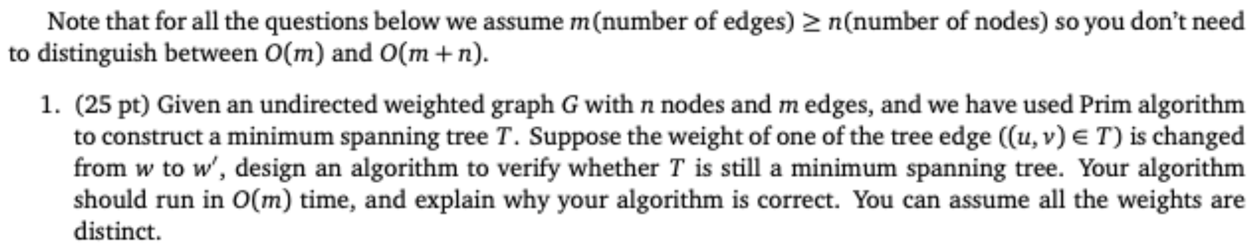


Note that for all the questions below we assume m(number of edges) $\geq n$(number of nodes) so you don't need to distinguish between $O(m)$ and $O(m+n)$.

1. (25 pt) Given an undirected weighted graph G with 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. Your algorithm should run in $O(m)$ time, and explain why your algorithm is correct. You can assume all the weights are distinct.

Algorithm:

Given a minimum spanning tree (MST) T, assume all the weights are distinct. First, Compare the value of weight w and w', there would be two cases as below:

- $w > w'$: (The weight is decreased): T is still a MST, returns true.
- $w < w'$: (The weight is increased): Remove the modified edge (u, v) from T, then T is disconnected due to the property of MST and we obtain two subtrees T_u and T_v. Use BFS from u and from v to determine which vertices belongs to T_u and which abelongs to T_v. Then examine each edge in graph G which has one endpoint belongs to T_u and another belongs to T_v(i.e., any possible edges can connect T_v and T_u). If there is any edge has weight less than w' , then T is not MST after modification, returns false. Otherwise, T is still MST, returns true.

Time Complexity:

Since the algorithm use BFS, it takes $O(m+n)$ time, and the iteration about edges takes $O(m)$ time. Since the hw assumes m ≥ n. The total time is $O(m)$.

Correctness Proof:

Base case: When m = 1, there is only one edge with weight w in T. By the algorithm, since there is no other edges, no matter what weight that w' would be, T is MST. Clearly the algorithm returns the correct answer.

IH: Assume m > 1, the algorithm returns correct answer for graph G with m-1 connected edges.

IS: Add one more edge, we need to prove the IH works for graph G with m connected edges. As the algorithm explains, if the weight w decrease, T is still a MST since the total edge weight of tree T also decrease which is still minimum.

If the weight w increase, the algorithm would check the edges which is outside tree T and comes out from the edge's endpoint node u. If the edge is still the edge with smallest weight connects node u, then it would still be used to construct the MST, which means that T is still MST. If there is other edges that smaller than weight w', that edge would be used to construct new MST, so T is not MST.

2. (21 pt) Give the time complexity for the following divide-and-conquer algorithms:

- Algorithm A solves the problem by dividing it into 9 subproblems of one third the size, recursively solves each subproblem, and then combines the solutions in quadratic time.
- Algorithm B solves the problem of size n by recursively solving two subproblems of size $n-1$, and then combine the solutions in constant time.
- Algorithm C solves the problem by dividing it into 5 subproblems of half the size, recursively solves each subproblem, and then combines the solutions in linear time.

The Master theorem applies of divide and conquer:

If $T(n) = aT(n/b) + cn^k$,

Case 1: If $a/b^k > 1$, $T(n) = \Theta(n^{\log_b a})$

Case 2: If $a/b^k = 1$, $T(n) = \Theta(n^k \log n)$

Case 3: If $a/b^k < 1$, $T(n) = \Theta(n^k)$

- **Algorithm A**: $T(n) = 9T(n/3) + cn^2$

  a = 9, b = 3, k=2. => $a/b^k = 1$

  $n^{log_b a} = n^{log_3 9} = n^2$

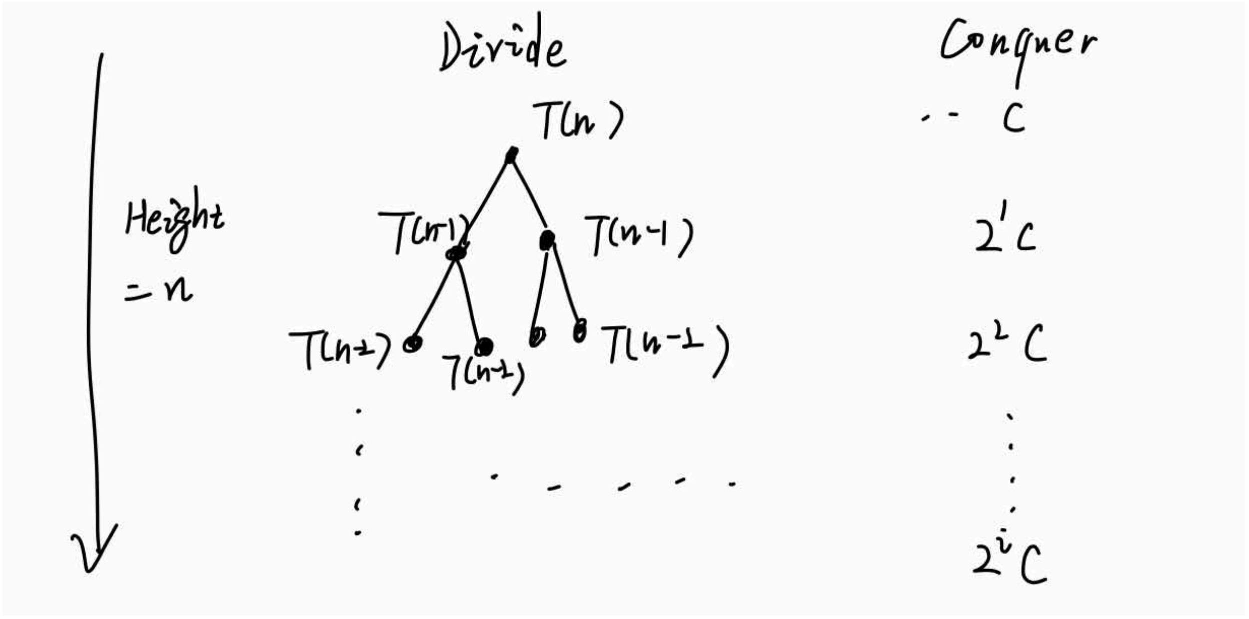
  So $f(n) = \Theta(n^{log_b a}) = \Theta(n^2)$

  Then apply case 2 of the master theorem. Therefore, $T(n) = \Theta(n^2 \log n)$

- **Algorithm B**: $T(n) = 2T(n-1) + c$

  The master theorem cannot be applied for this recurrence because the size of the subproblems is not a fraction of the size of the input problem, so that is no constant k. The number of subproblems doubles n times and each subproblem use $O(1)$ time.

  Using the recursion tree method:



$T(n) = \sum_{i=0}^{n-1} 2^i c = c\frac{2^n-1}{2-1} = (2^n - 1)c$

=> $T(n) = \Theta(2^n - 1) = \Theta(2^n)$

Hence, $T(n) = \Theta(2^n)$

- **Algorithm C**: $T(n) = 5T(n/2) + cn$

  a = 5, b = 2, k =1 . => $a/b^k = 2.5 > 1$

  $n^{log_b a} = n^{log_2 5} > n^2$

  So $f(n) = O(n^{log_b a - \epsilon})$, where $\epsilon = 1 > 0$, then case 1 of master theorem could be applied.

  Therefore, $T(n) = \Theta(n^{log_2 5})$

3. (25pt) Consider an n-node complete binary tree T, where $n = 2^d - 1$ for some d. Each node v of T is labaled with a real number x_v. You may assume that the real numbers labeling the nodes are all distinct. A node v of T is a local minimum if the label x_v is less than the label x_w for all nodes w that are joined to v by an edge.

   You are given such a complete binary tree T, but the labeling is only specified in the following implicit way: for each node v, you can determine the value x_v by probing the node v. Show how to find a local minimum of T using only $O(\log n)$ probes to the nodes of T. Explain why your algorithm is correct.

Algorithm:

First, start at the root of the binary tree T, check if the roor is smaller than its both two children.

- If yes, the root is local minimum, return the root node.
- If no, move to any smaller child node and iterate.

The algorithm will be ternimated when two cases:

- it visit a leaf node w -> return the node w
- visit a node v which is smaller than both its children. -> return the node v.

Time Complexity : The algorithm use $O(d) = O(\log n)$ probes of the tree

Correctness Proof (by induction):

Base Case: Assume there is only one node in tree T. As the algorithm explained above, since there is no children for this root node, the root is local minimum. Clearly the algorithm returns the correct answer.

IH: Assume n > 1, the algorithm returns the local minimum for tree T with n-1 nodes .

IS: Add one more node in T. Now we need to prove the IH works for T with n nodes. As the algorithm explained, there would be two cases when the algorithm terminate:

- Terminate at leaf node w:

  as the algorithm explains,the nodes would be iterated in descending order by their value. Hencel, node w is a local minimum because node w is smaller than its parent and it does not have children.

- Terminate at a node v which is smaller than both its children:

  Similarly, since node v was also chosen from previous iteration, it is smaller than its parent. In other hands, as the algorithm explained, the algorithm terminates because node v is smaller thatn its both two children. That is, node v is local minimum because node v is smaller than its parent and its two children.

  So the algorithm still returns local minimum in this case. Proved.

4. (29pt) Given a list of intervals $[s_i, f_i]$ for $i = 1, \ldots, n$, please design a divide-and-conquer algorithm that returns the length of the largest overlap among all pairs of intervals. The algorithm should run in $O(n \log n)$ time. For example, if $n = 3$, the intervals are $\{[1,6],[2,7],[6,8]\}$, then the length of the largest overlap is 4. (Hint: sort the intervals by s_i values, and divide them into two parts based on their s_i values).

First sort the intervals by s_i values, and divide them into two parts based on their s_i values, left side and right side, then conduct recursion function on them as below.

Pseudocode:

```
Sort[s_1,f_1], … , [s_n, f_n] by s_i vaules.

get_overlap(sorted list of intervals):
  if n==1
    return 0.
  Endif
  let mid = n/2
  /* divide into two parts */
  Left = [[s_1, f_1], ..., [s_mid, b_mid]]
  Right = [[s_mid+1, f_mid+1], ..., [s_n, f_n]]
  overlap1 = get_overlap(Left)
  overlap2 = get_overlap(Right)
  overlap3 = overlap_between_two_lists(Left,Right)
  largest_overlap = max(overlap1,overlap2,overlap3)
  return largest_overlap

/* function to get the largest overlap between two intervals in different list */
/* Assume x_1 ≤ ... ≤ x_w ≤ a_1 ≤ ... ≤ a_k */
overlap_between_two_lists([[x_1, y_1],...,[x_w, y_w]], [[a_1, b_1], ..., [a_k, b_k]]):
  if (w == 0 || k == 0)
    return 0
  Endif
  let min_a = a_1
  let max_y = max_overlap = 0
  for i in range(1, w):        /* get the most right ending point for left side */
    if max_y < y_i:
      max_y = y_i
    Endif
  Endforloop
  for j in range(1, k):
    if max_overlap < overlap([min_a, max_y], [a_k. b_k])
      max_overlap = overlap([min_a, max_y], [a_k. b_k])
    Endif
  Endforloop
  return max_overlap

/* function to get overlap between two intervals */
```

```
38  /* input: two intervals */
39  overlap([a, b], [c, d]):   /* assume a ≤ c */
40      if c > b:
41        return 0
42      Else:       /* c ≤ b */
43        if b ≤ d
44        return b-c
45        if b > d:
46        return d-c
47        Endif
48      Endif
49
```

Time Complexity:

Sorting the intervals take $O(n\log n)$ time. And function `overlap_between_two_lists` takes time O(n). Let T(n) denote the tine to run fuction `get_overlap` on the list of intervals with length n. Line 11&12 recursively call the function `get_overlap` on left half and right half takes 2T(n/2). Then we get the recurrence relation $T(n)=2T(n/2)+O(n)$, applying the master theorem, this solves to $O(n\log n)$. Hence, the totol time should be $O(n\log n)+O(n\log n)=O(n\log n)$.

Correctness Proof (by induction):

Base Case: When n = 1, there is only one interval. So the largest overlap is 0. Clearly the algorithm (line 2 of the pasudocode) handles this case, it it returns the correct answer.

IH: Assume n > 1, the algorithm works correctly on the list of intervals with size n-1.

IS: Add one more intervals into the list. Now we need to prove the IH also works correctly on the list of intervals with size n. After dividing the intervals into two parts, there would be three cases where the largest overlap comes from:

- Comes from two intervals in the left half.
- Comes from two intervals in the right half.
- Comes from one intervals in the left half and another in the right half.

Clearly as the algorithm explains, it finds the largest overlap for these three cases, and compare them, output the maximum of them, which is the largest overlap for the list of intervals. Hence the algorithm returns correct answer for this case.

Proved.