CSC2417
Assignment 2

Jared Simpson
Joseph C. Somody

Instructions

This assignment is due on **Monday 21 November 2016**, by **23:59**. To turn in your assignment, make a `tar.gz` file containing your source code and a README file, and e-mail it to `jared.simpson+csc2417-a2@gmail.com`. The README file should contain:

- your name, your student number, and your e-mail address;
- instructions on how to compile your code, if necessary;
- instructions on how to run your code to generate the answers to the programming problems; and
- answers to the written questions.

If you prefer to write in LaTeX rather than in plain text, you can instead turn in a PDF containing the same information. For the programming problems, you can use Python (preferred), C/C++, Java, or Perl. 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 questions requiring a written answer, please provide a complete description of both how the algorithm works and why it works; however, formal proofs are not required. If the instructions or problem descriptions are unclear, please ask via Google Groups.

Late submissions will be penalised by a deduction of 5% of the maximum grade per day (or partial day).

1 The Suffix Array

In class, we discussed how the suffix array data structure uses less memory than the suffix tree while retaining the ability to make `count` and `locate` queries (to count the number of times a pattern occurs in a text, and return the locations of those occurrences, respectively). The tradeoff, however, is that many queries that are easy with the suffix tree are more difficult with the suffix array. In this problem, we will explore how the use of the *longest common prefix* array can accelerate some of these queries. Let $\texttt{LCP}[i]$ be the length of the longest common prefix between the suffixes $\texttt{T}[\texttt{SA}[i-1]]$ and $\texttt{T}[\texttt{SA}[i]]$ for $1 < i \leq n$. Using the suffix array and LCP array of a text T, describe algorithms to solve the following problems (you do not have to code these, only describe the algorithm).

(a) Find the longest repeated substring of T.

(b) A substring S of T is a *minimal unique substring* if S appears exactly once in T and all proper prefixes of S occur more than once in T. Describe an algorithm using the suffix array and LCP array of T to find *all* minimal unique substrings of T in $O(n)$ time where $n = |T|$.

2 The FM-Index

The FM-index is an improvement over the suffix array in both space consumption and query time for `count` and `locate` queries. In this exercise, we'll use the FM-index on real data. Download the files from https://github.com/jts/csc2417/tree/gh-pages/code/a2 which provide a basic FM-index implementation and the BWT of a 1 Mbp region of Chromosome 20.

(a) Run the provided FM-index implementation to count the number of times a string occurs in the indexed region of chromosome 20:
`python ./fmindex.py -bwt small.bwt -count GCGCGGTGGCTCACGCCTGTAATCCCAG`

(b) The naïve implementation used in Part (a) has excessive memory usage as we store the Occurrence array used to calculate the character ranks for all $0 \le i < n$, requiring $O(n|\Sigma|\texttt{log}(n))$ space. In Slides 33–35 of the FM-index lecture, we described how to improve this using *checkpoints*, which only store the Occurrence array for i that is divisible by 128, and computing the others using the checkpoints and BWT string. Modify `fmindex.py` to use less memory by implementing this strategy. Verify you get the same count for the query string in (a).

3 Count–Min Sketch

The Count–Min Sketch (CMS) data structure is a probabilistic data structure similar to the Bloom filter (BF). The BF records whether some element is present in a set; the CMS records the (approximate) number of times the element appears in the set. In this problem, we'll explore the use of the CMS for error correction of sequencing data. Download the file `q3-reads.fa`, which is the input data for this problem. This file contains simulated sequencing data, with sequencing errors.

(a) Modify your exact k-mer counting code from Assignment 1 to calculate the k-mer frequency histogram of the input reads, where the i-th bin of the histogram is how many k-mers occur i times in the dataset. Use this histogram to identify how many 21-mers occur once. As discussed in the lectures, these k-mers likely contain a sequencing error.

(b) Now, rather than using the hash table from (a), which uses a lot of memory, use the CMS data structure to estimate the number of 21-mers that occur once. You can start from this open source implementation of the CMS: https://github.com/rafacarrascosa/countminsketch/blob/master/countminsketch.py rather than starting from scratch. Test your program for various array sizes (m parameter in the implementation). Try the values $m \in \{10000, 50000, 100000, 500000, 1000000\}$ for $d = 5$ (5 hash functions).

(c) The BF has the property that it will return false positives (but never false negatives). The false-positive rate can be reduced by allocating more memory to the BF. Describe the behaviour of the CMS data structure when more or less memory is given. What effect would this have on an error-correction algorithm that was implemented using the CMS to store k-mer counts?

4 Metagenome Assembly

The assembly algorithms discussed in class are designed for single genomes that have (approximately) uniform coverage across the genome. An interesting related problem is assembling a *metagenome*, which is a collection of genomes from an environmental sample. For example, if you extract DNA from a sample of soil or seawater, rather than sequencing a single genome, you will sequence all of the organisms that are present in that environment. This complicates the genome assembly problem as we can no longer assume that the genome is covered uniformly: the number of sequencing reads drawn from one species in a metagenome will depend on its abundance in the sample. For example, consider an environment that contains three bacterial species (A, B, and C), where 90% of the bacterial cells in the sample are species A, 9% are B, and 1% are C. When sequencing a sample of this environment, we would expect ≈ 90% of the sequencing reads to be drawn from bacteria A, and so on.

Describe the challenges this sequencing problem will present to de Bruijn graph (DBG) assembly.