Week 12 Applied Sheet

Objectives: The applied sessions, 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: You should actively participate in the class. The preparation problems are not assessed. We strongly recommend that everyone tries to solve the preparation problems before the applied session as you will benefit the most from the applied session if you come properly prepared. However, you can still attend the applied session if you have not solved those problems beforehand.

Instructions to Tutors: The purpose of the applied session is not to solve the practical exercises! 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 applied session, 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.

Implementation checklist

It will be most beneficial for your learning if you have completed this checklist **before** the applied class.

By the end of week 12, write Python code for a Trie structure (recommended to use a class). It should have an insert method and a search method.

Problems

Problem 1. **(Preparation)** 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. **(Preparation)** Draw a suffix tree for the string `ABAABA$`.

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 perform 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 an algorithm that given a string S of length n, counts the number of distinct substrings of S in $O(n)$ time. You can assume that you can build suffix trees in $O(n)$ time.

Problem 7. Earlier, we learned about the longest common **subsequence** problem. Now we consider a similar, related problem, the *longest common substring* problem. Given two strings s_1 and s_2, your goal is to find a string that is a substring of both s_1 and s_2 and is as long as possible. Describe how a suffix tree can be used to solve this problem. Your algorithm should run in $O(n+m)$ time, where n, m are the lengths of s_1, s_2 respectively. You may assume that you can construct a suffix tree in linear time.

Hint: It is often easier to think about how to solve problems in a trie context, and then convert your solution to work in a tree.

Problem 8. Given a trie and a query string, we want to find the lexicographically greatest string in the trie which is lexicographically less than or equal to the query string.
Your data structure should perform these 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 some queries have no string which satisfies the criterion above (if it is less than all of the strings in the trie), in which case you should return **null**.

Example: Suppose the trie contains "aaa", "aba", "baa". If the query is "ba", the result is "aba". If the query is "baa", the result is "baa".

Problem 9. 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 $3\times40=120$.

Supplementary Problems

Problem 10. 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 11. 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 12. 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 13. 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.

Problem 14. (Advanced) Given a list of w-bit integers $a_1, a_2, ..., a_n$, find the maximum possible cumulative XOR of a subarray $a_i, ..., a_j$, i.e. maximise

$$a_i \oplus a_{i+1} \oplus ... \oplus a_{j-1} \oplus a_j,$$

where $\oplus$ denotes the XOR operation. Your algorithm should run in $O(nw)$ time. Hint: Use the fact that $x \oplus x = 0$, and the ideas from Problem 13.

Problem 15. (Advanced) In a previous applied class, we wrote a dynamic programming solution to the following problem. Given a target string S of length n and a list L of m strings of length at most n, find the minimum number of strings from L to concatenate to form S. A word from L may be used multiple times. We devised an $O(n^2 m)$ algorithm for this problem. Describe how you can use a prefix trie to speed up this solution to $O(n^2+nm)$ time.

Problem 16. (Advanced) Recall that we deal with suffix trees because suffix tries are very space inefficient and slow to produce. It is easy to see that $O(n^2)$ is an upper bound on the size of a suffix trie of a string of length n, since there are n suffixes of length at most n. Prove that it is also a lower bound, i.e. that there exists a string of length n such that its suffix trie takes $\Omega(n^2)$ space.