FIT2004 S1/2017: Assignment 3 (Weight: 6 Marks)

THIS PRAC IS ASSESSED! Interview during Week 8 Lab

Submission deadline: Midnight 11:59pm Sun 23 April 2017

CLASS: This programming exercise has to be prepared in advance, and submitted via unit's Moodle page by 11:59pm Sunday 23 April 2017. Implement this exercise strictly using **Python programming language**. During your **assigned** Week 8 lab, your demonstrator will interview you based on your **submitted** version by downloading it from Moodle. This practical work is marked on the performance of your program **and also on your understanding of the program**. A *perfect* program with zero understanding implies you will get **zero** marks! "Forgetting" is not an acceptable explanation for lack of understanding. You must write all the code yourself, and may not use any external library routines that will invalidate the assessment of your learning. The usual input/output and other basic routines are exempted.
DEMONSTRATORS are not obliged to mark programs that do not compile or that crash. Time allowing, they will try to help in tracking down errors, but they are not required to mark programs in such a state, particularly those that do not compile. Therefore keep backup copies of working partial solutions.
OBJECTIVE: This exercise will test your understanding of search data structures for strings, and efficient pattern matching on a large text. You will be dealing with a very useful real-world problem. This problem instance has several learning elements that will enhance your ability to understand and implement a (relatively) nuanced algorithm. Repeated practice of thinking and writing programs will help you evolve into a proficient programmer.

Background

The background for this assignment is directly based on the topics covered in the week 7 lecture, including generating suffix arrays using Manber-Myers' prefix-doubling method as well as the 'magical' *Burrows-Wheeler Transform* (BWT) of some given reference string/text. Use the lecture notes and recording to understand the algorithms supporting this assignment.

Supporting material

To complete this assignment use the supporting material uploaded on FIT2004's Moodle page under `MyAssessments` section.

Task 1 of 3

Your Task 1 is to write a python program that accepts a text file containing a string of ASCII characters and generates the Burrows-Wheeler Transform (BWT) of that string. Recall from

© Arun Konagurthu, Monash University

the week 7 lecture, the BWT of a given string is derived by first computing its **suffix array**. In this task, you have to implement the Manber-Myers (prefix-doubling) algorithm to first compute the suffix array of the input string, and use that suffix array to compute its BWT.

Strictly follow these specifications when implementing Task 1

- Your program for this task **should be called** `genBWT.py`.
- Your program **must** accept an input (ASCII text) file as an argument. (That is, do NOT hard-code the file name or the string literal within your program; it should accept any input text file and process it.)
- Your input (text) file should should contain only a **one-line string** terminating in a '$' character, which your program will process. (See sample input file for Task 1 in the supporting material.)
- Your program **has to implement** the Manber-Myers' (prefix doubling) algorithm to compute the suffix array of the input string, before using that suffix array to compute the its BWT.
- The BWT output from your program **should be written to the file called** `outputbwt.txt`. (See sample output file for Task 1 in supporting material.)

Task 2 of 3

In this task you will write a program to **invert** any given BWT of a string and recover the original reference string from it. More formally, let $\mathcal{S}$ be the original (reference) string, which will be **hidden** from you. You will be given instead its BWT, formally $\mathcal{L} = \text{BWT}(\mathcal{S})$, as a text file. Your program should invert the given BWT string $\mathcal{L}$ to recover the original string $\mathcal{S}$.

Strictly follow these specifications when implementing Task 2

- Your program for this task **should be called** `invBWT.py`.
- Your program **must** accept a BWT file (same format as `outputbwt.txt` in Task 1). The supporting material provides two files containing BWT of two diffrent strings which your program should process. Specifically:

  bwt1000001.txt A file containing the BWT of a reference string made of 1 million characters (plus 1 special terminal character '$'). All characters (other than the special character) in this string are either 'A' or 'C' or 'G' or 'T'

  cipher.txt A file containing a BWT of a reference string made of 75 characters (+ 1 special terminal character '$'). All characters of this reference string are **ASCII** characters.
- Your inversion program **must** implement the Last (Column)-to-First(Column) mapping (or LF-mapping) approach introduced to you in Week 7 lecture.
- The output from your inversion program **should be written to the file called** `originalstring.txt`.

Efficiency Hint!

You should carefully consider the efficiency of all the key components of your program while implementing LF-mapping. For instance, LF-mapping involves computing the number of occurrences of some character '`x`' within the BWT in the range $[1, i)$. If implemented naively, the inversion has a $O(n^2)$-time complexity, where n is the length of the string. This makes inversion of a long BWT string impractical.

A slightly more efficient approach (with some trade-off of space/memory) would be to preprocess the BWT and have some auxiliary information containing the counts of each unique symbol (in the BWT) stored at regular intervals (analogous to '*milestones*'). Then the number of occurrences of any character '`x`' within the range $[1, i)$ of the BWT can be calculated by looking up the count of '`x`' up to the nearest milestone in constant time, and then adding to it, by computing on the fly, the count in the remaining part of the range from that nearest milestone.

Task 3 of 3

We have also seen in the week 7 lecture how the BWT serves as an **index** to perform **exact pattern matching** in time proportional to the pattern length. In this task you will implement the described exact pattern matching algorithm. You will use the BWT file `bwt1000001.txt` from Task 2 and its inverted `originalstring.txt` to complete this task.

Strictly follow these specifications when implementing Task 3

- Your program for this task **should be called** `patternMatching.py`.
- Your program should directly deal with the BWT file given in your supporting material `bwt1000001.txt` and its recovered/inverted `originalstring.txt`. (That is, these file names can be hard-coded into your program.)
- Additionally, your program should take as input a file containing a list of patterns which it will search one-by-one using the BWT index. (See `shortpatterns.txt` file in the supporting material; the string in each line is a pattern your program will have to search for using the BWT.)
- All output from this program should be written to the file called `outputTask3.txt`. For each pattern, your output should contain
  1. the number of occurrences of that pattern in the original text.
  2. the position(s) where that pattern appears in the original text (in a **sorted order**), if any.

  (See supporting material for the output format.)

```
-=o0o=-
  END
-=o0o=-
```