Week 8 Studio Sheet

(To be completed during the Week 8 studio class)

Objectives: The tutorials, in general, give practice in problem solving, in analysis of algorithms and data structures, and in mathematics and logic useful in the above.

Instructions to the class: Aim to attempt these questions before the tutorial! It will probably not be possible to cover all questions unless the class has prepared them in advance. There are marks allocated towards active participation during the class. You **must** attempt the problems under **Assessed Preparation** section **before** your tutorial class and give your worked out solutions to your tutor at the start of the class – this is a hurdle and failing to attempt these problems before your tutorial will result in 0 mark for that class even if you actively participate in the class.

Instructions to Tutors:

1. The purpose of the tutorials is not to solve the practical exercises!
2. The purpose is to check answers, and to discuss particular sticking points, not to simply make answers available.

Supplementary problems: The supplementary problems provide additional practice for you to complete after your tutorial class, or as pre-exam revision. Problems that are marked as **(Advanced)** difficulty are beyond the difficulty that you would be expected to complete in the exam, but are nonetheless useful practice problems as they will teach you skills and concepts that you can apply to other problems.

Assessed Preparation

Problem 1. Write the suffix array for the string `ACACIA$` and compute its Burrows-Wheeler transform.

Problem 2. Show the steps in the inverse Burrows-Wheeler transform for the string `N$RSOOCIMPSE`.

Studio Problems

Problem 3. Write the suffix array for the string `GATTACA$` and compute its Burrows-Wheeler transform.

Problem 4. Show the steps in the inverse Burrows-Wheeler transform for the string `TWOIPPR$ENO`.

Problem 5. Consider the string `woolloomooloo$`. The BWT of this string is `ooolooooolwml$`. Show the steps taken by the pattern matching algorithm for the following patterns:

(a) `olo`

(b) `oll`

(c) `oo`

(d) `wol`

Problem 6. Consider the prefix doubling algorithm applied to computing the suffix array of `JARARAKA$`. Write the partially-sorted suffix array after the length two prefixes have been sorted. Write the corresponding rank array and perform the next iteration of prefix doubling, showing the partially-sorted suffix array for the length four prefixes.

Problem 7. The *minimum lexicographical rotation* problem is the problem of finding, for a given string, its cyclic rotation that is lexicographically least. For example, given the string `banana`, its cyclic rotations are `banana`, `ananab`, `nanaba`, `anaban`, `nabana`, `abanan`. The lexicographically (alphabetically) least one is `abanan`. Describe how to solve the minimum lexicographical rotation problem using a suffix array.

Problem 8. Consider the following rank array, obtained during the prefix doubling algorithm applied to the strings `ABBAABABBA$`. At this point in the algorithm, we have just finished updating the ranks for some value of k.

String	A	B	B	A	A	B	A	B	B	A	$
Suffix ID	1	2	3	4	5	6	7	8	9	10	11
Rank	4	6	5	3	4	5	4	6	5	2	1

(a) Determine what the current value of k is. In other words, what lengths of strings are the ranks currently comparing?

(b) Describe in detail how the prefix doubling algorithm would compare the following pairs of suffixes during the next iteration (for strings of length $2k$):

- 1 and 2
- 1 and 5
- 2 and 8
- 3 and 9
- 1 and 7

Supplementary Problems

Problem 9. A string S of length n is called *k-periodic* if S consists of n/k repeats of the string $S[1..k]$. For example, the string `abababab` is two-periodic since it is made up of four copies of `ab`. Of course, all strings are n-periodic (they are made of one copy of themselves!) The *period* of a string is the minimum value of k such that it is k-periodic. Describe an efficient algorithm for determining the period of a string using a suffix array.

Problem 10. Write psuedocode for an algorithm that converts the suffix tree of a given string into the suffix array in $O(n)$ time.

Problem 11. (Advanced) Recall the pattern-matching algorithm that uses the Burrows-Wheeler transform of the text string. One downside of this algorithm is the large memory requirement if we decide to store the occurrences $\text{Occ}(c,i)$ explicitly for every position i and every character c. In this problem we will explore some space-time tradeoffs using *milestones*. Suppose that instead of storing $\text{Occ}(c,i)$ for all values of i, we decide to store it for every k^{th} position only, i.e. we store $\text{Occ}(c,0)$, $\text{Occ}(c,k)$, $\text{Occ}(c,2k)$, $\text{Occ}(c,3k)$, ... for all characters c.

(a) What is the space complexity of storing the preprocessed statistics in this case?

(b) What is the time complexity of performing the preprocessing?

To compute Occ(c,i), we will take the value of Occ(c,j) where j is the nearest multiple of k up to i and then manually count the rest of the occurrences in $S[j+1..i]$

(c) What is the time complexity of performing a query for a pattern of length m?

(d) Describe how bitwise operations can be exploited to reduce the space complexity of the preprocessed statistics to $O\left(\frac{|\Sigma|n}{w}\right)$ where w is the number of bits in a machine word, while retaining the ability to perform pattern searches in $O(m)$.

Problem 12. (Advanced) A given suffix array could have come from many strings. For example, the suffix array 7, 6, 4, 2, 1, 5, 3 could have come from `banana$`, or from dbxbxa$, or many other possible strings.

(a) Devise an algorithm that given a suffix array, determines any string that would produce that suffix array. You can assume that you have as large of an alphabet as you need.

(b) Devise an algorithm that given a suffix array, determines the lexicographically least string that would produce that suffix array. You can assume that you have as large of an alphabet as required. Your algorithm should run in $O(n)$ time

Problem 13. (Advanced) A useful companion to the suffix array is the *longest common prefix* array. It contains for each suffix in sorted order, the length of the longest prefix that is shared by that suffix and the one lexicographically preceding it. Formally, it contains:

$$\text{LCP}[i] = \{\text{The maximum value of } k \text{ such that } S[\text{SA}[i]..(\text{SA}[i]+k)] = S[\text{SA}[i-1]..(\text{SA}[i-1]+k)]\}$$

For example, consider our favourite string `banana$`. The common prefixes are highlighted in the figure below.

```
$
a$
ana$
anana$
banana$
na$
nana$
```

The suffixes `a` and `ana` share the prefix `a`, the suffixes `ana` and `anana` share the prefix `ana`, and the suffixes `na` and `nana` share the prefix `na`. The longest common prefix array would therefore be $0,0,1,3,0,0,2$.

(a) Describe an $O(n^2)$ algorithm to compute the LCP array (this should be easy)

(b) Recall that rank$[i]$ denotes the order of suffix i in the suffix array. Explain why

$$\text{LCP}[\text{rank}[i]] \geq \text{LCP}[\text{rank}[i-1]] - 1.$$

(c) Use the fact given in (b) to write an $O(n)$ algorithm to compute the LCP array.[1]

(d) Describe how the LCP array can be used to solve the longest repeated substring problem

(e) Describe how the LCP array can be used to count the number of distinct substrings of a string

[1] This is known as Kasai's algorithm.