Programming Project
CSC373 Winter 2010
Due Date: March 23rd, 11:59pm

Administrative details:
This project will be submitted electronically. A full list of receivables is lower. We encourage you to work in groups, with a maximum of 3 students per group. The main communication medium for this project will be the Google group; this is where we'll post clarifications, explanations, and other project-related details.

Motivation & Background:
Sequence alignment, that is the comparison of biological sequences, is one of the most successful applications of computer science to biology. DNA is a double stranded molecule, where each strand is a chain of nucleotides. There are four types of nucleotides (Adenine, Cytosine, Guanine, Thymine), which are commonly represented by the four letters A, T, C, and G. Because of the large size of biological data (the human genome has about 3 billion nucleotides, other mammalian genomes are about the same size) efficient algorithms are necessary when one is analyzing genomic sequences. In this project you will get to implement one such algorithm.

Edit Distance algorithm
One of the oldest algorithms for alignment of sequences is the Needleman-Wunsch algorithm, which was published in 1970. This algorithm is closely related to the Edit distance algorithm that you saw in lecture (and is in the book). This algorithm calculates the *edit distance* between the two strings. Given a match score T, a mismatch score R, and a gap penalty G per each gapped letter, the edit distance between two strings is defined to be the maximum possible weighted sum of the number of match, mismatch, & gap scores. The Edit Distance algorithm calculates the number of operation required to transform string$A:A_1 \ldots A_k$ int $B:B_1 \ldots B_n$, with the valid operation being insertions, deletions, and substitutions of a single letter. The algorithm works by creating an k×n matrix M, where the element $M_{i,j}$ will store the edit distance between strings $A_1 \ldots A_i$ and $B_1 \ldots B_j$. We can update the matrix by setting

$$M_{i,j} = MAX(M_{i-1,j} - 1, M_{i,j-1} - 1, M_{i-1,\, j-1} - S(A_i, B_j))$$

Where $S(A_i, B_j)$ returns 0 if the two letters are the same, and 1 otherwise. To reconstruct the alignment we can add back-pointers to keep track which move we made, and walk the backpointers from $M_{k,n}$ back to $M^{0,0}$ to get the actual alignment.

The problem with this approach is that it requires O(k*n) time and memory, and although it is possible to reduce the memory usage, reducing the time requirement significantly is impossible without sacrificing optimality.

Heuristic DNA alignment

In order to enable rapid alignment of longer DNA sequence, scientists commonly use the *anchoring* heuristic, where a speed up is achieved by first matching K-mers (exact matching strings of length ≥K) rather than individual letters. After finding matching K-mers, these are extended to the left and right until the first mismatch is detected, and the score of this extended *maximal match* is typically just its length. Some chain of these maximal matches are chosen as *anchor points*. This chain of anchor points must be *consistent*: for two adjacent maximal matches, the start of the second match must be (strictly) after the end of the first in both sequences. The score of a chain is the sum of the scores of all of the maximal matches that are part of the chain. Once this chain is found, we only need to do the alignment *between* the anchors, making the alignment procedure significantly faster.

Your task:

In this project you will implement a fast DNA aligner, which we will call *fastmatch*. *Fastmatch* can be used for aligning two long DNA sequences. Your data will come in the form of FASTA files, the most common format for biological sequences. A FASTA format file starts with a *header* line which has ">" as the first symbol. Every line after the header line may have zero or more letters from the alphabet {A,T,C,G}, making up the sequence to be aligned. Your program will take as input the names of two fasta files and one command-line options: the K-mer size (zero is a special case, indicating no anchoring).

Your output will be the same two sequences, with dashes ("-") inserted wherever you believe there have been insertions/deletions, as well as one additional line, starting with the "=" symbol and followed by two numbers: the total number of edits you used to transform one string into the other, and the total weight of all anchors. For example, given the strings

sequence1.fa:

```
>s1
ACTGACT
```

sequence2.fa:

```
>s2
ACGCTG
```

Here are two possible runs: (% is the UNIX prompt, and bold characters are so that you can see anchors, you don't have to print bold)

```
%fastmatch sequence1.fa sequence2.fa 2
>s1
A--CTGACT
>s2
ACGCTG---
=5 3
```

Note that there may be multiple valid alignments with the same edit distance and anchoring score. All are acceptable.

Grading criteria:
Your program will be grade on correctness and speed. In particular, to get full credit, your program must be able to handle query files of length up to several million letters long in a reasonable amount of time (reasonable depends on the k-mer size, but it should be minutes or seconds rather than hours given large enough K). The K-mer size may vary between zero (in which case you are just asked for the edit distance between the two strings) and 20. You will get some partial marks for printing the correct edit distance (with incorrect alignments), being able to compute correct alignments without anchors, and printing the correct anchoring score without alignments or edit distance.. You may assume that the computer on which we will be testing your submission has at least one gigabyte of memory. We will grade a large portion of your project automatically, so it is important that you follow the directions about output format precisely.

You may also receive bonus points for the project by implementing an algorithm that can efficiently align *extremely* similar long sequences and by implementing an alignment algorithm that uses linear (rather than quadratic) space. The first case only requires you to notice one special feature of the extended k-mer matches. The second algorithm we discussed in lecture.

Receivables:

- Your code, in your favorite programming language.
- A Makefile, that will compile your code, and create the executable *fastmatch*. This executable should be runnable from the command line. If you are using java, python, or another interpreted language please include a small script that will call your interpreted program with the right arguments.
- A README file explaining the algorithms that you are using, the time and memory complexity of these algorithms and general details about how you have implemented them. If you have done either of the bonus assignments, please indicate this at the beginning of your readme.

While you are welcome to use any standard libraries available in your language, you may not download any code from the web; all code you submit must be your own work. Submission information for the project and sample data will be available in the near future.

Plan of Attack:
This project, as almost all projects that you will ever work on, nicely divides into several distinct sections, which can be given to different group members. The division below is just a suggestion. You are free to go about this any way you wish.

- **Finding maximal matches (A)**
  Given two strings, you will want to index all of the K-long strings (and their locations) in one of them, and then use the K-long strings in the other to query this index. Whenever you find a match, you will want to extend it in both directions to the first mismatch and return a tuple indicating positions of this maximal match in both sequences. Note that implementing this algorithm trivially will lead to quadratic running time for extremely similar strings. Getting around this will yield bonus points.

- **Sub-selecting a set of Anchors (B)**
  Given a list of maximal matches, you will want to select a chain of them that can be used as anchors. You are asked to implement a greedy algorithm, that sorts all of the maximal matches by score, and accepts anchors in order of score if they form a consistent chain (you will need an efficient data structure to efficiently identify which anchors are consistent). Note that the Greedy algorithm should run in time $O(n \log n)$, where n is the number of maximal matches, rather than $O(n^2)$.

- **Edit Distance Aligner (C)**
  You will need to implement the edit distance algorithm to not only compute the edit distance, but also to reconstruct the full set of operations required to transform the strings (and print the alignments). This algorithm should take as input not only strings, but also ranges within the two strings in which to do the alignment.

- **Putting it all together**
  Once all of the pieces are working, you need to put all of them together into a single package. The way the project is organized, we will be able to test sections B and C independently by analyzing the line that starts with the "=" sign – so you will get partial credit for getting the subsections to work – but in order to get full credit you need to make the whole thing work.