**COMPSCI 311: Introduction to Algorithms** **Spring 2022**

# Homework 3

Released 3/4/2022 Due 3/22/2022 11:59pm in Gradescope

Brendan Cadogan
Collaborators: Nate Courchesne, Benjamin Tufano

**Instructions.** You may work in groups, but you must write solutions yourself. List collaborators on your submission. You are allowed to have at most 4 collaborators. Also list any sources of help (including online sources) other than the textbook and course staff.

If you are asked to give an algorithm, please provide: (a) the pseudocode or precise description in words of the algorithm, (b) an explanation of the intuition for the algorithm, (c) a proof of correctness, (d) the running time of your algorithm and (e) justification for your running time analysis.

**Submissions.** Please submit a PDF file. You may submit a scanned handwritten document, but a typed submission is preferred. Please assign pages to questions in Gradescope.

1. **(20 points) Snowy Roads**
In a mountainous country, roads at high altitudes risk getting blocked because of snowfall. Each road is marked with its altitude. People anxiously wait for the weather forecast which will announce "roads at or above an altitude $H$ will be impassable".
a) Design an algorithm that computes the *highest* altitude $H$ for which the country becomes disconnected (there are two cities between which one cannot travel). Do so efficiently.

A. To solve this problem, we can just use a modified minimum spanning tree algorithm like Prim's or Kruskal's. All we need to do is find the biggest edge weight in the MST and thats our answer. To do this efficiently,if we are using Prim's, as we construct our MST, we can have an integer value to keep track of the greatest edge weight added to our MST. If we are using Kruskal's, the last edge added should always be the greatest, so we can just return that value. Going forward, we'll be using Kruskal's because it is a little easier to code.

```
for i to m do
    let u and v be endpoints of e
    if find(u) != find(v) then
        union(find(u), find(v))
        highest = e.altitude
return highest
```

B. Our intuition was that a MST would give us the greatest edge length that would make the tree disconnected because a MST finds the smallest edges that don't create cycles.

C. We know that the MST of a graph contains all of the lowest edge weights except when there is a cycle, that still keep the graph connected. This means that we can ignore all of the edges outside of the MST because if one of them was the highest that disconnected the graph, it would be included in the MST. Now, we know that snowing in any edge in the MST will disconnect the MST because a MST has the minimum amount of edges to keep the graph connected. Since all edges except cycle edges are bigger than edges in the MST, and since the cycle edges do not matter, and since the MST is the minimum amount of edges to keep the graph connected, we know that the biggest edge in the MST will be the highest altitude to disconnect the country.

D. The time complexity is O(mlogn).

E. The time complexity is O(mlogn) because that is the complexity of Kruskal's when Kruskal's uses the union implementation, and our 2 lines of modification to Kruskal's does not add to the time complexity.

b) Now assume each road has a cost for snow clearing, and a total budget $S$ is available. Find the highest altitude $H$ for which it is not possible to reconnect the cities (establish a route between any two of them) given the snow-clearing budget, or report that this is always possible.

A. To solve this, we use our algorithm in part A, which finds a MST based off of altitude using Kruskal's. Once we have the MST, we want to take out the highest altitude and replace it with the edge in the cut with the lowest cost to plow. This may be the edge we have just removed. Once we have determined the lowest edge in the cut, we'll subtract the cost of it from the budget and if it still positive, we can continue. If it is negative, we'll return the altitude of the edge we tried to replace. We repeat this until we run out of money or until we complete this process for all edges in the original MST. In that secondary case, we report back that it is possible.

```
for i to m do
    let u and v be endpoints of e
    if find(u) != find(v) then
        T < − e
        union(find(u), find(v))
for(int i = 0; i < T.length; i++)
    let u and v be endpoints of T[i].pop and highest be T[i].altitude
    let e be the minimum cost of Cut(u, v)
    if(budget - e.cost ¡ 0)
        return highest
    else
        budget = budget - e.cost
        push e to t[i]
return that it is possible
```

B. This one was hard because what seemed to be the obvious answer of finding the MST by cost instead of altitude was not the answer. What we did know that lead us in the right direction was that the altitude would always be in the MST formed with the lowest altitude weights, so we knew we needed to build off of part A to solve it.

C. First off we know that if it is not possible to keep the country connected, the answer is in the MST from part A. Another thing we know is that it is not the best solution to plow only the edges in the MST because there could be alternative routes that are cheaper and allow us to stay connected for longer. Now we can use greedy stays ahead to prove this. At any instances of the second for loop, we are choosing the cheapest edge in the cut that keeps the graph connected. If we chose a different edge that was equal or costs more than the edge we chose, and if there was only one solution possible, since we plowed a more costly edge than needed to keep the graph connected, it is possible that we run out of money before plowing all of the necessary roads to keep our graph connected.

D. The time complexity is O(mn)

E. The time complexity is O(mn) because the first part is O(mlogn) because it is Kruskal's, but the second part is O(mn) which is bigger than O(mlogn). The second for loop is O(mn) because the for loop is n because there are n edges in the mst, and then the to find e which is the minimum cost edge in the cut is at most m.

2. **(40 points) Recurrences** Compute tight asymptotic bounds for the following recurrences. Justify your answers. Assume appropriate base cases for the unfolding. Hint: use guess (e.g., from unfolding) and verify.

a) $T(n) = \frac{2}{3}T(n/2) + 3T(n/3) + 8T(n/4)$

$T(n) = \Theta(n^2)$
First we will guess that $T(n) = cn^k$
$cn^k = cn^k(\frac{2^{1-k}}{3} + 3^{1-k} + 4^{1.5-k})$
$1 = (\frac{2^{1-k}}{3} + 3^{1-k} + 4^{1.5-k})$
$1 = (\frac{2^{1-2}}{3} + 3^{1-2} + 4^{1.5-2})$
$1 = \frac{1}{6} + \frac{1}{3} + \frac{1}{2}$ which is $1 = 1$
so k =2 and $T(n) = cn^2$ which is $\Theta(n^2)$

b) $T(n) = \frac{1}{3}T(n-1) + \frac{2}{3}T(n-2) + cn^2$

$T(n) = \Theta(n^3)$
First we unravel the recurrence.
$T(n) = \frac{1}{3}(\frac{1}{3}T(n-2) + \frac{2}{3}T(n-3) + c(n-1)^2) + \frac{2}{3}(\frac{1}{3}T(n-3) + \frac{2}{3}T(n-4) + c(n-2)^2) + cn^2$
If we simplify this we get
$T(n) = \frac{1}{9}T(n-2) + \frac{4}{9}T(n-3) + \frac{2}{3}T(n-4) + cn^2 + \frac{1}{3}c(n-1)^2 + \frac{2}{3}c(n-2)^2$
We notice that the back part of this unrolling is a summation which is something we talked about during a lecture.
We know $n^2$ is increasing, so $\int_n^{n-1} x^2dx \leq 2n^2 \leq \int_n^{n-1} x^2dx$
Thus$\frac{c}{2}\int_0^n x^2dx^2 + c(n-1)^2 + ... \leq \frac{c}{2}\int_0^{n+1} x^2dx$ so $\frac{c}{8}n^3 \leq T(n) \leq \frac{c}{8}(n+1)^3$, so $T(n) = \Theta(n^3)$

c) $T(n) = 2T(n/2) + n\log n$ for even $n$, $T(n) = T(n-1) + n\log n$ for odd $n$.

$T(n) = \Theta(nlog^2n)$
First we will solve the even bound using the second case of master theorem. We know that if we have $T(n) = aT(\frac{n}{b}) + f(n)$ and if $f(n) = \Theta(n^{\log_b a}\log^k n)$ then $T(n) = \Theta(n^{\log_b a}\log^{k+1} n)$
We cab then find that $f(n) = \Theta(n^{log_2 2}log^1 n)$
Because of this, we can determine that the even side is $T(n) = \Theta(nlog^2n)$ and that the bound for both is at least $\Omega(nlog^2)$ because the bound for the odd side could possibly be tighter.
Next we'll unroll the odd side of the problem.
$T(2k+1) = T((2k+1)-1) + n\log n$ and $T(2k+1) = T(2T(2k) + n\log n) + n\log n$
This means that for all odds, T(n) becomes even because $2k+1$ becomes $2k$ and since for evens our bound is $T(n) = \Omega(nlog^2n)$, we now know $T(n) = \Theta(nlog^2n)$ because odds and evens are the same.

d) $T(n) = n + \frac{1}{2}\max_{n/3 \leq k \leq 2n/3} T(k) + T(n-k)$

$T(n) = \Theta(n)$
The first thing we want to do is isolate the max part of the recurrence and solve it. We can then determine that it is equal to $2T(\frac{n}{2})$ because we keep spliting n into 2 segments which is n because $((n-k+k) = n$
This makes the whole recurrence become $T(n) = \frac{1}{2}(2T(\frac{n}{2})) + n$ which becomes $T(n) = T(\frac{n}{2}) + n$ which we can solve to using master's theorem$\Theta(n)$.

3. **(20 points) Under Pressure** Two divers are in a flooded mine which has rooms at several levels, connected by shafts. There is a single room at surface level, the entrance. Each room has a single shaft going to a room one level up, and at most two shafts going down. Divers must adjust to pressure at depth; getting between levels $k$ and $k-1$ (with $k \geq 1$) in either direction takes time $k$. Given a diagram of the mine (entrance, rooms and shafts), determine the maximum time it could take for a diver to reach the other one if they got stuck, no matter in which rooms they were initially.

A. In this question, the mine is a binary tree, and the maximum possible time between the two divers is the diameter. So all we need to do is find an algorithm that finds the diameter of a binary tree, which we have learned in previous classes, and modify it so that the time increasing as we go deeper into the mine is accounted for.

```
let maxTime be a global = 0
diameter(node, time)
    if(mine == null)
        return 0
    left = diameter(node− >left, time + 1)
    right = diameter(node− >right, time +1)
    maxTime = max(maxTime, 1 + left + right)
    return time + max(left, right)
maxTime is the answer
```

B. Since we already knew that the farthest path in a binary tree was the diameter of it, we knew that all we had to do was modify out algorithm that finds the diameter to fit our problem.

C. This works because by definition of our function, we visit each node once, and as we go there, we calculate the time each node takes by just adding one to the previous room time. So now we know our algorithm calculates all of the weights properly, we need to prove that it returns the diameter. It does because at every iteration, we return the max of the left or right side of the graph, so by the end we should have the max time from a right and left path which will be the diameter.

D.The run time is $\Theta(n)$

E. The run time is $\Theta(n)$ because the recurrence is $2T(n/2) + 1$ and that solves to $\Theta(n)$.

4. **(20+10 points) Tilted Towers** A city has a row of tall towers that must be demolished – they are old, ugly,
and all of different heights, $H[1..n]$. This is done over several weeks, as follows. An observer goes to the top of each tower. They look along the row and mark the closest taller tower (if any), both left and right. All marked towers are then demolished and the process continues next week, until just one tower is left.
a) Determine in $O(n \log n)$ what towers are marked by each observer in the first round.

A. What we want to do is write a divide and conquer algorithm similar to merge sort. Assuming we are given an array that has the towers sorted by their order instead of by height, we want to divide it into sub problems until we can't divide the sub problems anymore. From this point, we will solve the sub-problems, and then combine them in the same way we divided and then solve the combined solved problems until we have solved the entire problem.
To divide the sub problems, we'll just find the mid point of the arrays and divide it into two arrays. To solve the sub problems, we will compare the two edges of the combined sub problems and mark the bigger one. This will always mark one of the towers, but the tower that was bigger needs to continue looking to the direction of the smaller one, and the smaller tower also needs to look in that direction. What we can do is have them look together with the smaller one checking first, and the bigger one only checking when the smaller one has found a tower to mark. If there isn't a bigger tower to mark, they will join the end tower when it has to search when it is combined unless it doesn't because it is one of the two edges in the original big problem. Next, the bigger tower needs to look in the other direction, along with any towers that were in the other sub problem that are also missing a marked tower in the same direction.

B. Our intuition was to design an algorithm in same way merge sort works because merge sort is a divide and conquer algorithm that is in $\Theta(nlogn)$.

C. We have already proved recursive functions like merge sort so all we need to prove is that we solve the most basic sub problem. We do because comparing two towers x and y will always gives us a marked solution. Next we use induction and to prove that our algorithm solves the non base cases. It does because it all towers that haven't marked towers will check the towers in the directions they haven't checked yet.

D. The run time is $\Theta(nlogn)$.

E. The run time is $\Theta(nlogn)$ because the reccurance is $T(n) = 2T(\frac{n}{2})+n$ which we know solves to $\Theta(nlogn)$.

b) Show that at after the first week, at least $\frac{n-1}{2}$ towers are demolished.

We will use induction to prove this. Our base cases will be of all the possible orderings of 1, 2, and 3 towers, which all are at least (n-1)/2 because for $n = 1$, $\frac{1-1}{0} = 0$ and we destroy 0 towers, for $n = 2$, $\frac{2-1}{2} = \frac{1}{2}$ and we always destroy 1 tower, and then for n = 3, $\frac{3-1}{2} = 1$ and we always destroy 1 or 2 towers.
Now that we have our base cases, we know that any other number of towers is a combination of 2 or 3 towers which all work, so if we have 5 towers, we need to add 1 set of 3 towers to 1 set of 2 towers, each of which destroy 2 and 1 towers respectively. So $1 + 1 = 2$ and $\frac{5-1}{2} = 2$. Since our induction hypothesis is correct, this holds true for any quantity of towers!!!!!

c) (extra credit) How many weeks will the process take? Give an algorithm that determines this in linear time.

The worst case is logn, the best case is 1 week.
A. Our algorithm just does a for loop that runs n times that compares the tower at index i with the tower at index i+1. How many towers we have left is how many weeks it should take.

B.Just a guess.

D. The run time is O(n)

E. it is O(n) because its a for loop that compares two towers n times.

Attempt to make all of your algorithms efficient, and provide all elements listed in the instructions.