COS 226 — Fall 2016 — Design Exam Solution

These solutions are provided as is, and may contain typos. Please double check by asking a question on Piazza. This document tries to provide several ways of solving each problem, but there are alternate solutions, usually partially correct only, which also obtained partial credit, but are not mentioned here.

Exercise 1 (Prefix-free codes)

Two full solutions for the problem:

Binary tree solution

- A *binary tree* (or equivalently in this case a *2-trie* or *trie* or even a *TST*), in which each word is inserted by looking sequentially at each character: 0 goes to the left and 1 goes to the right; finally the node corresponding to the last character, the terminal node, of a word is set to contain the value **True**.
- The codes are prefix-free if the insertion of a word never traverses a node containing the value **True**, and if the insertion of a word always results in the creation of a new terminal node.
- Runtime: W (total number of characters), and space: W.

Sorting solution

- Equivalently, by using a string sorting algorithm (LSD, MSD, 3-way quicksort), we can also first sort the codes, and then sequentially traverse: if one code is a prefix of another it will appear alphabetically right before the code in question.
- The only pitfall to avoid is the situation where two prefixes have a common prefix (which is allowed), such as {010, 011}.
- Runtime: W and space: N
- Remarks: W refers to the total number of characters, not the maximum length of a word (so NW is incorrect, although it obtained partial credit); additionally, using any other sorting algorithm did not provide sufficiently efficient runtime.

Pitfalls include forgetting to sort before sequentially filtering, sorting with a wrong algorithm, or considering that all characters between two words need to be unique (they do not).

Exercise 2 (1D nearest neighbor)

Floor and ceiling. The best solution:

- Insert all the points in any type of *balanced tree* (such as a LLRB tree).
- As described in the course's lecture and library, we can then retrieve the value of the element directly below x and directly above by using the *floor* and *ceiling* functions—respectively provides the nearest element that is smaller, or larger than a given x (that does not need to be in the tree).
- Finally to compute the nearest neighbor of x, we compute both the floor and ceiling then return the one that is closest (in absolute distance) to x.

Rank and select. Another simple solution consists in calling r = rank(x), which will get the rank (order from 1 to N) of x—if x is not in the tree, it will provide the rank of the closest element. The calling select(r) returns the value of the actual element.

In both cases, it was easier to cite the functions (and it was not necessary to explain how to implement them), but it was also possible to describe how those functions operate instead.

The biggest pitfall was being vague with how the tree is searched: when traversing the tree, if you recurse both on the left of a subtree and on its right, it is easy to reach a linear runtime (which is not desirable).

Fatal solutions tried to use inappropriate tools, for instance: a heap, graph, array, *etc*..

Exercise 3 (Shortest path with landmark)

Given a graph G, and one fixed vertex x, the landmark query computes, given two vertices u and v, the *length of* shortest path between u and v that goes through x. Because the general problem (given in part B) is a bit tricky, a preliminary question was asked in part A, which is a simplification of the problem.

It is also important to notice that we only seek to compute the *length of the shortest path*.

In part A:

- With Dijkstra's algorithm, compute the shortest path from u (source) to x (landmark), and thus its length.
- Compute the shortest paths tree starting from x, the landmark.
- With this tree (which can be stored as a symbol table in hash table, in which every vertex is associated with its shortest distance to x), you can then return shortestDistance(u, x) + shortestDistance(x, v) in constant time under reasonable assumptions.

In part B:

- The idea is similar, the only difference is we must now be able to accept any choice for u, the source vertex, which was fixed in part A.
- To do this, we have to compute the *reverse graph* G^R, and then we compute the shortest paths tree starting from x; this will look at the inverse paths, and so compute the distances for the shortest paths from all possible sources u to x.
- We can then sum the two parts of the path as we did previously.

Exercise 4 (MST with weight restrictions)

Again, the problem in this exercise is broken down in two parts (but the more interesting, and harder problem, is in the second part).

For part A: the weights are three positive double values (which means that we cannot use a string sorting algorithm to beat the N log N lowerbound we are trying to beat).

- **Kruskal solution (sort edges, add in increasing order when doesn't create a cycle).** Instead of sorting, we are able to determine, in constant time, the order of the three weights. Then we can scan all edges and only look at the smallest weight edges, then repeat for the middle edges, and largest edges. In the end, the runtime is V log* V, because we must use a Union Find data structure when checking if an edge creates a cycle.
- **Prim solution (start at vertex and grow a minimum tree).** The modification to Prim's algorithm is that instead of using a priority queue, we use three “normal” queues, or an array of three queues. Each queue contains a different of edge. We can compute in constant the time the relative order of the three types of weights, and so we can deplete each queue in order.

For part B: using Prim is a bad idea, because the weights need to be resorted every time a new vertex is added (remember that Prim can only pick among the edges that are connected to the connected component that Prim is growing).

Therefore the best solution uses Kruskal + *radix sort*, or some other linear time string sort, and that is why the values are integers—because it is well known that integers can be sorted in linear time.