SOLUTIONS: Algorithms

Comprehensive Exam: Analysis of Algorithms (60 points) **Autumn 1994**

Problem 1

First we compute $\alpha = \log_6 3$ and compare n^α it with n. In our case, $n = \Omega(n^{\alpha+\epsilon})$ for sufficiently small constant ϵ. Moreover, the "regularity condition" is satisfied, *i.e.* $3n/6 < \beta n$ for a constant β (for example, $\beta = 3/4$ works). Hence, we can conclude that $T(n) = O(n)$.

Problem 2

Binary relations on n elements have 1-to-1 correspondence with 0/1 $n \times n$ matrices. Entry $A_{ij} = 1$ if and only if (i,j) pair belongs to the relation. Symmetric means that $A_{ij} = A_{ji}$, and irreflexive means that the diagonal of A is 0. Hence, in order to determine a relation we need to set $n(n-1)/2$ entries in the matrix. There are $2^{n(n-1)/2}$ possibilities.

Problem 3

If the edge with reduced weight is in the tree, then there is nothing to do. If it is outside the tree, then adding it to the tree creates a cycle. To construct a new tree, we delete the largest weight edge on this cycle. This can be implemented in $O(n)$ time.

It remains to prove that the resulting tree is MST. We need a bit of notation. Let w denote the original edge weights and w' denote the new edge weights. Denote the deleted edge by e_1 and the new edge by e_2. Let T denote the original MST and let T' denote the computed MST, i.e. $T' = T - e_1 + e_2$.

Assume there exists an MST T'' where $w'(T'') < w'(T')$. Observe that T'' has to use e_2, otherwise T is not MST. Consider the cycle that includes e_1 and e_2 in $T + e_2$. One of the edges, say e_3, on this cycle is not in T''. Add it to T'' and remove e_2. The weight of the resulting tree is at most

$$\begin{aligned} w'(T'') + w'(e_3) - w'(e_2) &\leq w'(T'') + w(e_1) - w'(e_2) \\ &< w'(T') + w(e_1) - w'(e_2) \\ &= w(T) \end{aligned}$$

Thus, we got a tree that does not use e_2, but which is cheaper than T, which is a contradiction.

Problem 4

Our data structure will be a complete binary tree with $2^{\lceil \log_2 n \rceil}$ leaves. Each node of the tree holds a single number in addition to the pointers to the parent and its children, and hence the total storage requirements are $O(n)$.

Given a point $x \in [1 \ldots n]$, to compute the value of the sum of the already revealed functions at x, we start at the leaf that corresponds to x and traverse the tree up, adding all the numbers that we encounter up to the root. Clearly, this takes $O(\text{depth})$ of the tree, *i.e.* $O(\log n)$.

Assume we are given a new function that is 1 from 1 to s and 0 from $s+1$ to n. Represent s as a binary number with $\lceil \log_2 n \rceil$ digits. Traverse the tree starting at the root, going left if the digit is 0 and going right if a digit is 1. Each time we go right, we add 1 to the number stored at the left child of the current node. This takes $O(\text{depth})$ of the tree, *i.e.* $O(\log n)$.

Problem 5

Observe that the question corresponds to probing an array of n elements where all elements up to k are 0 and the rest are 1. A crash corresponds to probe returning "1".

5a Probe at $i\sqrt{n}$ for $i = 1, 2, \ldots$, until the first time a probe returns 1. This identifies a range $i^*\sqrt{n} \leq k \leq (i^*+1)\sqrt{n}$ for some i^*. Explore this range by linearly probing $i^*\sqrt{n}, i^*\sqrt{n}+1, \ldots$. Both stages of this strategy use at most $O(\sqrt{n})$ probes.

5b Assume we are allowed q crashes. Probe the array at $O(n^{1/q})$ equally spaced places, starting from 1, until the first time a probe returns 1. This identifies a range of size $O(n^{1-1/q})$ where k might be. Again, probe at most $O(n^{1/q})$ equally spaced places, starting at the smallest index of the identified range, until the first 1 is found. This identifies a range of size $O(n^{1-2/q})$. Continue until the size of the range is $O(n^{1/q})$. Search this range linearly.

At each one of the q stages of this strategy we used $O(n^{1/q})$ probes, and thus we used $O(qn^{1/q})$ probes total.

5c We will use the "adversary" argument. In other words, the adversary will let the algorithm probe, and decide on the value of k as it goes along. The only constraint is that this value should be consistent with all the answers to the probes.

Initially, k is in the range of 1 to n. The algorithm has to issue probes that force the adversary to pick a specific k. The algorithm can not terminate as long as the adversary has any freedom in picking k, since in this case the value of k returned by the algorithm will not be correct.

We say that a probe is of "type 1" if it is closer than $\sqrt{n}$ to the largest index probed up until now. Probes below the maximum-probed index do not bring any new information and hence can be discarded. We classify the rest of the probes as "type 2".

As long as the algorithm uses "type 1" probes, the adversary returns 0. Note that each one of these probes can decrease the range of possible values for k by at most $\sqrt{n}$, and hence the algorithm needs at least at least $O(\sqrt{n})$ type 1 probes to decrease the range of possible values for k to below $\sqrt{n}$.

Thus, if the number of probes is below $\Omega(\sqrt{n})$, the algorithm makes at least one type 2 probe. As a response to the first such probe, the adversary returns 1. At this moment, the only restriction on k that the adversary has is that k is larger than the largest probed index that returned 0 and smaller than the index probed by the type 2 probe. From now on the only valid algorithm is to linearly probe this range. This is true because if the algorithm misses an index, say i, then the adversary sets k to i and returns 1 at the next probe. Hence, in this case, the total number of probes is $O(\sqrt{n})$.