CS180-Spring20 Homework3

Gurbir Singh Arora

TOTAL POINTS

100 / 100

QUESTION 1

1 MST 25 / 25

✓ **- 0 pts** Correct

- 2 pts minor mistake

- 10 pts major mistake

- 25 pts incorrect algorithm

- 2 pts minor mistake 2

QUESTION 2

2 Complexcity 21 / 21

✓ **- 0 pts** Correct

- 7 pts Analysis of A incorrect; T(n) = 9T(n/3) + Theta(n^2), T(n) = theta(n^2logn)

- 7 pts Analysis of B incorrect; T(n) = 2T(n - 1) + Theta(1), T(n) = Theta(2^n)

- 7 pts Analysis of A incorrect; T(n) = 5T(n / 2) + Theta(n), T(n) = theta(n^(log(2)5))

QUESTION 3

3 Local Minimum 25 / 25

✓ **- 0 pts** Correct

- 10 pts not optimal

- 2 pts minor mistake

- 5 pts major mistake

- 2 pts unclear handwritting or scanning

- 10 pts insufficient explanation

QUESTION 4

4 Overlap 29 / 29

✓ **+ 5 pts** Sort the intervals

✓ **+ 5 pts** Split all intervals to two halves and recursively find the largest overlap on the left half, and on the right half

✓ **+ 5 pts** Three cases in the merge part

✓ **+ 7 pts** Find the interval who is on the left side and whose right endpoint is the highest, then check its overlap with all of the intervals on the right half

✓ **+ 7 pts** Present psuedocode

+ 0 pts Nothing written

Gurbir Arora
105178554

CS 180 HW #3

1) Given an undirected weighted graph G w/ n nodes and m edges, and we have used Prim algorithm to construct a minimum spanning tree T. Suppose the weight of one of the tree edge ($(u,v) \in T$) is changed from w to w', design an algorithm to verify whether T is still a minimum spanning tree. Runtime: O(m)

Proof of Correctness/Time Complexity: Since the minimum weight edge across any cut is included in the minimum spanning tree (MST), given the edge (u,v) of which w changes to w', let (u,v) be across cut V_1, V_2. If $w' \leq w$, then the edge must continue to be the minimum weighted edge across cut V_1, V_2. Thus, the MST still includes the given edge & the MST doesn't change. If $w' > w$, then you create an array of size n to store $v_i \in V_1$ or $v_i \in V_2$ where i is the index representing position. Remove edge (u,v) from T and start a DFS from u. For all vertices discovered during the DFS, mark them as part of V_1 using the array of size n previous created. Once the DFS is completed, mark all unmarked nodes as part of V_2. The runtime of DFS over a tree takes O(n) and all other steps can be done in linear time. Next, for all edges, check if the edge is across V_1, V_2. If endpoints of the edge are (u', v'), then if $u' \in V_1, v' \in V_2$ or $u' \in V_2, v' \in V_1$, then this edge crosses the cut. The runtime for this is O(m) because

the check takes O(1) time for m edges, O(m). Now, if the edge crosses the cut, check if $w(u',v') < w'$. If true, then (u,v) is no longer the minimum weighted edge across the cut, meaning that T is no longer the MST. If all of the edges which cross V_1, V_2 pass the condition $w(u',v') \geq w'$, then T continues to be the MST as the given edge continues to be the minimum weighted edge across the cut. The total runtime is $O(m+n)$ which is equal to $O(m)$ as $m \geq n-1$, which must be the case for a MST to exist.

Time Complexity

Algorithm: Given edge (u,v) of weight change $w \to w'$ Pseudo code

```
function isMST:
    if (w'<w):
        return true
    else:
        remove edge (u,v)
        arr[n] = {0}
        DFS(u)
        if(node discovered in DFS)
            arr[i] = 1 // marking as visited
        for(all edges){
            if endpoints of edge are(u',v'){
                if((u' ∈ V1 && v' ∈ V2 ||
                u' ∈ V2 && v' ∈ V1){
                    if(w(u',v') < w'){
                        return false}}}}
        return true
```

1 MST **25 / 25**

✓ **- 0 pts** **Correct**

- 2 pts minor mistake

- 10 pts major mistake

- 25 pts incorrect algorithm

- 2 pts minor mistake 2

2) Give the time complexity for the following divide and conquer algorithms

Alg A solves the problem by dividing it into 9 subproblems of 1/3 the size, recursively solves each problem, and then combines the solutions in quadratic time.

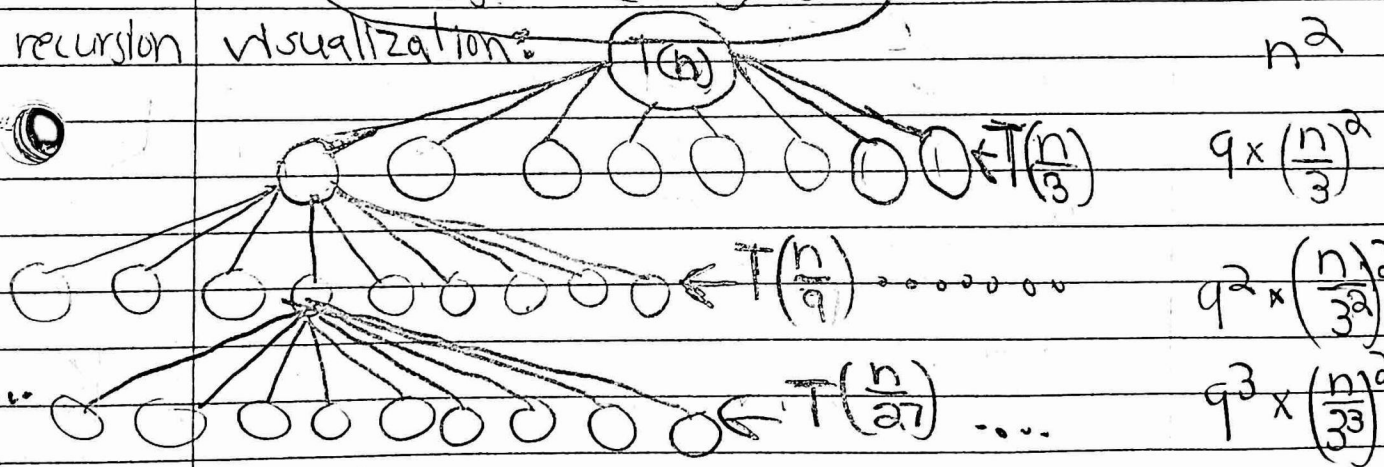
Using master's theorem:

$a=9, b=3, f(n)=n^2, k=\log_3 9 \Rightarrow n^{\log_3 9}=n^2$

so, $f(n)=O(n^{\log_b a})$, thus case 2 so,

$$T(n)=O(n^2 \log n)$$

recursion visualization:

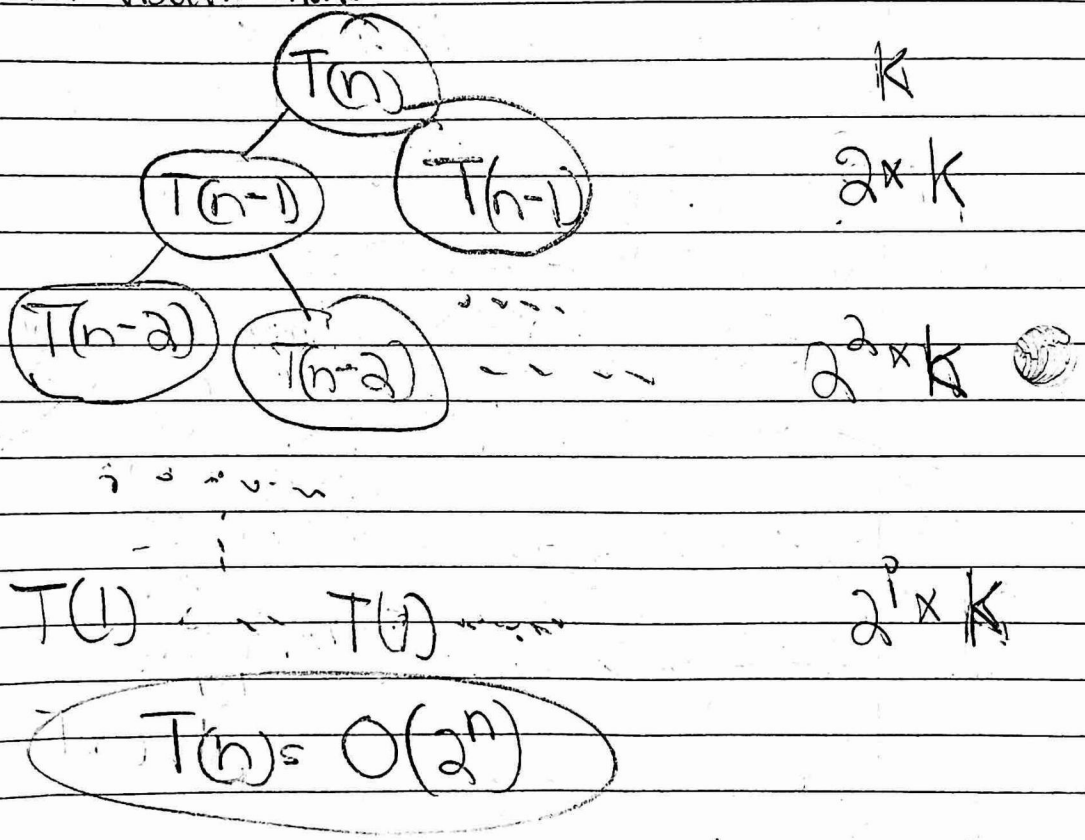


$$T(n)=\sum_{i=0}^{\log_3 n} n^2 = n^2 \log_3 n = O(n^2 \log n)$$

Algorithm B: Solves the problem of size n by recursively solving 2 subproblems of size n-1 & then combine the solution in constant time

$T(n) = 2T(n-1) + O(1)$ where the number of subproblems doubles n times where each subproblem uses $O(1)$ time, thus $T(n) = O(2^n)$

Recursion visualization:

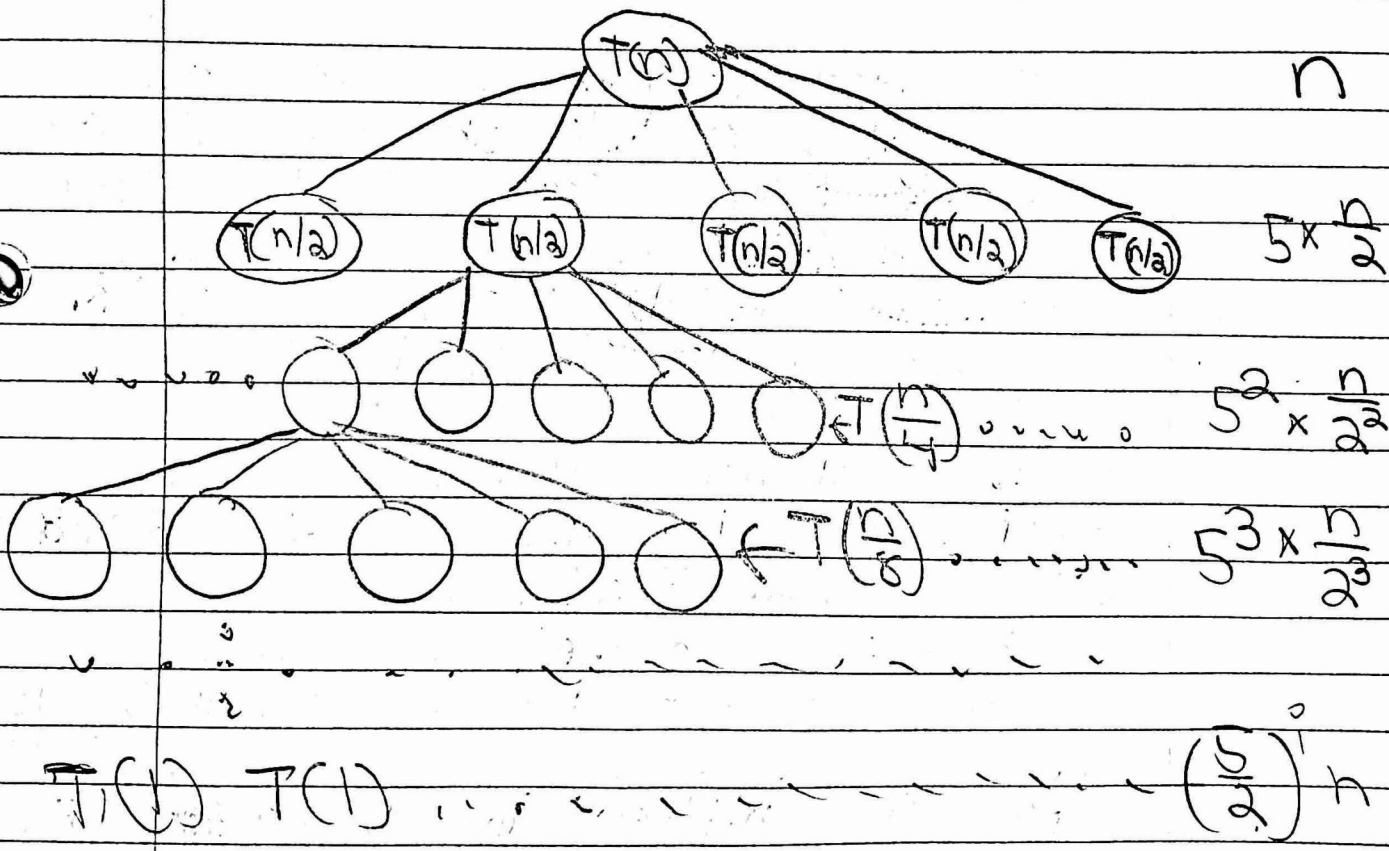


Master's theorem doesn't fit for this algorithm

Algorithm C Solves the problem by dividing it into 5 subproblems of half the size, recursively solves each subproblems, and then combines the solution in linear time

Using Masters theorem: $a=5, b=2$ $f(n)=n$,
$K = \log_2 5 \Rightarrow n^{\log_2 5}$ (case 1 of Masters Theorem
thus $T(n) = O(n^{\log_2 5})$

Recursive Visualization



$T(b) = O(n^{\log_2 5})$

2 Complexcity 21 / 21

✓ **- 0 pts** **Correct**

- 7 pts Analysis of A incorrect; T(n) = 9T(n/3) + Theta(n^2), T(n) = theta(n^2logn)

- 7 pts Analysis of B incorrect; T(n) = 2T(n - 1) + Theta(1), T(n) = Theta(2^n)

- 7 pts Analysis of A incorrect; T(n) = 5T(n / 2) + Theta(n), T(n) = theta(n^(log(2)5))

3). An n-node complete binary tree T where $n = 2^d - 1$ for d. Each node v of T is labeled w/ a x_v. Node v of T is a local minimum if label x_v is less than x_w for all nodes w joined to v by an edge. You are given such a complete binary tree T, but the labeling is only specified in the following way: for each node v, you can determine x_v by probing node v. Show how to find a local minimum of T using $O(\log n)$ probes to the nodes of T.

Basic Idea & proof of correctness:
Since the tree is finite, we know that there's a least element in the set of labels x. Since Labels are distinct, we can require the local minima to be less than its neighbors. This proves there's atleast 1 minima.

Case 1: If T has a root node, then Since T is a complete binary tree, each node contains at most 3 neighbors which are its parent node and the two children nodes. This means that by definition, a node is a local minimum if its label is less than the labels of its two children and the parent. (For root node, label less than 2 children). Thus we can determine if a vertex is a local minimum or not with at most four probes and if T is traversed in an ordered manner, it will take at most 3 probes.

To find the minimum in the rooted case; we must traverse the tree, starting w/ the root as the current node by doing:

1. Probe current node's label, and the label of the 2 children.
2. If current label is the smallest, break and return the current node as a local minimum.
3. Else set current node to the child w/ the smallest label and repeat steps starting at step 1.

This works as we start from the root, so we don't need to worry about the parent's label in the first iteration (bc root has no parent) and in the next iterations, when nodes have a parent, the parent has already been considered/viewed in the previous iteration.

Time Complexity of Case 1: As we choose one node of the two children, the traversal picks a path from the root to a leaf, at most. This means we have a binary tree of depth $d < \log n + 1$ because we have a complete binary tree with $n = 2^d - 1$ nodes, meaning we perform at most $3d \in O(\log n)$ nodes. So, a time complexity of $O(\log n)$.

Case 2: Unrooted Complete Binary Tree Algorithm

1. Pick any node to be the initial current node
2. Probe the label of the current node and all of its neighbors
3. If current node has the min label, break & return it as a local minimum

Prob 3 (cnt)

4. Else, select the neighbor w/ the lowest label as the new current node and return to Step 2. (recursive)

Case 2: Proof of Correctness & Time Complexity:

Through each iteration, we only perform at most 4 probes and using the fact that we cannot backtrack (bc we know the previous had a bigger label), we must following a simple path. Since this is a complete binary tree, the max length of the simple path is $2d$, where $d \leq \log n - 1$. So $4(2)(d) \in O(\log n)$ probes, so the time complexity is $O(\log n)$.

Since the algorithm can never backtrack & graph is finite, we are guaranteed that it will terminate. Now, we only need to prove that vertex v will terminate as a local min. By proof of contradiction, assume v doesn't terminate as a local min, then it must have some neighboring node x that has a label with a lower label value. This leaves us with 3 cases

1. x is the prev. node in the traversal path, which is a contradiction as if $x_v > x_x$, then it wouldn't have chosen v as the next node.
2. x appears in the path, but more than one iteration earlier, but this would mean the graph is not a tree.
3. x is not in the path of traversal, but the algorithm would select x in Step 4 if this was the case. ☒

3 Local Minimum 25 / 25

✓ **- 0 pts** **Correct**

- 10 pts not optimal

- 2 pts minor mistake

- 5 pts major mistake

- 2 pts unclear handwritting or scanning

- 10 pts insufficient explanation

4) Given a list of intervals $[s_i, f_i]$ for $i = 1, \ldots, n$, design a divide and conquer algorithm that returns the length of the largest overlap among all pairs of intervals. $O(n \log n)$

Basic Idea: If the size of the list n, equals 1 return 0. Next, sort the intervals by the value of s. Find the midpoint of the interval (n/2) and create two sets (empty), with one being the left & the other the right. Recursively call the function on the left side and the right side and call a helper function that takes in the left & right side. This helper function finds the greatest overlap of two intervals such that they come from different sets sorted by starting point. Finally, return the max returned value of the 3 function calls described above.

Proof of Correctness: This algorithm works because the stopping condition of n==1 return 0 stops the recursion. Since the list is sorted by s, it's guaranteed that the largest overlap will be found as the pairs with the same starting time s_i will be removed to only have the largest interval and creating a left side & right side will recursively result in each interval combo being checked with the maximum one being returned.

Time complexity: Sorting the input list takes $O(n \log n)$ with the recursive calls taking $T(\frac{n}{2})$ and the helper function taking $O(n)$. Using masters theorem, we get $O(n \log n) + O(n \log n)$, which is still $O(n \log n)$.

list notation: $[[s_0, f_0], [s_1, f_1], \ldots [s_i, f_i]]$

Pseudocode:

Runtime:

function helper($[[a_1,b_1], \ldots [a_i,b_i]], [[c_1,d_1], \ldots [c_j,d_j]]$):
* where $a_1 \le a_2 \le \ldots a_i \le c_1 \le c_2 \ldots c_j$)
  if $i=0$ or $j=0$
    then return 0

```
min = c1
max = 0
x = 0
for i from 1 to i
    if max < bi
        max = bi
for j from 1 to j
    if x < overlap([min, max], [cj, dj])
        x = overlap([min, max], [cj, dj])
return x
```

$O(\frac{n}{2})$ $O(\frac{n}{2})$

total: $O(n)$

function overlap(sort($[[a_1,b_1], \ldots [a_n,b_n]]$))
  if n==1
    then return 0
  mid = $\lfloor n/2 \rfloor$

  LS = $[[a_1,b_1], \ldots [a_{mid}, b_{mid}]]$
  RS = $[[a_{mid+1}, b_{mid+1}], \ldots [a_n, b_n]]$
  var1 = overlap(LS)
  var2 = overlap(RS)
  var3 = helper(LS, RS)
  return max(var1, var2, var3)

$O(n \log n)$

$T(\frac{n}{2})$
$T(n/2)$
$O(n)$

total:

$O(n \log n) + O(n \log n)$
$= O(n \log n)$

4 Overlap 29 / 29

✓ + 5 pts Sort the intervals

✓ + 5 pts Split all intervals to two halves and recursively find the largest overlap on the left half, and on the right half

✓ + 5 pts Three cases in the merge part

✓ + 7 pts Find the interval who is on the left side and whose right endpoint is the highest, then check its overlap with all of the intervals on the right half

✓ + 7 pts Present psuedocode

+ 0 pts Nothing written