COMPSCI 311: Introduction to Algorithms **Spring 2019**

Homework 3 Solutions

Released 2/21/2019 Due 3/7/2018 11:59pm in Gradescope

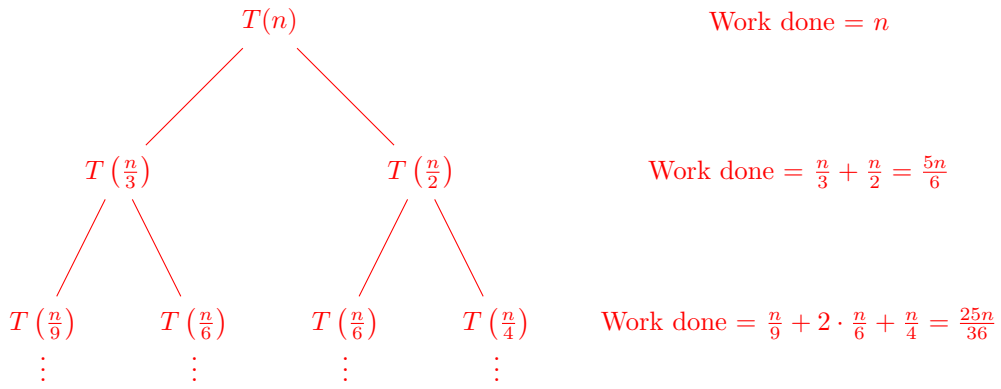
1. **(20 points) Recurrences.** Give asymptotic upper and lower bounds in the following recurrences. Assume $T(n)$ constant for $n \leq 2$. Make your bounds as tight as possible and justify your answers.

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

   (b) $T(n) = T(n-1) + \log n$

   (c) $T(n) = 2T(n/2) + n/\log n$ Fact: $\sum_{k=1}^{n} 1/k = \Theta(\log n)$

   (a) The first few levels of the recursion tree look as follows.



   At each level, the total work done is multiplied by 5/6 (**Think:** why? Formally speaking, this would still require a proof.). We stop at the leaves of the tree (the base cases), which have constant values. The number of leaves is at most $\Theta(n)$ (**Think:** why?), where the value of the function is constant. Therefore, the leaves contribute $O(n)$ work, and together with the rest of the work done on the levels, this adds up to

$$T(n) = n(1 + \frac{5}{6} + \frac{25}{36} + \ldots) + O(n) \leq n \cdot \frac{1}{1 - \frac{5}{6}} + O(n) = 6n + O(n) = O(n).$$

   Here we have used the important identity, for any positive integer k and positive real number $r < 1$, we have

$$1 + r + r^2 + \ldots + r^k \leq \sum_{j=0}^{\infty} r^j = \frac{1}{1-r}.$$

   The lower bound holds trivially, since we need n work just at the first level of the recursion. Therefore, $T(n) = \Theta(n)$.

   By the way, this is very easy to verify, using the upper bound found from the recursion tree. Simply take $T(n) = 6n$ (which we obtained as a bound for the work done on the levels), and check whether the recurrence is satisfied.

(b) We could use recursion trees for this as well, but instead we use the "unrolling" technique shown in discussions. We have, for any $1 \leq k \leq n$,

$$T(n) = T(n-1) + \log n = T(n-2) + \log(n-1) + \log n = \ldots = T(n-k) + \sum_{i=1}^{k} \log(n+1-i).$$

Therefore, using the base case $T(0) = \Theta(1)$, we get, setting $k = n$,

$$T(n) = T(0) + (\log 1 + \log 2 + \ldots + \log n) = \Theta(1) + \log(1 \cdot 2 \cdots n) = \Theta(\log(n!)).$$

To get a more explicit approximation, note that

$$\sum_{i=1}^{n} \log i \leq \sum_{i=1}^{n} \log n = n \log n,$$

while

$$\sum_{i=1}^{n} \log i \geq \sum_{i=n/2}^{n} \log i \geq \sum_{i=n/2}^{n} \log(n/2) = \frac{n}{2} \cdot \log(n/2) = \frac{n}{2}(\log n - 1),$$

where we have used the fact that $\log n$ is a positive, monotone increasing function. It follows that $\log(n!) = \Theta(n \log n)$, showing that $T(n) = \Theta(n \log n)$ as well.

(c) We can use the "unrolling" technique again to get

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + \frac{n}{\lg n} = 2\left(2T\left(\frac{n}{4}\right) + \frac{n/2}{\lg(n/2)}\right) + \frac{n}{\lg n} \\ &= 4T(n/4) + \frac{n}{\lg n - 1} + \frac{n}{\lg n} \\ &\vdots \\ &= 2^k T(n/2^k) + \sum_{i=1}^{k} \frac{n}{\lg n - (i-1)}. \end{aligned}$$

This process stops when $2^k = n$, or when $k = \log n$, with

$$T(n) = n \cdot T(1) + n \cdot \sum_{i=1}^{\log n} \frac{1}{\log n - (i-1)}.$$

This sum is equal to, by a change of variables (**Think:** why?),

$$\sum_{i=1}^{\log n} \frac{1}{\log n - (i-1)} = \sum_{j=1}^{\log n} \frac{1}{j} = \Theta(\log \log n),$$

using the Fact given in the problem. It follows that

$$T(n) = n \cdot T(1) + \Theta(n \log \log n) = \Theta(n \log \log n).$$

2. **(20 points) Greedy Stays Ahead.** You fail to land a good internship for the summer so you end up working in the UMass mail room. The job is really boring. You stand at a conveyor belt and put mail items from the conveyor belt into boxes. It turns out that all of the mail is headed to the CS department! Each box has a fixed limit W on how much weight it can hold, and the items arrive on the conveyor belt one by one: the ith item that arrives has weight w_i. The rules of the job are really draconian: you must fill one box at a time and send it to the CS department before starting on the next box, and you must pack items into boxes in exactly the order they arrive on the conveyor belt.

So, your only real decision is how many items to pack in each box before you send it off to the CS department.

You decide to try a simple greedy algorithm: pack items into the current box in the order they arrive, and, whenever the next item does not fit, send the current box and start a new one.

Is it possible that this will cause you to use more boxes than necessary? That is, could you decrease the overall number of boxes by packing one box less full, so that items somehow fit more efficiently into later boxes?

Prove that, for a given set of items with specified weights, your greedy algorithm minimizes the number of boxes that are needed. Your proof should follow the type of analysis used in the book for the Interval Scheduling Problem: it should establish the optimality of this greedy packing algorithm by identifying a measure under which it "stays ahead" of all other solutions.

Here is some notation and a few definitions to help formulate the problem precisely.

- Assume the items are numbered $1, 2, \ldots, n$ and arrive in order, and that item i has weight w_i.
- Let i_k be the number of items packed in the first k boxes by the greedy algorithm (equivalently, i_k is the number of the last item packed in box k).
- Similarly, consider any optimal solution O and let j_k be the number of items packed in the first k boxes by O.

(a) (3 points, work independently) Create an example where you select a specific value for W, and make up weights for a sequence of n items that requires at least three boxes. Design your example so there are at least two different optimal solutions. Indicate the values $i_1, i_2, \ldots, i_p$ for the greedy solution, as well as the values $j_1, \ldots, j_q$ for a *different* optimal solution.

(b) (2 points, work independently) Write down an inequality that is always true for the quantities i_1 and j_1, and explain your reasoning.

(c) (2 points, work independently) Formulate a "claim": an inequality comparing i_k and j_k that is true for $k \geq 1$, which you will prove by induction.

(d) (10 points) Prove your claim by induction.

(e) (3 points) Now argue that your claim implies that the greedy algorithm is optimal, i.e., that $p = q$ where p is the number of boxes used by the greedy algorithm and q is the number of boxes used by the optimal solution.

(a) (Example omitted)

(b) $i_1 \geq j_1$ because the first box in the greedy algorithm packs as many items as possible

(c) **Claim:** $i_k \geq j_k$ for all $k = 1, 2, 3, \ldots$

(d) Proof by induction on k.

For the base case, define $i_0 = j_0 = 0$. Then $i_k \geq j_k$ for $k = 0$.

For the inductive step, note that the items assigned to box k by the greedy and optimal solutions, respectively, are:

$$A_k = \{i_{k-1} + 1, \ldots, i_k\}$$
$$O_k = \{j_{k-1} + 1, \ldots, j_k\}$$

Assume inductively that $i_{k-1} \geq j_{k-1}$. Suppose for the sake of contradiction that $i_k < j_k$. But this means that A_k is a *strict subset* of O_k. Since the total weight of items in O_k is at most W, the greedy algorithm could have fit more items in truck k, which contradicts the fact that the greedy algorithm packs as many items as possible without exceeding W.

(e) To see why this means the greedy solution is optimal, suppose the optimal solution uses p boxes to pack all n items. Then $j_p = n$, because it ships all n items. But $i_p \geq j_p = n$. This means the greedy solution also fits all n items in p boxes, so it is optimal.

3. **Database Medians.** You are working as a programmer for UMass administration, and they ask you to determine the median GPA for all students. However, student GPAs are stored in two different databases, one for in-state students and one for out-of-state students. Assume there are n students of each type, so there are $2n$ students total. You'd like to determine the median of this set of $2n$ values, which we will define here to be the nth smallest value.

   However, security is very tight, so the only way you can access these values is through queries to the databases. In a single query, you can specify a value k to one of the two databases, and the chosen database will return the kth smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible.

   Give an algorithm that finds the median value using at most $O(\log n)$ queries.

   There may be students with equal GPA, but this does not affect the solution. We only need to use that if the k^{th} smallest number is x, then k numbers are $\leq x$, and the rest are $\geq x$.

   Clearly, if we remove the same number of students that are before the median and after the median, the remaining set will have the same median. The recursive idea is to find the medians in both databases, compare them, and remove the first and last quarter of students, then recurse on the two remaining sets of equal size. Since we can't remove elements from database, we'll keep track of the number of students left to consider in each database (initially n) and the two starting offsets a and b (initially 0).

   Let $k = \lceil n/2 \rceil$ and compare $m_A = A[a+k]$ with $m_B = B[b+k]$ (these are the midpoints if n is odd and the last elements of the first halves if n is even). Suppose $m_A \leq m_B$. Then we claim the median m satisfies $m_A \leq m \leq m_B$. There are $2\lceil n/2 \rceil \geq n$ elements up to $A[b+k]$ and up to $B[b+k]$, all at most m_B, so $m \leq m_B$. Any elements *after* $A[a+k]$ or starting at $B[b+k]$ are at least m_A, and there are $2(n - \lceil n/2 \rceil) + 1 \geq n$ of them. Thus, we have $m \geq m_A$ for the median.

   We claim we can eliminate the first $\lfloor n/2 \rfloor$ elements of A and the same number from the end of B. In B, these are all *after* $B[b+k] = m_B \geq m$. In A, these are up to $A[a+k] = m_A \leq m$ (including $A[a+k]$ if n is even), so at most up to the median. Thus, removing the two sets does not affect the median.

   The symmetric reasoning applies when $m_A \geq m_B$ (and in fact, if $m_A = m_B$, we've found the median).

   The algorithm `median`(n, a, b) finds the median of $\{A[a+1], \ldots, A[a+n], B[b+1], \ldots, B[b+n]\}$. Here, a and b (initially 0) are the starting offsets in databases A and B, and n is the number of elements left to consider in each database (initially, the total element count in each).

```
median(n, a, b)
  if n = 1 then return min(A[a + 1], B[b + 1])          ▷ Base case
  k = ⌈n/2⌉                                             ▷ Find midpoint
  m_A = A[a + k]                                        ▷ median of remaining elements of A
  m_B = B[b + k]                                        ▷ median of remaining elements of B
  if m_A ≤ m_B then return median(k, a + ⌊n/2⌋, b)
  else return median(k, a, b + ⌊n/2⌋)
```

   Each recursive step divides the input size in half and makes one recursive call. There are $\lceil \log n \rceil$ levels of recursion, and each level makes two queries. The overall number of queries is therefore $O(\log n)$.

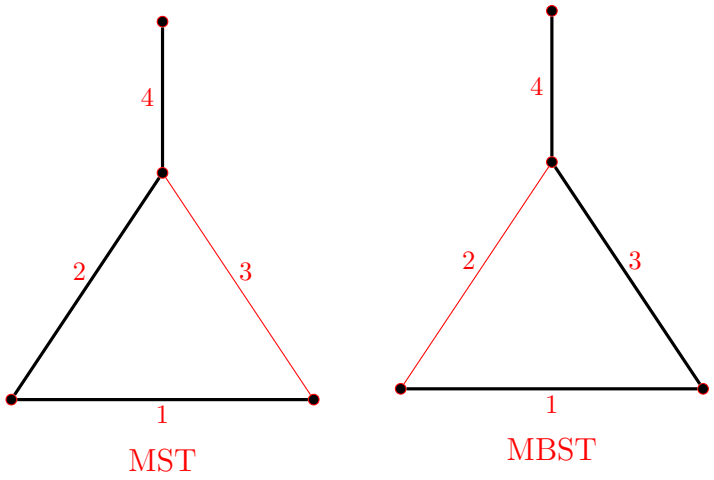
4. **(10 points) Spanning Tree.** Design a spanning tree for which the most expensive edge is as cheap as possible. Let $G = (V, E)$ be a connected graph with n vertices, m edges, and positive edge costs assumed all distinct. Let $T = (V, E')$ be a spanning tree of G; we define the bottleneck edge of T to be the edge of T with the greatest cost.

   A spanning tree T of G is a minimum-bottleneck spanning tree if there is no spanning tree T' of G with a cheaper bottleneck edge.

   (a) Is every minimum-bottleneck tree a minimum spanning tree of G? Prove or give a counterexample.

   (b) Is every minimum spanning tree a minimum-bottleneck tree of G? Prove or give a counterexample.

a) No, a minimum-bottleneck tree need not be an MST of G. For example, consider a graph G which is a triangle with edge weights 1, 2, 3, and a weight-4 edge sticking out of it, as shown below. There is a unique MST for this graph, shown on the left. But the tree shown on the right is a minimum-bottleneck tree as well, with higher cost.



b) Assume the MST of G is not the MBT of G. Observe that the largest edge of the MST must be larger than the largest edge of the MBT, by the definition of the MBT. This means that the two points directly connected by the largest edge of MST are indirectly connected through an edge in the MBT that is smaller than the largest edge of the MST. Adding that smaller edge to the MST and will create a cycle. Deleting the largest edge of this cycle creates a smaller MST, contradicting the MST definition. Therefore, the assumption is false and the MST of G is always a MBT of G.

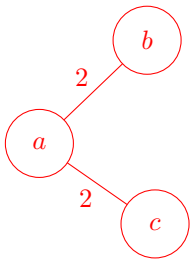
5. **(10 points) More Spanning Tree.** Show that a graph has a unique minimum spanning tree if, for every cut of the graph, there is a unique light edge crossing the cut. Show that the converse is not true by giving a counterexample.

Let us show that any two minimum spanning trees T_1 and T_2 of G are the same tree. Let (u, v) be an arbitrary edge of T_1. From MST property, we know that any edge in an MST is a light edge crossing some cut of the graph. Let $(S, V - S)$ be a cut for which (u, v) is a light edge.

Consider the edge $(x, y) \in T_2$ crossing $(S, V - S)$. (x, y) must exist, as otherwise T_2 would not be a spanning tree. (x, y) must also be a light edge, as otherwise T_2 would not be a minimum spanning tree.

By problem statement, there is an unique light edge crossing any cut of G. Thus, $(u, v) \in T_1$ and $(x, y) \in T_2$ must be the same edge. As (u, v) is an arbitrary edge of T_1, every edge in T_1 is also in T_2 and thus T_1 and T_2 are the same tree.

The converse (if a graph has a unique MST, then light edges for all cuts are unique) is not true, as demonstrated by a counterexample:



Here, the graph is its own (unique) MST, but the cut $(\{a\}, \{b, c\})$ has two light edges (a, b) and (a, c).

6. **(20 points) Shortest Road Trip.** Mary and Tom are taking a road trip from New York City to San Francisco. Because it is a 44-hour drive, Mary and Tom decide to switch off driving at each rest stop they visit. However, because Mary has a better sense of direction than Tom, she should be driving both when they depart and when they arrive (to navigate the city streets). Given a route map represented as a weighted undirected graph $G = (V, E, w)$ with positive edge weights, where vertices represent rest stops and edges represent routes between rest stops, devise an efficient algorithm to find a route (if possible) of minimum distance between New York City and San Francisco such that Mary and Tom alternate edges and Mary drives the first and last edge.

   [Hint: one way to solve this problem is to construct a new graph G' to represent the alternate driving]

   There are two correct and efficient ways to solve this problem. The first solution makes a new graph G'. For every vertex u in G, there are two vertices u_M and u_T in G': these represent reaching the rest stop u when Mary (for u_M) or Tom (for u_T) will drive next. For every edge (u, v) in G, there are two edges in G': (u_M, v_T) and (u_T, v_M). Both of these edges have the same weight as the original.

   We run Dijkstra's algorithm on this new graph to find the shortest path from $NewYorkCity_M$ to $SanFrancisco_T$ (since Mary drives to San Francisco, Tom would drive next if they continued). This guarantees that we find a path where Mary and Tom alternate, and Mary drives the first and last segment. Constructing this graph takes linear time, and running Dijkstra's algorithm on it takes $O(VlogV + E)$ time with a Fibonacci heap (it's just a constant factor worse than running Dijkstra on the original graph).

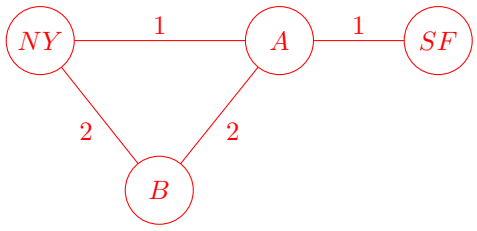
   The second correct solution is equivalent to the first, but instead of modifying the graph, we modify Dijkstra's algorithm. Dijkstra's algorithm will store two minimum distances and two parent pointers for each vertex u: the minimum distance d_{odd} using an odd number of edges, and the minimum distance d_{even} using an even number of edges, along with their parent pointers π_{odd} and π_{even}. (These correspond to the minimum distance and parent pointers for u_M and u_T in the previous solution). In addition, we put each vertex in the priority queue twice: once with d_{odd} as its key, and once with d_{even} as its key (this corresponds to putting both u_M and u_T in the priority queue in the previous solution).

   When we relax edges in the modified version of Dijkstra, we check whether $v.d_{odd} > u.d_{even} + w(u, v)$, and vice versa. One important detail is that we need to initialize $NewYorkCity.d_{odd}$ to ∞, not 0. This algorithm has the same running time as the previous one.

   A correct but less efficient algorithm used Dijkstra, but modified it to traverse two edges at a time on every step except the first, to guarantee a path with an odd number of edges was found. Many students incorrectly claimed this had the same running time as Dijkstra's algorithm; however, computing all the paths of length 2 (this is the square of the graph G) actually takes a total of $O(VE)$ time, whether you compute it beforehand or compute it for each vertex when you remove it from Dijkstra's priority queue.

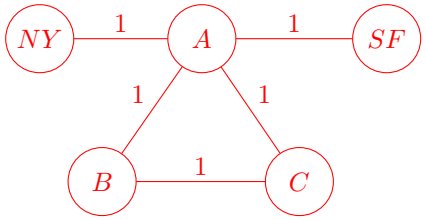
   The most common mistake on the problem was to augment Dijkstra (or Bellman-Ford) by keeping track of either the shortest path's edge count for each vertex, or the parity of the number edges in the shortest path. This is insufficient to guarantee that the shortest odd-edge-length path is found. Here

is an example of a graph where the algorithm fails: once the odd-edge-count path of weight 1 to A is found, Dijkstra will ignore the even-edge-count path of weight 4 to A since it has greater weight. As a result, the odd-edge-count path to SF will be missed entirely.



Another common mistake was to use Dijkstra, and if the path Dijkstra found had an even number of edges, to attempt to add or remove edges until a path with an odd number of edges was obtained. In general, there is no guarantee the shortest path with an odd number of edges is at all related to the shortest path with an even number of edges.

Some algorithms ran Dijkstra, and if Dijkstra found a path with an even number of edges, removed some edge or edges from the graph and re-ran Dijkstra. This algorithm fails on the following graph, where the shortest path with an odd number of edges uses all the edges and vertices (note that we visit A twice; the first time, Mary drives to A, and the second time, Tom drives to A):



One last common mistake was to attempt to use Breadth-First Search to label each vertex as an odd or even number of edges from New York City (or sometimes to label them as odd, even, or both). This does not help: the smallest-weight path with an odd number of edges could go through any particular vertex after having traversed an odd or even number of edges, and BFS will not correctly predict which. Algorithms which returned the correct answer but with exponential running time got less points.