Week 7 Tutorial Sheet

(To be completed during the Week 7 tutorial 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. Draw a prefix trie containing the strings `cat`, `cathode`, `canopy`, `dog`, `danger`, `domain`. Terminate each string with a $ character to indicate the ends of words.

Problem 2. Draw a suffix tree for the string `ABAABA$`.

Tutorial Problems

Problem 3. Describe an algorithm that given a sequence of strings over a constant-size alphabet, counts how many different ones there are (i.e. how many there are ignoring duplicates). Your algorithm should run in $O(T)$ time where T is the total length of all of the strings.

Problem 4. Draw a suffix tree for the string `GATTACA$`.

Problem 5. Describe how to modify a prefix trie so that it can performs queries of the form "count the number of strings in the trie with prefix p". Your modification should not affect the time or space complexity of building the trie, and should support queries in $O(m)$ time, where m is the length of the prefix.

Problem 6. Describe how to implement predecessor queries in a trie. The predecessor of a string S is the lexicographically greatest string in the trie that is lexicographically less than or equal to S. Your data structure should perform predecessor queries in $O(n+m)$ time, where m is the length of the query string, and n is the length of the answer string (the predecessor). It is possible that a string has no predecessor (if it is less than all of the strings in the trie), in which case you should return **null**.

Problem 7. You are given a sequence of n strings $s_1, s_2, ..., s_n$ each of which has an associated weight $w_1, w_2, ..., w_n$. You wish to find the "most powerful prefix" of these words. The most powerful prefix is a prefix that maximises the following function

$$\text{score}(p) = \sum_{\substack{s_i \text{ such that } p \\ \text{is a prefix of } s_i}} w_i \times |p|,$$

where $|p|$ denotes the length of the prefix p. Describe an algorithm that solves this problem in $O(T)$ time, where T is the total length of the strings $s_1, ..., s_n$. Assume that we have three strings `baby`, `bank`, `bit` with weights 10, 20 and 40, respectively. The score of prefix `ba` is $2\times10+2\times20 = 60$, the score of prefix `b` is $1\times10+1\times20+1\times40 = 70$ and the score of prefix `bit` is $3 \times 40 = 120$.

Supplementary Problems

Problem 8. Given a string S and a string T, we wish to form T by concatenating substrings of S. Describe an algorithm for determining the fewest concatenations of substrings of S to form T. Using the same substring multiple times counts as multiple concatenations. For example, given the strings S = `ABCCBA` and T = `CBAABCA`, it takes at least three substrings, e.g. `CBA + ABC + A`. Your algorithm should run in $O(n+m)$ time, where n and m are the lengths of S and T. You can assume that you have an algorithm to construct suffix tree in linear time.

Problem 9. Describe an algorithm for finding a shortest unique substring of a given string S. That is, finding a shortest substring of S that only occurs once in S. For example, given the string `cababac`, the shortest unique substrings are `ca` and `ac` of length two. Your algorithm should run in $O(n)$ time, where n is the length of S. You can assume that you have an algorithm to construct suffix tree in linear time.

Problem 10. Describe an algorithm for finding a shortest string that is not a substring of a given string S over some fixed alphabet. For example, over the alphabet `{A,B}` the shortest string that is not a substring of `AAABABBBA` is `BAA` of length three. Your algorithm should run in $O(n)$ time, where n is the length of S.

Problem 11. (Advanced) Prefix tries also have applications in processing integers. We can store integers in their binary representation in a prefix trie over the alphabet $\{0, 1\}$. Use this idea to solve the following problem. We are given a list of w-bit integers $a_1, a_2, ..., a_n$. We want to answer queries of the form "given a w-bit integer x, select one of the integers a_i to maximise $x \oplus a_i$, where $\oplus$ denotes the XOR operation. Your queries should run in $O(w)$ time.