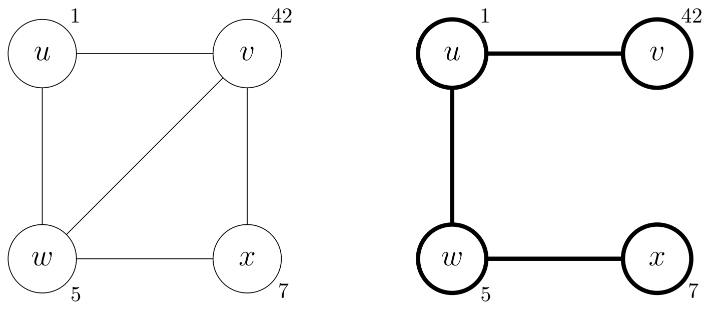
This assignment is **due on May 22** and should be submitted on Gradescope. All submitted work must be *done individually* without consulting someone else's solutions in accordance with the University's "Academic Dishonesty and Plagiarism" policies.

As a first step go to the last page and read the section: Advice on how to do the assignment.

Problem 1. (10 points)
We are given a simple connected undirected graph $G = (V, E)$. Every vertex v has a distinct positive integer p_v associated with it. We define a monotone spanning tree as a spanning tree where every root to leaf path in the tree encounters a non-decreasing sequence of integers. We want to compute such a monotone spanning tree T of graph G, if one exists (return *null* otherwise).

Example:



In the example above, the graph on the left is the input graph. The numbers next to the vertices are their associated integers. The tree on the right, rooted at u, is one of the possible monotone spanning trees of the graph. It contains two root to leaf paths: u, v (1, 42) and u, w, x (1, 5, 7). Both of these paths visit non-decreasing integers and are thus monotone.

We are given two algorithms for this problem:

MODIFIEDPRIM works similar to Prim's MST algorithm: we maintain a set S of vertices that are already part of our tree and in every iteration of the algorithm, we add the vertex (adjacent to a vertex in S) with smallest integer to our tree. Initially, our set S consists of only the vertex s that has the smallest integer associated with it. Of course, before adding a vertex to S, we need to check whether its integer is at least as large as that of its parent in the tree. If this isn't the case, we report that the graph has no monotone spanning tree.

MODIFIEDDFS runs DFS starting from the vertex with the smallest integer associated with it. Every time it processes a vertex, it visits its neighbors that haven't been visited yet and have an integer that's at least as large as its own. If at the end we have visited all vertices of the graph, we return the tree, and *null* otherwise.

```
function MODIFIEDPRIM(G)
    $T \leftarrow \emptyset$
    $\mathcal{H} \leftarrow$ new heap containing only $(p_s, s)$, i.e., the pair of vertex $s$ and its associated positive integer using the integer as the key.
    while $\mathcal{H} \neq \emptyset$ do
        $u \leftarrow \mathcal{H}$.REMOVEMIN()
        Add $u$ to $T$, along with the edge to its parent ($s$ has no parent)
        for $(u, v)$ incident to $u$ do
            if $v \notin T$ and $v$ doesn't have a parent then
                if $p_u > p_v$ then
                    return $null$
                else
                    Set $u$ to be the parent of $v$
                    Insert $(p_v, v)$ into $\mathcal{H}$
    Set $s$ to be the root of $T$
    return $T$
```

```
function MODIFIEDDFS(G)
    Initialize visited as in DFS
    Let $s$ be the vertex with smallest integer.
    $T \leftarrow (\{s\}, \emptyset)$
    Set $s$ to be the root of $T$
    MODIFIEDDFSVISIT($s$)
    if $T$ contains all vertices of $G$ then
        return $T$
    else
        return $null$
```

```
function MODIFIEDDFSVISIT(u)
    Set visited[$u$] to true
    for $(u, v)$ incident to $u$ do
        if not visited[$v$] and $p_u \leq p_v$ then
            Add vertex $v$ and edge $(u, v)$ to $T$
            MODIFIEDDFSVISIT($v$)
```

a) Argue whether MODIFIEDPRIM always returns the correct answer by either arguing its correctness (if you think it's correct) or by providing a counterexample (if you think it's incorrect).

b) Argue whether MODIFIEDDFS always returns the correct answer by either arguing its correctness (if you think it's correct) or by providing a counterexample (if you think it's incorrect).

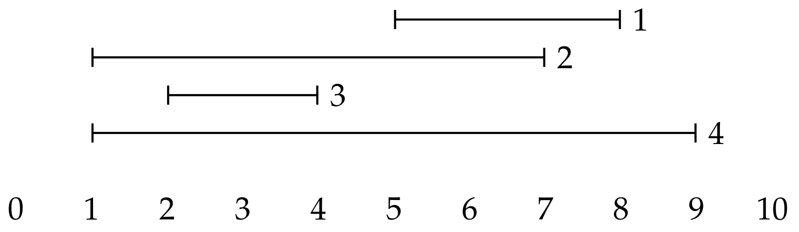
Solution 1.

a) This algorithm is correct, so we need to show that if it returns *null* then no monotone spanning tree exists, and that if it returns a tree, this tree is indeed a monotone spanning tree (i.e., a valid solution). Together these two statements prove the correctness of our algorithm.
Let's start with the first statement. We return *null* only when we find that during our execution we encounter a neighbor without a parent such that its integer p_v is less than our current integer p_u. Since this vertex v doesn't have a parent, we know that no previous vertex in T reached it. Furthermore, since we process the integers in $\mathcal{H}$ in non-decreasing order (every pair added has an integer key at least as large as the key that added it), we know that there are no vertices with integers smaller than p_u in $\mathcal{H}$. Hence, v can only be reached by going through a vertex storing an integer that is at least p_u. Since $p_u > p_v$, this means that v can't be reached using a monotone path from the vertex with the smallest integer, implying that no monotone spanning tree exists for this graph.
To see that when we return a tree, this is indeed a monotone spanning tree, we first observe that we indeed return a tree (since every vertex except for the root has exactly one parent) and that it spans all vertices (G is connected, so if we reach line 14, we have processed all neighbors of all vertices without ever executing line 9). Hence, it only remains to prove that the vertices along every root to leaf path store a non-decreasing sequence of integers. This follows from the way we set our parents: we only set the parent of a vertex v if it isn't part of the tree yet and doesn't have a parent (i.e., only for vertices that we've never seen before) and then only when its integer p_v is at least as large as the integer p_u of the current vertex. This latter check ensures that when we set a parent, the step from the parent to the child is monotone. Since this property holds for every parent-child pair, it follows that every root to leaf path is monotone as well.

b) This algorithm is correct, so we need to show that if it returns *null* then no monotone spanning tree exists, and that if it returns a tree, this tree is indeed a monotone spanning tree (i.e., a valid solution). Together these two statements prove the correctness of our algorithm.
We first observe that any path traversed by the DFS algorithm is non-decreasing, as we explicitly check for this when processing the neighbors of a vertex. Since we only add an edge (u, v) immediately before recursing on vertex v and thus setting visited[v] to true, we will also construct a tree. Hence, after executing line 6 of MODIFIEDDFS, T is a monotone tree. On line 7, we explicitly check if T is spanning and if so, we have a monotone spanning tree and can thus return it safely. If not all vertices are in T, this means that there exists at least one vertex v such that every time we reached a neighbor u of it, we had that $p_u > p_v$. Hence, we couldn't reach v using a monotone path and thus the input graph didn't admit a monotone spanning tree and we correctly return *null*.

Problem 2. (25 points)
You've been hired by Telstra to set up mobile phone towers along a straight road in a rural area. There are n customers. Each customer's mobile device has a specific range in which it can send and receive signals to and from a tower. The goal is to ensure that each customer is within range of a tower while using as few towers as possible. More formally, each customer has an associated interval $[\ell_i, r_i]$, where ℓ_i and r_i are positive integers ($l_i \leq r_i$). The interval of a customer specifies the range of their mobile device. Serving customer i requires us to place a tower at a location x satisfying $\ell_i \leq x \leq r_i$. A solution is a set L of tower locations and is feasible if for each customer i, there exists a location $x \in L$ satisfying $\ell_i \leq x \leq r_i$. We're interested in constructing the *smallest* set L that is feasible.

Example:



Suppose we have 4 customers with intervals $[5,8], [1,7], [2,4], [1,9]$ (see the above figure). In this case, $L = \{2,6\}$ is an optimal solution. $L = \{3\}$ isn't a solution, as we customer 1 doesn't have any tower in their range. $L = \{2,3,6\}$ is a solution, but it isn't the smallest one, as it contains three towers, while having two would have sufficed.

Your task is to design a greedy algorithm that returns the *smallest* set of tower locations that is feasible. For full marks, your algorithm should run in $O(n \log n)$ time. Remember to:

a) Describe your algorithm in plain English.

b) Prove the correctness of your algorithm.

c) Analyze the time complexity of your algorithm.

Solution 2.

a) Our greedy algorithm first sorts the intervals in increasing order of r_i; call this sorted list of intervals S. We also renumber the intervals so that $r_1 \leq r_2 \leq \ldots \leq r_n$. Then, we initialize an empty list L and an integer $x = 0$ representing the location of the last-added tower. Next, we iterate through S: for each interval $[\ell_i, r_i] \in S$, if x does not satisfy $\ell_i \leq x \leq r_i$, add r_i to L and set $x = r_i$. Once we have iterated through all of S, we return L as the solution.

b) First, note that the greedy algorithm iterates through every interval $[\ell_i, r_i]$ in S, and if the last-added tower x does not satisfy $\ell_i \leq x \leq r_i$, adds a tower at r_i. Thus, the solution L returned by the greedy algorithm is feasible.
Next, we prove that L is optimal. Let L^* be an optimal solution. If $L^* = L$, then we are done. Suppose $L^* \neq L$. We now show using an exchange argument that there exists a feasible solution L' with $|L' \cap L| > |L^* \cap L|$ and $|L'| = |L^*|$, i.e. L' is an optimal solution that is closer to L. Let x_k be the k-th location in L (starting from the left) and y_k be the k-th location in L^* (also starting from the left). Let j be the smallest index such that $x_j \neq y_j$. We claim that removing y_j from L^* and adding in x_j results in a solution $L' = L^* \setminus \{y_j\} \cup \{x_j\}$ that is feasible.
By definition of the greedy algorithm, $x_j = r_i$ for some interval $[\ell_i, r_i]$ and locations $x_1, \ldots, x_{j-1}$ serve customers $1, \ldots, i-1$ but not customer i. By definition of j, we have that $x_k = y_k$ for every $k < j$. Thus, in order for L^* to serve customer j, its j-th location must satisfy $\ell_j \leq y_j \leq x_j = r_i$. Since customer $1, \ldots, i$ are served by $x_1, \ldots, x_j$, the new solution L' can only be infeasible if there exists a customer $i' > i$ that is served by y_j but not by x_j. However, this cannot happen as $r_i \leq r_{i+1} \leq \ldots \leq r_n$ and so any customer $i' > i$ that is served by y_j is also served by x_j. Hence, we have that L' is feasible. It is also clear that $|L'| = |L^*|$. Repeating this argument allows us to conclude that L is indeed optimal.

c) The sorting operation takes $O(n \log n)$ time. Initializing L and x takes $O(1)$ time. The subsequent check for whether $\ell_i \leq x \leq r_i$, and adding r_i to L if not, requires $O(1)$ time per customer, or $O(n)$ time in total. The runtime of the algorithm is therefore $O(n \log n)$.

Problem 3. (25 points)
Alice and Bob want to split a log cake between the two of them. The log cake is n centimeters long and they want to make one slice with the left part going to Alice and the right part going to Bob. Both Alice and Bob have different values for different parts of the cake. In particular, if the slice is made at the i-th centimeter of the cake, Alice receives a value $A[i]$ for the first i centimeters of the cake and Bob receives a value $B[i]$ for the remaining $n - i$ centimeters of the cake. Alice and Bob receives strictly higher values for larger cuts of the cake: $A[0] < A[1] < \ldots < A[n]$ and $B[0] > B[1] \ldots > B[n]$. Ideally, they would like to cut the cake fairly, at a location i such that $A[i] = B[i]$, if it exists. Such a location is said to be *envy-free*.

Example:
When $A = [1, 4, 6, 10]$ and $B = [20, 10, 6, 4]$ then 2 is the envy-free location, since $A[2] = B[2] = 6$.

Your task is to design a divide and conquer algorithm that returns an envy-free location if it exists and otherwise, to report that no such location exists. For full marks, your algorithm should run in $O(\log n)$ time. Remember to:

a) Describe your algorithm in plain English.

b) Prove the correctness of your algorithm.

c) Analyze the time complexity of your algorithm.

Solution 3.

a) The main observation here is that if there exists an envy-free location k, then because A is monotonically increasing while B is monotonically decreasing, we have that $A[i] < B[i]$ for all $i < k$ and $A[i] > B[i]$ for all $i > k$. Thus, we can use an algorithm similar to binary search.
We maintain two variables lb and ub. These are initialized to 0 and n, respectively. While lb $<$ ub, we do the following. Consider the index m in the middle of lb and ub. If $A[m] = B[m]$, then we return m; otherwise, if $A[m] < B[m]$, we repeat with lb $= m + 1$, and if $A[m] > B[m]$, we repeat with ub $= m - 1$. If we ever end up with an empty range (i.e., lb $>$ ub), we report that no envy-free location exists.

b) We show that if there exists an envy-free location k, then the invariant that lb $\leq k \leq$ ub is maintained throughout the execution of the algorithm. Initially, ub $= n$ and lb $= 0$ so the invariant holds prior to the start of the while loop. Let us now consider some iteration of the while loop and assume the invariant held at the start of the iteration. So, we have lb $\leq k \leq$ ub. Recall that, as observed above, $A[i] < B[i]$ for all $i < k$ and $A[i] > B[i]$ for all $i > k$. Thus, if $A[m] < B[m]$, then $m + 1 \leq k \leq$ ub, and if $A[m] > B[m]$, then lb $\leq k \leq m - 1$. Thus, since we set lb $= m + 1$ in the first case and ub $= m - 1$ in the second case, the invariant holds at the end of the iteration. The invariant implies that if there is an envy-free location, the algorithm will eventually find it. Hence, this proves the correctness of the algorithm.

c) Initially, ub $-$ lb $= n$ and halves in each iteration until ub $=$ lb. Thus, there are $O(\log n)$ iterations of the while loop. Each iteration consists of a constant number of $O(1)$ time operations, thus all iterations together take $O(\log n)$ time. The initialization takes $O(1)$ time. Thus, overall, the algorithm is $O(\log n)$.
Alternatively, we can use the Master Theorem on the recurrence $T(n) = T(n/2) + O(1)$ (since every time we recurse on an instance of half the size and the amount of time needed per iteration is constant). This falls in Case 2 with $k = 0$, since $\log_2 1 = 0$. Hence the running time is $O(\log n)$.

Advice on how to do the assignment

- Assignments should be typed and submitted as pdf (no pdf containing text as images, no handwriting).
- Start by typing your student ID at the top of the first page of your submission. Do **not** type your name.
- Submit only your answers to the questions. Do **not** copy the questions.
- When asked to give a plain English description, describe your algorithm as you would to a friend over the phone, such that you completely and unambiguously describe your algorithm, including all the important (i.e., non-trivial) details. It often helps to give a very short (1-2 sentence) description of the overall idea, then to describe each step in detail. At the end you can also include pseudocode, but this is optional.
- In particular, when designing an algorithm or data structure, it might help you (and us) if you briefly describe your general idea, and after that you might want to develop and elaborate on details. If we don't see/understand your general idea, we cannot give you marks for it.
- Be careful with giving multiple or alternative answers. If you give multiple answers, then we will give you marks only for "your worst answer", as this indicates how well you understood the question.
- Some of the questions are very easy (with the help of the slides or book). You can use the material presented in the lecture or book without proving it. You do not need to write more than necessary (see comment above).
- When giving answers to questions, always prove/explain/motivate your answers.
- When giving an algorithm as an answer, the algorithm does not have to be given as (pseudo-)code.
- If you do give (pseudo-)code, then you still have to explain your code and your ideas in plain English.
- Unless otherwise stated, we always ask about worst-case analysis, worst case running times, etc.
- As done in the lecture, and as it is typical for an algorithms course, we are interested in the most efficient algorithms and data structures.
- If you use further resources (books, scientific papers, the internet,...) to formulate your answers, then add references to your sources and explain it in your own words. Only citing a source doesn't show your understanding and will thus get you very few (if any) marks. Copying from any source without reference is considered plagiarism.