
14      50      tacgaggggtggcgcaccgg
4       21      tataagaaggggagcacgag
3       19      attgaggattacataactat
2       2       gagaaggtctccaggcgccg
5       34      tatatttaatcccccccgcg
6       42      tttcagagtagatgcccctg
7       21      tgtaagtaataatactacac
8       15      tccggacatagcatctgcag
9       13      tatcagtcctgtcgcctcta
10      12      tatgatcatctcagtgggaa
11      23      gttaagataaatagccgcac
12      39      cattaagaaggcagggctga
13      25      attatcgagaatagcaccag
1       15      tataagagctttgggcttag
15      34      cgatagtaaaccggtcccgg
Score:  165
ubuntu@ubuntu:~/Desktop/675#p2/alg2_src$
```

The randomly selected two sequences are the same because the same seed is passed in both of the two executions. The score is much lower providing a great mutation ratio.

d. Which algorithm do you believe will do a better job at finding the optimal motif? Do we have a guarantee for that?

Answer: For randomly generated DNA sequence samples, algorithm 1 doesn't have any advantage. Spending time looking for the closest pair did do us any good. Of course, in comparison, this time spent looking for the closest pair of sequences is inferior than the time consumption the main routine – greedy searching.

For random sample algorithm 1 is not any better than algorithm 2. It makes a lot of sense, because the samples are completely *random*. How can we trust the selected pair to be any substantially closer? In fact, they are not, they are just happen to be slightly closer. The way they are close with each other doesn't have anything to do with the motifs! Because, the motifs are randomly mutated as well! This is, however, no t the case in real organisms. If two organisms are closely related with each other in the context of evolution, we should expect a overall closer target sequences and a overall closer motifs on these sequences.

So, in the real world research, under the circumstance that the researchers already know that they are looking at some sequences that are related with each other closely, it might be beneficial to spend time looking for the closest pair and make the best guess to start with.

On the other hand, although the greedy algorithm implementing searching for the closest sequences might provide a good starting point, we still need to evaluate whether it worth the time. As we saw in algorithm 2, by running it several times, we would be able to pick a pretty good overall score. We implemented hamming distance as the quantifier of how close the sequences are. It doesn't cost much time, so we'll say yes, to this little extra effort. But it doesn't do that good of a job comparing the sequences either! It would greatly depend on the implementation to say which is better. I imagine the second approach, algorithm 2, would be a better choice in most cases, especially if we aren't so sure about the phylogenetic relations of the sequences we are looking at.

There is no guarantee for either of these two algorithms to be better than the other. Just like there is no guarantee that either of these two algorithms provides the best option. If we are going for a guarantee, we would take the brute force approach for all input sequences. The reason implementing greedy algorithms is that we would rather trade optimization with time consumption.