G22.3250-001 Honors Analysis of Algorithms October 31, 2025

Solutions to Problem Set 2,3

Name: Daniel Wichs, Srihari Narayanan *Due: October 17, 2025*

Problem 1

Given a tree with (possibly negative) weights assigned to its vertices, give a polynomial time algorithm to Find a subtree with maximum weight. Note that a subtree is a connected subgraph of a tree.

Solution: Let T be a tree with root v and let $v_1, \dots, v_t$ be the children of v. Let $T_1, \dots, T_n$ denote the n trees rooted at $v_1, \dots, v_n$.

The key realization is that the maximal-weight subtree of T – call it $\mathbf{maxWeight}(T)$ – either includes the root v or excludes it. In the latter case, it must just be the maximal-weight subtree of one of the children:

$$\mathbf{maxWeightSans}(T) = \begin{cases} v & v \text{ is a leaf, } weight(v) \geq 0 \\ \emptyset & v \text{ is a leaf, } weight(v) < 0 \\ \max\{\mathbf{maxWeight}(T_1), \dots, \mathbf{maxWeight}(T_n)\} & \text{otherwise} \end{cases}$$

In the former case, if the maximal-weight subtree of T does include v, then the question is what other "subtrees" to add under v. To do so, we define $\mathbf{maxWeightWith}(T)$ to be the maximal-weight subtree of T which *includes* the root v. Then this is simply the union of all $\mathbf{maxWeightWith}(T_i)$ which have weight greater than 0 together with v.

Then

$$\mathbf{maxWeightWith}(T) = \begin{cases} v & \text{if } v \text{ is a leaf} \\ \bigcup_{\geq 0}\{\mathbf{maxWeightWith}(T_i)\} \cup v & \text{otherwise} \end{cases}$$

Lastly, putting this together we get:

$$\mathbf{maxWeight}(T) = \max\{\mathbf{maxWeightWith}(T), \mathbf{maxWeightSans}(T)\}$$

We note that the number of distinct recursive calls is at most $3|T|$ (three recursive functions which can get applied to any node) so, by cashing the answers on each recursive call (i.e. using dynamic programming) the run time of the algorithm is $3|T|$.

□

Problem 2

Let $G = (V, E)$ be a directed acyclic graph (i.e. it does not contain any directed cycle).

1. Prove that the graph must have a vertex t that has no outgoing edge.

2. Suppose $|V| = n$. A topological ordering of the acyclic graph is a labeling of its vertices by integers from 1 to n such that

   - Any two distinct vertices receive distinct labels.
   - Every (directed) edge goes from a vertex with a lower label to a vertex with a higher label.

   Give a polynomial time algorithm to Find a topological ordering of the graph.

3. Fix a node t that has no outgoing edge. For every node $v \in V$, let $P(v)$ be the number of distinct paths from v to t. Define $P(v) = 0$ if no such path exists and define $P(t) = 1$ for convenience. Give a polynomial time algorithm to compute $P(v)$ for every node v.

Solution:

1. Pick an arbitrary vertex v and follow an arbitrary path "away from" v until you reach a vertex t that has no outgoing edges. Since there are no cycles, the path will never visit any vertex twice and hence the above process must terminate after some finite number of steps proving the existence of t.

2. See http://www.cs.nyu.edu/courses/fall06/G22.3520-001/lec14.pdf. Slides 20-23. Or CLRS pages 549-551.

3. Run a topological sort on G. Let label(v) be the value assigned to v by the sort, and $N(v)$ be the neighbor-set of v. Then

$$\text{npaths}(v,t) = \begin{cases} 0 & \text{label}(v) > \text{label}(t) \\ 1 & v = t \\ \sum_{v' \in N(v)} \text{npaths}(v', t) & \text{otherise} \end{cases}$$

□

Problem 3

Let $p(1), \ldots, p(n)$ be positive real numbers. A k-shot strategy S is a sequence of at most k ordered integer pairs $(b_1, s_1), \ldots, (b_m, s_m)$, with $1 \le b_1 < s_1 < b_2 < s_2 \ldots < b_m < s_m$. Let $\text{val}(S) = \sum_{i=1}^{m} (p(s_i) - p(b_i))$. For any k we want to find the k-shot strategy which maximizes val(S).

Solution:

Let

$$M_{(i,j)} := \max_{i \le b < j} (p(j) - p(b))$$

be the maximum amount of money you can make in one buy/sell transaction with sell date j and buy date $b \ : \ i \le b < j$. It is easy to compute $M_{(i,j)}$ for all $1 \le i \le j$ in time $\mathcal{O}(n^3)$.

Now the best k-shot strategy in the days $i, i+1, \ldots, n$ must consist of making the best possible transaction with a sell date prior to some date b and the following the best $k-1$-shot strategy in the days $b+1, \ldots, n$. Formally,

$$BEST(k,i) = \begin{cases} 0 & k \leq 0 \text{ or } i \geq n \\ \max_{i \leq b \leq n} \left(M_{(i,b)} + BEST(k-1, b+1) \right) & \text{otherwise} \end{cases}$$

There are at most nk distinct values of $BEST(k,i)$ that need to be computed and each runs in time n for a total run time of $\mathcal{O}(n^3 + kn^2)$. The above algorithm needs to be modified to return the actual strategy rather than just the profit, but this just requires some simple book-keeping. □

Problem 4

Given a graph G with some edge weights such that the all cycles in G have positive weight, together with vertices s, t find the number of shortest paths from s to t.

Solution:

Use Bellman-Ford to find the length best(u,t,n) of the shortest path from any node u to the node t which uses fewer than n edges. Now we define npaths(u,t,n) to be the number of shortest-paths from u to t using fewer than n edges. Then npaths$(u,t,n) = \sum_{w \in S(u)}(\text{npaths}(w,t,n-1))$ is the sum of the number of shortest paths from w to t using fewer than $n-1$ edges, for all neighbors w such that some shortest path from u to t goes through w. We call this set $S(u)$. But $w \in S(u) \Leftrightarrow c(u,w) + \text{best}(w,t,n-1) = \text{best}(w,t,n)$ (where $c(u,w)$ is the cost of the edge (u,w)).So it is easy to check if a vertex is in $S(u)$. Therefore we get

$$\text{npaths}(u,t,n) = \begin{cases} 1 & u = t \\ 0 & u \neq t, n = 0 \\ \sum_{w \in S(u)} \text{npaths}(w,t,n-1) & \text{otherwise} \end{cases}$$

There are $|V|^2$ distinct problems each of which takes at most $|V|$ steps to compute for a run-time of $\mathcal{O}(|V|^3)$. □

Problem 5

Given n jobs such that job i takes time t_i and must finish before deadline d_i find a schedule which runs the maximum number of jobs.

Solution:

1. First we show that there is an optimal schedule in which the jobs run in order of increasing deadlines. Imagine that S is an optimal schedule and that, in S, tasks do not run in order of increasing deadlines. Then there must be two tasks which run adjacent in S such that the later one has an earlier deadline. But we can always switch the order of these tasks and they finish within their deadlines (and the rest of the schedule is unchanged). By performing this re-ordering operation many times, we get a schedule where jobs run in order of increasing deadlines.

2. Sort tasks in order of increasing deadlines. In sorted order, let the deadlines be $d_1, \ldots, d_n$ and the run-times $t_1, \ldots, t_n$. Let $\text{sched}(i, s)$ be the optimal (value) of the schedule for tasks $i, i+1, \ldots, n$ starting at time s. Then the optimal schedule either runs the first task, and then runs the optimal schedule of the remaining $n-1$ tasks from time $s + t_i$, or it does not run the first task and just runs the optimal schedule of the remaining tasks from time s.

$$\text{sched}(i, s) = \begin{cases} 0 & s > d_n \text{ or } i > n \\ \max(1 + \text{sched}(i+1, s+t_i), \text{sched}(i+1, s)) & \text{otherwise} \end{cases}$$

Figuring out the actual schedule requires simple additional book-keeping which we skip. We see that there are at most $n \times d_n$ possible problems each of which takes $\mathcal{O}(1)$ time so, using dynamic programming the run-time of the above recursion is $\mathcal{O}(n \times d_n)$ together with sorting we then get a run time of $\mathcal{O}(n \log n + nd_n)$ (where d_n is the maximal deadline).

□

Problem 6

An independent set Iin a graph is called maximal if the graph does not contain an independent set I' such that $I \subseteq I'$, $|I| < |I'|$. Given a tree on n vertices, and an integer $0 \leq k \leq n$, give a polynomial time algorithm to determine whether the tree has a maximal independent set of size k. (Hint: Design an algorithm that solves the problem for all possible values of k.).

Solution:
As the main idea, each node v of the tree will store two sets:

maxWith(v) : the set of all k such that the subtree rooted at v contains some maximal independent subset of size k which *includes* v.

maxSans(v) : the set of all k such that the subtree rooted at v contains some maximal independent subset of size k which *excludes* v.

We also define **maxAny**$(v) =$ **maxSans**$(v) \cup$ **maxWith**(v).

It is clear that, if v is a leaf then **maxWith**$(v) = \{1\}$ and **maxSans**$(v) = \{0\}$. If v has children $v_1, \ldots, v_m$ then

$$\mathbf{maxWith}(v) = \{1 + t_1 + t_2 + \ldots + t_m \ : \ t_1 \in \mathbf{maxSans}(v_1), \ldots, t_n \in \mathbf{maxSans}(v_n)\}$$

since an independent subset containing v must not include its children, and the maximal subset of the subtree rooted at v must contain some maximal independent subset of the trees rooted at $v_1, \ldots, v_m$. Also

$$\mathbf{maxSans}(v) = \{t_1 + t_2 + \ldots + t_m \ : \ t_1 \in \mathbf{maxAny}(v_1), \ldots, t_n \in \mathbf{maxAny}(v_n)\}$$

Now we just need to recursively compute **maxAny**$(root)$ and check if k is included in the answer. By caching the values at the nodes (i.e. using dynamic programming) we see that we actually only solve two problems per node. Also, the amount of work done at the nodes is only the merging of the sets **maxWith**(v_i) **maxSans**(v_i) computed for the children v_i. Each such set is of size at most $|V|$ and hence the merging as well as the total algorithm run in polynomial time. □

Problem 7

Binary search of a sorted array takes logarithmic search time, but the time to insert a new element is linear in the size of the array. We can improve the time for insertion by keeping several sorted arrays. Specifically, suppose that we wish to support SEARCH and INSERT on a set of n elements. Let $k = \lceil \log(n+1) \rceil$, and let the binary representation of n be $< n_{k-1}, \ldots, n_0 >$. We have k sorted arrays $A_0, \ldots, A_{k-1}$, where the size of array A_i is 2^i. Each array is either full or empty, depending on whether $n_i = 1$ or $n_i = 0$, respectively. The total number of elements held in all k arrays is therefore $\sum_{i=0}^{k-1} n_i 2^i = n$. Although each individual array is sorted, elements in different arrays bear no particular relationship to each other.

a. Describe how to perform the SEARCH operation for this data structure. Analyze its worst-case running time.

b. Describe how to perform the INSERT operation. Analyze its worst-case and amortized running times.

c. Discuss how to implement DELETE.

Solution:

a. We binary search on each sorted array. For array A_i, of size 2^i, this takes $O(i)$ time, so in total the search operation takes at most $\sum_{i=0}^{k-1} i = O(k^2) = O(\log^2 n)$ time.

b. Suppose the smallest b arrays $A_0, \ldots, A_{b-1}$ are all populated, and $A_b = \emptyset$. We implement INSERT by merging these b sorted arrays along with the element x to be added, and putting the resulting sorted array in A_b. Specifically, the most efficient implementation merges x with A_0, then the resulting array with A_1, then with A_2, and so on. The worst case running time of this is the cost of running merge sort on multiple arrays, which is $c \sum_{k=1}^{b} 2^k = c(2^{b+1} - 2) = O(n)$ for some constant c. As for amortized cost, define the potential function

$$\Phi(A) = -c' \sum_{k | A_k \neq \emptyset} k 2^k$$

where c' is some constant that will be determined later. Clearly, we have $\Phi(\emptyset) = 0$. When we add the element x to A to get A', if A_b is the smallest empty array, we have that

$$\Phi(A') - \Phi(A) = c' \left(-b2^b + \sum_{k=0}^{b-1} k2^k \right) \leq c' \left(-b2^b + (b-1) \sum_{k=0}^{b-1} 2^k \right) = c'(-b2^b + (b-1)(2^b - 1)) \leq -c'2^b$$

Thus, if we choose $c' = 2c$, we get an amortized cost of

$$c(2^{b+1} - 2) + \Phi(A') - \Phi(A) = c(2^{b+1} - 2) - c'2^b = \Theta(1)$$

c. Let x be the element we want to delete. We first search for x, taking time $O(\log^2 n)$, and suppose we find x in A_b. Let A_c be the smallest non-empty array. We take any element y out of A_c and replace x with y in A_b, and then move y to its correct place in A_b, taking time $O(n)$ in the worst case of $c = k - 1$. We then take the remaining elements out of A_c and populate arrays $A_{c-1}, A_{c-2}, \ldots, A_0$, again taking time $O(n)$ in the worst case of $c = k - 1$.

□

Problem 8

Consider an ordinary binary search tree augmented by adding to each node x the field size$[x]$ giving the number of keys stored in the subtree rooted at x. Let α be a constant in the range $\frac{1}{2} \leq \alpha < 1$. We say that a given node x is *α-balanced* if size[left$[x]$] $\leq \alpha \cdot$ size$[x]$ and size[right$[x]$] $\leq \alpha \cdot$ size$[x]$. The tree as a whole is *α-balanced* if every node in the tree is α-balanced.

a. Given a node x in an arbitrary binary search tree, show how to rebuild the subtree rooted at x so that it becomes $\frac{1}{2}$-balanced. Your algorithm should run in time $\Theta(\text{size}[x])$, and it can use $O(\text{size}[x])$ auxiliary storage.

b. Show that performing a search in an n-node α-balanced BST takes $O(\log n)$ worst-case time.

For the remainder of this problem, assume $\alpha > \frac{1}{2}$. Suppose that INSERT and DELETE are implemented as usual for an n-node binary search tree, except that after every such operation, if any node in the tree is no longer α-balanced, then the subtree rooted at the highest such node in the tree is "rebuilt" so that it becomes $\frac{1}{2}$-balanced.

We shall analyze this rebuilding scheme using the potential method. For a node x in a binary search tree T, we define $\Delta(x) = |\text{size}[\text{left}[x]] - \text{size}[\text{right}[x]]|$, and we define the potential of T as

$$\Phi(T) = c \sum_{x \in T : \Delta(x) \geq 2} \Delta(x)$$

where c is a sufficiently large constant that depends on α.

c. Argue that any binary search tree has nonnegative potential and that a $\frac{1}{2}$-balanced tree has potential 0.

d. Suppose that m units of potential can pay for rebuilding an m-node subtree. How large must c be in terms of α in order for it to take $O(1)$ amortized time to rebuild a subtree that is not α-balanced?

e. Show that inserting a node into or deleting a node from an n-node α-balanced tree costs $O(\log n)$ amortized time.

Solution:

a. We perform an in-order traversal on the BST rooted at x to store it as a sorted array. We then construct a $\frac{1}{2}$-balanced BST out of these elements by first rooting our tree at the median element (or one of the two medians if we have an even-size array). Next, we recurse on the elements to the left and right of m to form left$[m]$ and right$[m]$, respectively. A simple proof by induction shows that the resulting tree is $\frac{1}{2}$-balanced. Additionally, size we spend $O(1)$ time per element (to put it into the sorted array, and then into the new BST), we clearly spend $O(\text{size}[x])$ time in total, and $O(\text{size}[x])$ space as well since all the data structures used have linear size.

b. To see that binary search on this BST takes $O(\log n)$ time, we realize that, after each step, we have at most an α-fraction of nodes left to search. Therefore we obtain the recursion

$$T(n) \leq 1 + T(\alpha n)$$

One way to proceed from here is by induction, where we assume the strong inductive step $T(\alpha n) \le c \log(\alpha n)$ for some constant c we will pick later. Then we have

$$T(n) \le 1 + T(\alpha n) \le 1 + c\log(\alpha n) = 1 + c\log\alpha + c\log n$$

Since we have $\alpha < 1$, we get $\log(\alpha) < 0$, so we pick c large enough so that $1 + c\log\alpha \le 0$ and $T(n) \le c\log n$, as desired.

Another way to proceed is to calculate the runtime directly. Since we have at most an α-fraction of vertices left after each step, we take at most k steps where $\alpha < n\alpha^k \le 1$. Rearranging this and taking the log of both sides gives $(k-1)\log\alpha > -\log n$, which can be rearranged again to give $k < \frac{\log n}{\log(\frac{1}{\alpha})} + 1 = O(\log n)$, as desired.

c. Since $\Delta(x)$ is obtained by taking an absolute value, any BST must have nonnegative potential if $c > 0$. Now, for any node x in a $\frac{1}{2}$-balanced BST, we have that $\Delta(x) \le 1$ since the subtree rooted at x is maximally balanced (can formally justify this by working out the inequalities), so by the definition of Φ, $\frac{1}{2}$-balanced BSTs have potential 0.

d. Let x' be the highest node in the subtree rooted at x that is not α-balanced, and let $m =$ size$[x']$. The drop in potential after we rebalance the subtree rooted at x' can be bounded from below by $c\Delta(x')$, since the potential after rebalancing must be 0 by part c.. Now, suppose WLOG we have size[left$[x']$] $> \alpha m$. We then have size[right$[x']$] $< (1-\alpha)m - 1$, so we can lower bound $c\Delta(x')$ in turn by $c((2\alpha - 1)m - 1)$. Since we are assuming m units of potential pays for rebalancing, we need the drop in potential to be at least m, which follows from requiring $c((2\alpha-1)m-1) \ge m$ and $c \gtrsim \frac{1}{2\alpha - 1}$.

e. Without rebalancing, inserting or deleting a node into the BST takes $O(\log n)$ time without amortization. With amortization, we add at most c units of potential per level, since we add or remove at most one element from every subtree. By b., our BST has $O(\log n)$ levels, so the increase in potential is also $O(\log n)$. With rebalancing, by d., the decrease in potential fully pays for the cost of rebalancing if we choose c carefully enough, so the amortized cost of insertion and deletion is $O(\log n)$ as desired.

$\square$