CS161 Final Solution

Problem 1: Reviewing the Basics (20 points)

- Prove that $(n+1)^2 = O(n^2)$ by giving constants n_0 and c in the definition of O-notation.

$$\begin{aligned}(n+1)^2 &= n^2 + 2n + 1 \\ &\le n^2 + 2n^2 + n^2 \quad \text{for } n \ge 1 \\ &\le 4n^2\end{aligned}$$

Therefore, $(n+1)^2 \le 4n^2$ for $n \ge 1$. By choosing $n_0 = 1$ and $c = 4$, we prove that $(n+1)^2 = O(n^2)$, as per the definition of O.

- Consider a priority queue that supports the operations **`Insert`** and **`Extract-Min`**. Argue that in the worst case, if we perform a mixture of n **`Insert`** and **`Extract-Min`** operations on the priority queue, there is at least one operation that takes $\Omega(\lg n)$ time. Assume that the order of elements in the priority queue is determined by comparisons only. (Hint: Recall that for any set of numbers, at least one number must equal or exceed the average.)

Can sort $n/2$ numbers by doing $n/2$ **`Insert`**s and $n/2$ **`Extract-Min`**s (n operations all together). But comparison sorts take $\Omega(n \lg n)$ time, or in this case $\Omega\left(\frac{n}{2}\lg\frac{n}{2}\right) = \Omega(n \lg n)$. The average time for this set of n operations (and hence for the worst mix of n operations, which is at least as bad as this one) is therefore $\Omega(\lg n)$, so at least one operation takes $\Omega(\lg n)$.

- When performing arithmetic on large numbers, it is often necessary to do the calculations in smaller parts. In this problem, you're going to multiply two n-bit numbers, X and Y. Assume that $X = A\ 2^{n/2} + B$ and $Y = C\ 2^{n/2} + D$, where A, B, C, and D are each $(n/2)$-bit numbers.

  - We can calculate XY using the fact that $XY = AC\ 2^n + (AD + BC)2^{n/2} + BD$. Assume that adding and subtracting two k-bit numbers can be done in $O(k)$ time, and also assume that multiplying any number by 2^k, requiring exactly k left shifts, can also be done in $O(k)$ time. Give a recurrence for $T(n)$, the time required to multiply two n-bit numbers, and then solve the recurrence.

    There are three additions, and two perfect-power-of-two multiplications, and each takes $O(n)$. Since there are five such operations, all of the non-recursive elements to the computation take $O(n)$ time.

    One needs to compute AC, AD, BC, and BD recursively. The problem size for each is half that of the original, so the recurrence relation is therefore easily expressed as:

    $$T(n) = 4T(n/2) + O(n)$$

    Master method says that $T(n) = \Theta(n^2)$

  - We can also compute $XY = AC\ 2^n + ((A - B)(D - C) + AC + BD)2^{n/2} + BD$, reusing the calculations for AC and BD. Give a recurrence for $T(n)$, the time needed to multiply two n-bit numbers using this method, and solve the recurrence.

    While the actual mathematics is a little message, and the number of simple $O(n)$ time operations goes up (six additions and two power-of-two multiplications), the number of recursively computer products goes down by one. Here, the number of subproblems is important, because as we saw from the first version, the recursion time was the dominating factor.

    $$T(n) = 3T(n/2) + O(n)$$

    Master method says that $T(n) = \Theta(n^{\lg 3}) = O(n^{1.7})$

  - Which method is better, asymptotically?

    The second one. It is **polynomially** smaller than the first version.

Problem 2: True or False (24 points)

For each of the following problems, answer **`True`** or **`False`**, unless you don't know the answer. Correct answers will receive 2 points, incorrect answers will receive -2 points, and no responses will be awarded 0 points. If you are guessing, it is to your advantage to mark **`No Response`**.

○ **True**
● **False**
○ **No Response**

For any asymptotically nonnegative function $f(n)$, we have that $f(n) + o(f(n)\lg n) = \Theta(f(n))$. (That's a little o, not a big O).

There are a ton of functions in between $f(n)$ and $f(n)\lg n$: Take $f(n)\lg\lg n$ as an example.

○ **True**
● **False**
○ **No Response**

The recurrence relation $T(n) = 3T(n/3) + \lg n$ is not covered by any of the three cases of the Master Theorem.

Covered by case 1, since $n^{\log_3 3}$ is polynomially larger than $\lg n$.

○ **True**
● **False**
○ **No Response**

All sorting algorithms require $\Omega(n\lg n)$ in the worst case.

Only comparison sorts are bound from below. Counting, radix, and bucket sort all operate in linear time, because they are allowed to make assumptions about the range of integers involved.

● **True**
○ **False**
○ **No Response**

The worst-case running time of **`Randomized-Quicksort`** on an array of size n is $\Omega(n^2)$.

Even a random number generator might pick the worst pivot with every single call to **`Partition`**.

● **True**
○ **False**
○ **No Response**

The $(\lg n)^{th}$ smallest number of n unsorted numbers can be determined in $O(n)$ worst-case time.

Use the truly linear version of **`Select`**—the one using the median of medians approach to choose a good pivot every time.

● **True**
○ **False**
○ **No Response**

A legal binary search tree on n integer keys in the range from 1 to n^2 can be synthesized in $O(n)$ worst-case time.

Alas, yes! Treat each of the n integers as radix-n numbers, use counting sort to sort the integers in two passes, and then build a sorted linked list (setting all left pointers to **`NIL`**).

○ **True**
● **False**
○ **No Response**

The number of legal binary search tree structures on the first n counting numbers is $O(2^n)$.

Much larger: $O\left(\frac{4^n}{n^{3/2}}\right)$.

○ True
● False
○ No Response

The problem of determining the optimal order for multiplying a chain of matrices can be solved by a greedy algorithm, since it displays the optimal substructure and overlapping subproblems properties.

The algorithm uses a dynamic programming approach.

○ True
● False
○ No Response

We can always improve the asymptotic running time by choosing dynamic programming over top-down recursion.

Not always. Dynamic programming only helps us beyond a constant factor when a purely recursive algorithm recomputes the solution to lots of subproblems.

● True
○ False
○ No Response

The topological sort of an arbitrary directed graph $G = (V,E)$ can be computed in $O(\max(V,E))$ time that.

Yes, the true running time is $O(V+E) \equiv O(\max(V,E))$. The correction stating that the graph was acyclic was presumably available to everyone, but if not, please contact Jerry ASAP.

● True
○ False
○ No Response

In $O(E + V \lg V)$ time, Dijkstra's algorithm with Fibonacci heaps can be used to compute shortest paths from a source vertex to all other vertices in a graph $G = (V,E)$ with nonnegative edge weights.

Straight from the book.

○ True
● False
○ No Response

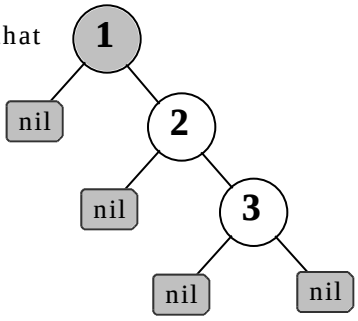
In a directed acyclic graph $G = (V,E)$ with nonnegative edge weights, the presence of an edge $(u,v) \in E$ implies that $d[u] \le d[v]$ is true at all times during Bellman-Ford.

Not always the case. If multiple edges are incident to vertex v, then v may very well be closer to the source vertex than u is. Consider the case where (s, v) is the edge of least weight, and s isn't even incident to u.

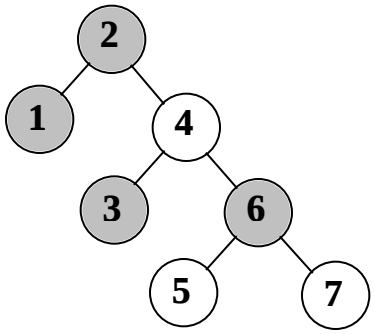
Problem 3: Red-Black Trees (26 points)

a) Consider the ordered insertion of the keys 1, 2, 3, 4, 5, …, n-1, and finally n, into an initially empty red-black tree. Which insertion is the first to force a rotation? Draw the state of the tree just prior to the rotation, including all nil nodes and all colors.

The third one will, as shown by just brute force illustration. Notice that the root is always black, that when 2 is inserted, it's just fine to be left red, but when 3 is inserted, it is colored red and therefore provokes a color change. Because 3 has no uncle, the red-black properties must be restored via a left-rotation around the 1-2 link.

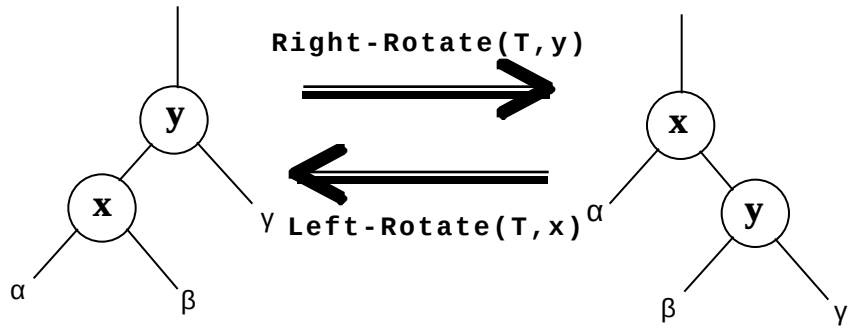


b) [5 points] Draw the legal red-black tree storing 7 keys of maximum height. (Hint: what must the black-height of such a red-black tree be? What does that tell you about the tree height?) Make sure your tree is a legal red-black tree.



Here, the `nil` nodes are left out, but all nil nodes are black. Note that the black-height everywhere is two, and that red nodes are used quite on the right side of the tree to help realize the maximum imbalance.

c) Consider a red-black tree on n keys that maintains, in each node of the tree, the height of the node. (Recall that the height of a node is the maximum number of edges on any simple path from the node to one of its descendant leaves.) Describe how to update this height information efficiently when a rotation occurs. Analyze the running time of your update algorithm.



- The heights of all nodes in α, β, and γ stay the same.
- The heights of x, y, and their ancestors may change.

Let h be the height field of a node. In the following, h of a subtree refers to the height field of the root of the subtree.

For the right rotation above, update the heights as follows:

First, $h[y] \leftarrow \max(h[\beta], h[\gamma]) + 1$.

Second, $h[x] \leftarrow \max(h[\alpha], h[y]) + 1$

Lastly, update y's ancestors' heights. For each node on the path from y to the root, set the node's height to one more than the maximum of its two children's heights. (It may not be necessary to propagate new heights all the way to the root. If x's new height is the same as y's old height, no propagation is really necessary. And if the height of some ancestor does not change, there is no need to continue updating the rest of the path toward the root. Stopping early, however, does not affect the worst-case time.

The worst-case run time is $O(\lg n)$ because:

- $O(\lg n)$ heights are updated, because the longest possible path of updates is the height of the tree, and the height of an n-node red-black tree is $O(\lg n)$.
- Each update step takes constant time.

d) You've already shown how to maintain a dynamic set of integers that supports **`Insert`**, **`Delete`**, **`Search`**, and **`Min-Gap`**. Leveraging off of either your answer to this problem set question, or off of my solution, explain how you would augment the dynamic set even further to support the set operation **`Nearest-Neighbors`**, which returns the two entries which are closer together than any other pair of numbers (thereby defining the **`Min-Gap`** value). The **`Nearest-Neighbors`** operation should run in constant time.

Assume that each node maintains the nearest-neighbor interval between all keys in the subtree rooted at that node. The nearest neighbor interval can be recomputed for a node in constant time, provided that the min, max, and min-gap fields are also maintained as in the original problem. The computation of the nearest neighbor interval is always as follows:

$$\text{nearest}-\text{neighbors}[x] = \text{narrowest}-\text{of}\begin{cases} \text{nearest}-\text{neighbors}[\text{left}[x]] & (\text{or } [-\infty,\infty] \text{ if left}[x] = \text{nil}) \\ \text{nearest}-\text{neighbors}[\text{right}[x]] & (\text{or } [-\infty,\infty] \text{ if right}[x] = \text{nil}) \\ [\text{key}[x], \text{min}[\text{left}[x]]] & (\text{or } [\text{key}[x],\infty] \text{ if left}[x] = \text{nil}) \\ [\text{max}[\text{right}[x], \text{key}[x]] & (\text{or } [-\infty,\text{key}[x]] \text{ if right}[x] = \text{nil}) \end{cases}$$

Because this information can be updated in constant time based on information available from itself and its two children, the **`Insert`**, **`Delete`**, **`Search`**, and **`Min-Gap`** running times are unaffected, and Nearest-Neighbors can simply return the value of nearest-neighbors[root[T]], or some sentinel value if the tree is either empty or a singleton.

Problem 4: Maximizing profit (20 points)

Suppose you have one machine and a set of n jobs $a_1, a_2, a_3, \ldots, a_n$ to process on that machine. Each job a_k has a processing time t_k, a profit p_k, and deadline d_k. The machine can only process one job at a time, and job a_k must run uninterruptedly for t_k consecutive time units. If a_k is completed by its deadline, you receive a profit p_k, but if it is completed after its deadline, then you receive a profit of 0. Present an algorithm to find the maximum profit attainable by any schedule. Imagine that the first job is launched at time 0, and that the deadlines come as expiration times relative to the start of the first job. All jobs require between 1 and m time units, inclusive. You needn't actually construct the schedule.

$$D[j,k] = 0 \qquad j > n$$

$$D[j,k] = 0 \qquad k \geq d_n$$

$$D[j,k] = \max\begin{cases} D[j+1,k] \\ D[j+1,k+t_j] + p_j \left[k + t_j \leq d_j\right] \end{cases} \qquad \text{otherwise}$$

The key observation is that $D[j,k]$ is easily formulated in terms of $D[j+1,k]$ (because sometimes a_j should be skipped altogether) and in terms of $D[j+1,k+t_j]$ (for when $k + t_j \leq d_j$, so that we should see what happens when we do consider this job.) A purely recursive computation would make either one or two recursive calls, depending on whether any profit can be made from the j^{th} job.

To compute $D[1,0]$, create an $n \times mn$ matrix, where the first dimension is determined by the number of jobs, and the second is determined by the maximum amount of time needed to process all of them. The bottom row of the matrix can be filled in from $D[n,mn]$ down through $D[n,0]$, and then the second to last row can be filled in from $D[n-1,mn]$ down through $D[n-1,0]$. Entries in are equal to the entry directly below it, unless including it happens to be more profitable, in which case, the entry would be set to the sum of p_{n-1} and entry in D one down and t_{n-1} to the right. All subsequent rows can be filled up from bottom to top, right to left, and once $D[1,0]$ is filled in, you have your answer. The grid is clearly of size mn^2, and each entry is updated in constant time, provided the entries are filled in the proper order.

Problem 5: Graph Theory (30 points)

a) In as much detail as possible, explain why Dijkstra's algorithm sometimes fails to report the correct set of minimal path distances when some of the edge weights are negative.

When a vertex v is pulled from the priority queue, the shortest-path estimate is supposed to have fully relaxed to the true shortest-path value. If any incoming edges are negative, and any one of those edges hasn't relaxed yet, the shortest-path estimate will actually be too high.

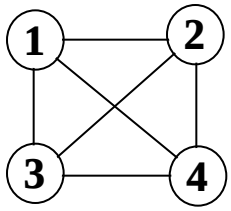
b) Explain why the Bellman-Ford algorithm requires only one additional iteration (beyond the first $|V|-1$) to detect the presence of negative weight cycles? More precisely, why must at least one $d[v]$ value relax on that iteration if there are negative-weight cycles?

The $k+1^{st}$ iteration reduces $d[v]$ estimates based on the results of the k^{th} iteration. If no edges relax during the k^{th} iteration, then all edges are fully relaxed, and no more iterations are needed. When there are negative weight cycles, some $d[v]$ values never bottom out. Because some of these $d[v]$ values inch toward $d[v]=-\infty$ with each iteration, at least one edge must relax on every iteration, else the $d[v]=-\infty$ value could never be realized.

c) Argue that if all of the weights of a graph are positive, then any subset of edges that connects all of the vertices and has a minimum total weight must also be a tree. Give an example to show that the same conclusion does not follow if we allow some weights to be nonpositive.

$|V|-1$ edges or more are required to connect all vertices. However, if the subset costs precisely the same as any minimal spanning tree, then the subset itself must be a tree. If not, we could remove any one of the extra edges beyond the requisite $|V|-1$ to get an edge subset spanning the tree that cost even less than any MST.

If we allow zero- or negative-weight edges, then a minimal-weight spanning graph need not be a tree. The simple example is K_4 where all edges have a weight of zero. Assume the edges are all directed from low number to high, and that all edges weigh in at zero. The full graph itself—six edges and all—costs precisely the same as the MST where only edges leaving vertex 1 are needed.



· A directed graph $G = (V,E)$ is **unipathic** is for any two vertices $u,v \quad V$, there is at most one simple path for u to v. Suppose that a unipathic graph G has both positive and negative weight edges. Design an efficient algorithm to determine the shortest-path weights from a source s to all vertices $v \quad V$ for such a unipathic graph. If some shortest-path weights to vertices reachable from s do not exist, then your algorithm should report that a negative-weight cycle exists in the graph.

The observation is that since there is at most one simple path from s to any vertex v, any method of finding a path from s to v finds the shortest path. Thus we can use breadth-first search from s or depth-first search from s (starting only at s) to find the paths in $O(V+E)$ time. In order to get the shortest-paths weights as well as the paths themselves, augment **`BFS`** or **`DFS`** to store shortest-path estimates for vertices by relaxing the edges traversed, as in Dijkstra's algorithm. This doesn't affect the asymptotic running time of the algorithm.

There are many ways to detect negative-weight cycles. One way is as follows: When the path search via BFS or DFS completes, do the test that is done for negative-weight cycles at the end of **`Bellman-Ford`**. This checks each edge to see if relaxing it would decrease a shortest-path estimate. As estimate decreases at this point if and only if there is a negative weight cycle (for the same reason as in **`Bellman-Ford`**.) This negative cycle detection doesn't affect the $O(V+E)$ running time.