CSC2417
Assignment 1

Dr. Jared Simpson

Instructions

This assignment is due **November 2nd**. To turn in your assignment, make a `tar.gz` file containing your source code and a README file and email it to `jared.simpson+csc2417-a1@gmail.com`. The README file should contain:

- Your name, student number and email address
- The instructions to compile your code (if necessary)
- Instructions on how to run your code to generate the answers to the programming problems
- Answers to the written questions

Here is an example (partial) README:

```
#Name:  Jared Simpson
#Student number:  0000000000
#Email:  jsimpson@cs.toronto.edu

# Compilation
To compile the programs I wrote to solve the programming problems simply run
‘make’.

# Solve problem 1.1
./reverse-complement.py ACTGATGT

# Solve problem 1.2
./generate-sorted-strings.py 5

# Answer to question 3.1
To calculate the shortest unique substring of a string X we can...
```

If you prefer to write in LaTeX rather than a plain text file, you can turn in a PDF containing the same information instead. For the programming problems, you can use Python, C/C++, Java or Perl, with Python preferred. If you want to use a different language, please contact me first to ask.

The assignments are intended to be solved individually—you can discuss the problems with your classmates, but please do not give the answers away. For the questions requiring a written answer (like 2.3 and 3.1 here), please provide a complete description of both how the algorithm works and why it works (formal proofs are not required however). If the instructions or problem descriptions are unclear, please ask on the Google Group.

1 DNA Sequences and Strings

1.1 Reverse complementing a DNA sequence

DNA consists of two strands of nucleotides, bound together in a double helix. This configuration allows DNA to be (nearly) perfectly copied by using either strand as a template for the newly

synthesized strand. Typically, when we work with DNA sequence data, we represent the molecule as a simple string representing one of the two strands: `ACTGATGT`.

Often we'll want to calculate the sequence of the *complementary* strand, in its 5' → 3' orientation. We call this calculating the *reverse complement* of the sequence. Write a function to calculate the reverse complement of a given DNA string. Test it by calculating the reverse complement of `ACTGATGT`.

1.2 Generating strings from the DNA alphabet

In class, we defined the alphabet used to represent DNA sequence data as $\Sigma = \{\texttt{A,C,G,T}\}$. Write a program that will generate all strings of a given length, k, over the DNA alphabet and write them out in lexicographic order with one k-mer per output line. Test your program for $k = 5$.

1.3 Hamming neighbourhood

Write a program that will take the results from the previous problem and find the subset of 5-mers that are within Hamming distance of 1 of the string `ACGTA`.

1.4 *k*-mer counting

Genomes contain *repeats*, substrings that occur in multiple locations. Repeats can accumulate over time as a consequence of replication errors, the insertion of foreign DNA into the genome or the accumulation of transposons that copy themselves around the genome (like the Alu transposons). A common method of repeat detection is looking for over-represented k-mers (substrings of length k that occur more times than you would expect by chance).

Download and decompress human chromosome 20 from http://hgdownload.cse.ucsc.edu/goldenPath/hg38/chromosomes/chr20.fa.gz. Write a program to find the top 100 most frequent 8-mers. The output of your program should have one <k-mer,count> pair per line (e.g. `ACGTACGT,984`) where the most frequent 8-mer is output first and the remainder are in decreasing order of frequency. Note that human chromosome 20 is in the FASTA format. You'll need to write a parser for FASTA or use a package like BioPython's SeqIO module to read in the input.

In the README file answer the following questions: a) how much memory did your solution require? b) will the memory usage of your program increase if we try to count longer k-mers (say for $k = 50$)? if so, can you describe an alternative solution (you don't have to implement it) that would be more memory efficient?

2 Dynamic Programming

2.1 Backtracing through a dynamic programming matrix

In class, we discussed how to use dynamic programming to calculate the edit distance between a pair of strings X and Y. The lecture slides describe how we can *backtrace* through the dynamic programming matrix to find the optimal alignment (or edit transcript) between the two strings. Use dynamic programming and the backtrace algorithm to find the alignment that minimizes the edit distance between:

```
X=AGTAGGCATAGAATGATAGTAGACCAGTAGACAGTACTTAGA
Y=AGTAGCATAGAATGATATGTAGACCAGTCGACAGTACTTAGA
```

Write the alignment in "bars and dashes" format we saw in class like this:

```
ACCG-AC
||  | ||
ACGGTAC
```

2.2 Counting the number of optimal alignments

The edit distance algorithm with backtracing is guaranteed to find the alignment that minimizes the edit distance between the pair of input strings. There is no guarantee, however, that this is the only alignment that minimizes the edit distance. Extend your code from the previous exercise to output *all* alignments with the minimum edit distance for the strings:

```
X=AGTAGGCATAGAATGATAGTAAAGACCAGTAGACAGTACTTAGA
Y=AGTAGGCATTAGAATGATAGTAGACCAGTAGACAGTACTTAGA
```

In the README file, describe why there is more than one alignment with the minimum edit distance in this case (what is the feature of the strings that cause multiple equivalent alignments?).

2.3 Constrained edit distance

Often, when we want to compare two DNA sequences, we expect them to be similar—that is, we expect their optimal alignment to have few mismatches and gaps. For example, the error rate of Illumina reads is roughly 0.5%, so when comparing an Illumina read to a reference genome, we would expect most bases to match. In the README file, describe a refinement to the edit distance algorithm that is faster than $O(mn)$ when we constrain the optimal alignment to have at most d edits (where d is some small constant much less than the length of the strings). You do not need to implement your solution, just describe it in the README file.

3 Suffix Trees

3.1 Shortest unique substring

In class, we discussed how the MUMmer program uses a generalized suffix tree (GST) to calculate *maximal unique matches* between two genomes X and Y. This algorithm traversed the GST of `X#Y$` to find internal nodes that had a single suffix of X and a single suffix of Y in its subtree. The suffix tree can be used to efficiently solve other interesting string problems, one of which we'll explore here.

In the README file, describe an algorithm that will find the *shortest unique substring* of a string X (the shortest substring that occurs exactly once in X). For this problem, you do not have to implement your algorithm in code, just describe in the README how it would work.