CSE 101 FALL 2019 Homework 5 Due: Thursday, 11/7 at 11:59pm

1. Suppose that you are given a connected undirected graph G with positive edge weights and a subset of the vertices $B \subseteq V$. You wish to design an algorithm that returns a spanning tree of G with the property that all vertices in B are leaves that has the minimum weight. If no such spanning tree exists, your algorithm should return "not possible".

   (5 points for reasonably efficient correct algorithm (no correctness proof necessary), 5 points for correct time analysis.)

   Algorithm Description: If your graph G has two or fewer vertices, then the solution is just the graph itself.
   Otherwise, create G' from G by removing all vertices from B. Run explore on G' to see if it is connected. If it is not then return "not possible". Run Kruskal's on G' to get the minimum spanning tree of G'. Then, for each vertex $v \in B$, add the lightest edge (v, u) such that $u \notin B$. If no such edge exists for a particular vertex then return "not possible."
   Time analysis: Creating G' takes $O(|E|)$ time (the input is a connected graph.) Then Kruskal's on G' takes at most $O(|E| \log |V|)$. Then finding the lightest connecting edge for each vertex in B takes at most $O(|E|)$ time so all in all, the total runtime is $O(|E| \log |V|)$.

2. Consider the following divide and conquer algorithm that claims to return a set of edges that define an MST of a connected undirected graph G with positive edge weights $w(e)$.

   Algorithm Description: Given an undirected connected graph $G = (V, E)$ where $V = [v_1, \ldots, v_n]$,

   - If $n = 1$ then return the empty set of edges.
   - Otherwise, split the set of vertices into two sets: $V' = [v_1, \ldots, v_{n/2}]$ and $V'' = [v_{n/2} + 1, \ldots, v_n]$.
   - Create two new graphs $G' = (V', E')$ and $G'' = (V'', E'')$ where $E' \subseteq E$ is the set of edges with both endpoints in V' and $E'' \subseteq E$ is the set of edges with both endpoints in V''.
   - Recursively run the algorithm on G' and G'' to get M' and M'', respectively.
   - Find the lightest edge e among the remaining edges E''' that connect G' to G'' ($E''' = E - (E' \cup E'')$).
   - Return $M' \cup M'' \cup \{e\}$.

   (a) Calculate the runtime of this algorithm in terms of $n = |V|$ and $m = |E|$.

   Solution: We can show that in the worst case, this algorithm will run in $O(n^2)$ time and if $m = O(n)$, we can show it runs in $O(m \log m)$ time.
   If G is a complete graph, then let $N = n + m = (n+1)n/2$, then each recursive call is on an input of roughly $N/4$ and the non-recursive part takes $O(N)$ time. So the recursion is $T(N) = 2T(N/4) + O(N)$ so by the master theorem, $a < b^d$ so $T(N) = O(N)$. But $N = O(n^2)$ so for a complete graph, we get $O(n^2)$.
   If G is just a line of points, $N = n + m = 2n - 1$ and we were (un)lucky enough that it always split the line right in half, then each recursive call would be of size $N/2$ so the recursion is $T(N) = 2T(N/2) + O(N)$ so by the master theorem, $a = b^d$ so $T(N) = O(N \log N)$. But $N = O(n)$ so for this graph, we get $O(n \log n) = O(m \log m)$.

   (b) Prove or disprove the correctness of this algorithm. (7 points)

   Solution: Incorrect. Counter-example:
   $A : (B, 2), (C, 1)$
   $B : (A, 2), (D, 1)$
   $C : (A, 1), (D, 2)$
   $D : (B, 1), (C, 2)$

then if you split the vertices into the subsets $\{A, B\}$ and $\{C, D\}$ then you will create the graphs with A and B connected with a 2 edge and C and D connected with a 2 edge then connect the two subgraphs with a 1 edge, resulting in a ST of weight 5. The MST has weight 4.

3. Suppose you have a $n \times n$ 2-dimensional array A filled with distinct integers such that $A[i, j] \leq A[i+1, j]$, $A[i, j] \leq A[i, j+1]$. In other words the numbers are increasing in each dimension (the top left corner is the least number and the numbers increase to the right and increase down.) You also have a target integer x. You wish to know if x appears somewhere in A.

   Consider the following algorithm that will return TRUE if x is in A and FALSE otherwise:

   - If A is only a 1×1 array, then check to see if x is equal to $A[0, 0]$. If so, return TRUE. Else return FALSE.
   - Otherwise, compare x with $A[n/2, n/2]$.
   - if $x == A[n/2, n/2]$ then return TRUE
   - if $x < A[n/2, n/2]$ then we know that x is not in the lower right quarter of A. So recurse on the 3 other quarters. If you find x in any one of them return TRUE, otherwise return FALSE
   - if $x > A[n/2, n/2]$ then we know that x is not in the upper left quarter of A. So recurse on the 3 other quarters. If you find x in any one of them return TRUE, otherwise return FALSE

   (a) Calculate the runtime of this algorithm in terms of n. (3 points)

   Solution: If we let $m = n^2$ then the recursion is: $T(m) = 3T(m/4) + O(1)$. So by the master theorem with $a = 3, b = 4, d = 0$ and $a > b^d$, we have $T(m) = O(m^{\log_4(3)}) = O(m^{0.8}) = O(n^{1.6})$

   (b) How could you change your algorithm to work on a 3-dimensional $n \times n \times n$ array of integers that are increasing in all three dimensions? How would your runtime change?

   Solution: You could compare x to the middle array value and based on the result, eliminate one of the subcubes of size $n^3/8$. Then recurse on the remaining 7 cubes. Assuming that $m = n^3$, the recursion is: $T(m) = 7T(m/8) + O(1)$. So by the master theorem with $a = 7, b = 8, d = 0$ and $a > b^d$, we have $T(m) = O(m^{\log_8(7)}) = O(m^{0.94}) = O(n^{2.81})$

4. Suppose that you are the news director of a local news station. On a particular day, in the morning, you get a schedule of n events each with a start and end time $E = [(s_1, f_1, e), \ldots, (s_n, f_n, e)]$ and the schedule of n news reporters $R = [(s'_1, f'_1, r), \ldots, (s'_n, f'_n, r)]$ each with a start and end time that they are available. You only have budget for one news reporter, so you would like to choose the (reporter, event) pair that has the maximum schedule overlap. Design an algorithm that returns just the duration of the maximum schedule overlap in $O(n \log n)$ time. If none of the reporters are available during any of the events, return 0.

   For example: Let's say the event schedules are: $[(1, 7, e), (4, 7, e), (2, 5, e), (6, 8, e)]$ and the reporter schedules are $[(2, 4, r), (6, 11, r), (3, 6, r), (5, 9, r)]$ then the reporter, event pair with the largest overlap is the $(3, 6, r)$ reporter with the $(1, 7, e)$ event and there overlap is the interval $(3, 6)$ which has a length of 3.

   (5 points for $O(n \log n)$ correct algorithm (no correctness proof necessary), 5 points for correct time analysis.)

   The following algorithm will take as input k events and ℓ reporters with k and ℓ not necessarily equal.

   High-level description: On the input: $E = [(s_1, f_1, e), \ldots, (s_k, f_k, e)], R = [(s'_1, f'_1, r), \ldots, (s'_\ell, f'_n \ell, r)]$
   First sort the whole list together before inputting it into the algorithm, the third argument will indicate which interval is an event and which is a reporter.
   Start of recursive algorithm on sorted list:

- Count the number of reporters and events. If there are no events or no reporters then return 0.
- Split the list into L, the left half and R, the right half.
- Recurse on L and R to get OL and OR, the duration of the max overlap pair from L and R, respectively.
- Then, in order to find the maximum overlap for a reporter in L paired with an event in R, it is sufficient just to compare the reporter in L with the latest end time with all events of R. Call the maximum duration of such a pair A_1.
- Then, in order to find the maximum overlap for an event in L paired with a reporter in R, it is sufficient just to compare the event in L with the latest end time with all reporters in R. Call the maximum duration of such a pair A_2.
- Then return the maximum of OL, OR, A_1, A_2.

Runtime Analysis: Sorting the original list of size $2n$ will take $O(n \log n)$. Then let's calculate the runtime of the algorithm based on $k + \ell = m$. Then counting the number of reporters and events takes $O(m)$ time. Each recursive call will take $T(m/2)$ time. Finding the reporter (event) with latest end time in L will take $O(m)$ time. Then comparing it with all events (reporters) to find the maximum will take $O(m)$ time. Therefore the non-recursive part takes $O(m)$ time so the runtime of the algorithm will follow the recursion: $T(m) = 2T(m/2) + O(m)$ which simplifies to $T(m) = O(m \log m)$. Then since the original list has $2n$ elements, this algorithm will take $O(n \log n)$ from sorting + $O(n \log n)$ from the algorithm. All in all, it will take $O(n \log n)$.

5. Given a directed, rooted binary tree G with positive vertex weights, root r and a threshhold H. You are given the pointer to the root r. Each vertex v has pointers to $v.lc$ and $v.rc$, the left and right children of v and a vertex weight $w(v)$. The value NIL represents a null pointer, showing that v has not child of that type.

   Design a *linear time* algorithm to find the longest path in G starting from the root r such that the sum of the vertex weights along the path is less than or equal to H. (Your algorithm can just return the length of the path and you can assume that the tree is full and all levels are filled.)

   (5 points for *linear time* correct algorithm (no correctness proof necessary), 5 points for correct time analysis.)

   Algorithm Description: For the input root r and the threshhold H, The algorithm will return L which is the number of vertices of the longest path starting from r such that the sum of the vertex weights is less than or equal to H.

   Pseudocode: ThreshholdTree(r, H)**:**

   1. **if** $w(r) > H$:
   2. **return** 0
   3. **if** r is a leaf:
   4. **return** 1
   5. $L =$ ThreshholdTree$(r.lc, H - w(r))$
   6. $R =$ ThreshholdTree$(r.rc, H - w(r))$
   7. **return** $1 + \max\{L, R\}$

   Runtime analysis: Since the tree is full and all levels are complete, each recursive call is about half the size of the original tree. The non-recursive part of the algorithm is constant time. So $T(n) = 2T(n/2) + O(1)$ which simplifies to $T(n) = O(n)$.