FIT2004 S1/2015: Lab questions for week 8

THIS PRAC IS ASSESSED!
(Weight: 6 Marks)

CLASS: This programming exercise has to be prepared in advance. Your 3 hour lab is mainly to iron out bugs, clarify any doubts you have with your demonstrator, and finalize your implementation. Once past the first hour of your assigned lab (or earlier if someone volunteers), your demonstrator will come to you based on a random draw and assess your work. This lab assignment should be implemented in Python programming language. 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 objectives. 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 assignment contain two independent tasks. The first task will enhance your practical learning of sorting a large number of keys under certain assumptions, using radix sort. It will give you more practice to read a given set of instructions (algorithm) and convert it into a well implemented program. The second task, independent of the first, will help you consolidate your understanding of *Burrows-Wheeler Transform* (BWT) of strings that will be introduced in Lecture 7.1. This transformation is extremely useful and underpins modern pattern matching on very large reference strings (that run into terabytes). It is one of the modern string data structures that every computer science student would benefit from knowing. This second task involves the inversion of a BWT that will be clearly explained in the Lecture on Monday 20 April 2015.

Supporting Material

For this exercise use the supporting material (uploaded under the Week 8 section) on Moodle.

1 Background for Task 1

All the sorting algorithms that we explored in your unit so far (Selection, Insertion, Merge, Heap and Quick sorts) were **comparison based** sorting algorithms, as they all (at some stage

in their respective algorithms) explicitly involved comparing two keys (integers for example) to decide their relative order of precedence. In general, the worst case time complexities of these algorithms varied from $O(n\log(n))$ to $O(n^2)$.

In contrast, *Radix sort* is a **non-comparison based** sorting algorithm. It does not need any explicit comparisons between keys. In addition, the worst case time complexity of radix sort is $O(n)$, under the assumption that the keys come from a fixed alphabet and are of some **constant width**, w, that is insignificantly smaller than the number of keys, n, being sorted (i.e., $w \ll n$).

There are many variants of radix sort, but in this assignment you will consider the *Least significant digit (**LSD**) radix sort*. Given n keys all of fixed number of digits (or characters/bits/byte/radices etc.), LSD radix sort is very simple to understand and works as follows. Looping from the least significant digit to the most significant digit of the keys,* LSD radix groups the keys based on the current digit alone, while retaining the original order of the keys when the digits happen to be the same.†

Below is an illustration of how the LSD radix sort works. Let us consider a set of $n = 12$ keys each containing a fixed width $w = 5$ characters. The `red` coloured columns below are the digits (radices) based on which the entire set of keys are being sorted at each step, starting form the least and progressively proceeding to the most significant digit. It is **important** to observe that at each step, when the digits (radices) are the same then the *relative order of keys with same digits* before and after the grouping remain the same.† The `blue` coloured columns show the digits that are being progressively sorted from the least to the most significant digits until the set of keys are fully sorted.

↓		↓		↓		↓		↓		
TGCTC		GAGAA		GAGAA		TGACC		GAGAA		CCCTA
TGACC		CCCTA		CGTAA		TCATG		TATCG		CGATG
GTGTT		CGTAA		CGGAC		CGATG		TCATG		CGGAC
GAGAA		TGCTC		TGACC		CCCTA		CCCTA		CGGGC
TATCG		TGACC		TATCG		TGCTC		TGACC		CGTAA
GGTTT	→	CGGGC	→	CGGGC	→	GAGAA	→	CGATG	→	GAGAA
CGGGC		CGGAC		CCCTA		CGGAC		TGCTC		GGTTT
TCATG		TATCG		TGCTC		CGGGC		CGGAC		GTGTT
CCCTA		TCATG		TCATG		GTGTT		CGGGC		TATCG
CGATG		CGATG		CGATG		CGTAA		CGTAA		TCATG
CGTAA		GTGTT		GTGTT		TATCG		GGTTT		TGACC
CGGAC		GGTTT		GGTTT		GGTTT		GTGTT		TGCTC

In the above example, irrespective of the number of keys, n, the algorithm takes (at most) $w = 5$ linear passes to sort the whole set of keys, giving a $O(n)$ worst-case time complexity.‡

Notice also that there are no comparisons necessary in this algorithm, as the sorting of keys **using the red colored digits** at each step can be done by a simple grouping (into bins/buckets corresponding to each digit/radix alone) without any explicit comparisons of values between pairs of digits.

*Recall that keys are all of the same fixed width.

†This makes LSD radix a *stable* sorting algorithm.

‡Space complexity, depending on the implementation, varies from $O(1)$ to $O(n)$, although for this exercise you are free to implement this in $O(n)$-space, without any penalties for (space) efficiency.

Assessment Task 1

In your supporting material directory, you will find a file entitled '`longstring.txt`'. This string contains characters coming from an alphabet containing four letters: `{A,C,G,T}`. Let us define the term k-mer to be any (contiguous) substring of length k within a larger source string. In this task you will use 16-mers from the source string. Any 16-mer from this four letter alphabet can be represented as a unique integer in a base-4 number system. In fact, if you think about it, these integers fit snugly within a 32-bit integer word on your computers. We will call such integers corresponding to 16-mers, '*integer encoded* 16-*mers*'.

Your task 1 Write a program to read the long string in the file stated above. Further, by sliding across this source string using a window of size $k = 16$, your program should collect the integer encoded 16-mers *and* their corresponding indexes in the source strings where they occur. Once this is done, implement LSD radix method to sort this collection of (integer encoded 16-mer, index) pairs using the integer encoded 16-mers as the **keys** to be sorted. Your program should write to an output file '`sorted16mers.txt`' that should contain the sorted 16-mers as **decoded character strings** in the letters from the original alphabet and their corresponding indexes in the source string.

Efficiency issues

Since you are sorting keys with fixed-width (32-bit) integers, use a radix size that is 1 Byte (equivalent to 8 bits or 4 characters). This will imply that you can sort the 32-bit (4-Byte) keys in 4 linear passes on the collection. It is an expectation that your implementation should be $O(n)$ time. However, you are free to use no more than $O(n)$ space for this task, although it is possible to implement LSD radix in $O(1)$ space.

Do the following as a Non-assessed task

Once you have completed this assignment, even if it means you come back to doing this in later weeks, please undertake a comparison of the performance of your LSD radix implementation with one of $O(n \log(n))$ algorithms (eg. Quicksort) on the same data set. Which is quicker? By how much?
Caveat: One thing we should all remember is that the radix sort works under the assumption that the keys are of a very small fixed width and the alphabet size is also fixed. This is not generally true in all real world cases, hence, often Quicksort and its myriad variants are more *versatile* and may be the right sorting algorithm to use more broadly.

2 Background for Task 2 —This task is independent of Task 1!

This task is directly based on the Week 7 Lecture 1[§] on 'magical' *Burrows-Wheeler Transform* (BWT) of some given reference string/text, and the method to invert a BWT to reproduce the original text.

In this exercise you will write a program to **invert** any given BWT of a string over a fixed alphabet, and recover the original reference string from it. More formally, let $\mathcal{S}$ be the original

[§]This lecture will be delivered on Monday 20 April 2015

(reference) string, which will be hidden from you. Given to you is $\mathcal{L} = \text{BWT}(\mathcal{S})$. Your program should invert the given BWT string $\mathcal{L}$ to recover the original string $\mathcal{S}$, **SPECIFICALLY** using the Last (Column)-to-First(Column) mapping (or LF-mapping) approach introduced to you in the lecture (see l7.1.pdf on moodle).

The supporting material uploaded on Moodle provides two plain text files each containing a BWT string. Specifically:

bwt1000001.txt A file containing the BWT of a reference string made of 1,000,000 (+ 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.

Assessment Task 2

Your task 2 Write a program that accepts a BWT file (provided to you in the supporting material) as an argument and is able to recover correctly the original reference string from the supplied BWT. Your program must output the recovered string to a file '`originalstring.txt`'.

Efficiency issues

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 intractable (for example, the BWT given in the file bwt1000001).

A slightly more efficient approach (with some trade-off of space/memory) would be to preprocess the BWT so that the counts of each unique symbol (in the BWT) are stored at regular intervals (analogous to '*milestones*'). Then the number of occurrences of any character '`x`' within the range $[1, i)$ can be calculated by looking up the count of '`x`' upto 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.

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