Problem 1 (10 pts)

Construct an example to show that the following Greedy algorithm for the interval partitioning problem can allocate more than depth many classrooms (so it is not optimum): Sort the lectures based on their **finishing time**. When considering the next job, allocate it to the available classroom with the smallest index. If no classroom is available, allocate a new classroom.
Note: An example for the interval partitioning problem includes starting and ending time for all lectures. You need to show that by applying the greedy algorithm mentioned in the problem over the example will allocate more classrooms than the depth.

Solution

Counterexample: Consider the following four lectures with starting and finishing times:

$$\begin{aligned} L_1 &: \ [1, 4] \\ L_2 &: \ [2, 5] \\ L_3 &: \ [3, 6] \\ L_4 &: \ [1, 6] \end{aligned}$$

Notice that the *depth* (i.e. the maximum number of overlapping lectures) is 3 (for example, at time $t = 3$, lectures L_1, L_2, and L_4 overlap).

Now, suppose we apply the following greedy algorithm: *Sort the lectures by increasing finishing times. When considering a lecture, assign it to the available classroom with the smallest index; if no classroom is available, allocate a new classroom.*

A possible order after sorting by finish times is:

$$L_1\,(f = 4), \quad L_2\,(f = 5), \quad L_3\,(f = 6), \quad L_4\,(f = 6).$$

The algorithm proceeds as follows:

1. Assign L_1 to Room 1.
2. L_2 starts at 2, but Room 1 is occupied until 4; assign L_2 to Room 2.
3. L_3 starts at 3. Room 1's last lecture (L_1) finishes at 4 and Room 2's last lecture (L_2) finishes at 5; hence, neither is available. Allocate Room 3 for L_3.
4. L_4 starts at 1. Checking all rooms, we see that in each room the last scheduled lecture finishes after time 1. Therefore, a new Room 4 is allocated for L_4.

Therefore, the algorithm uses 4 classrooms even though the depth is only 3. This shows the algorithm is not optimal.

Problem 2 (10 pts)

Suppose you are given a connected graph G, with edge costs that are all distinct. Prove that G has a unique minimum spanning tree.

Solution

Claim: If all edge costs in a connected graph G are distinct, then G has a unique minimum spanning tree (MST).

Proof: Suppose, for contradiction, that there exist two different MST's T and T'. Then there is at least one edge e that is in T but not in T'. Removing e from T disconnects T into two components. Since T' is spanning, there is an edge f in T' that reconnects these two components. By the *cut property* (which states that the unique lightest edge crossing any cut must be in every MST), and since edge weights are distinct, we must have $w(e) < w(f)$ or $w(f) < w(e)$. But by exchanging f for e in T (or vice-versa), one obtains a spanning tree with a smaller total cost, contradicting the minimality of the MST. Therefore, the MST is unique. □

Problem 3 (10 pts)

Prove or disprove the following: Given any undirected graph G with weighted edges, and a minimum spanning tree T for that G, there exists some sorting of the edge weights $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$, such that running Kruskal's algorithm with that sorting produces the tree T.

Solution

Statement: Given any undirected graph G with weighted edges and a minimum spanning tree T for G, there exists some sorting of the edge weights

$$w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$$

such that running Kruskal's algorithm with that sorting produces the tree T.

Proof: The idea is to choose a total order on the edges so that all edges in T are ordered before any edge not in T. More precisely, order the edges as follows:

- For any $e, f \in T$, if $w(e) < w(f)$, then e precedes f.
- For any $e \in T$ and $g \notin T$, let e precede g.
- For any $g, h \notin T$, order them by their weights.

With this ordering, when Kruskal's algorithm is executed, every edge of T will be considered before any non-tree edge that could create a cycle with already chosen edges. Therefore, the algorithm will choose exactly the edges of T, and so T is produced. □

Problem 4 (10 pts)

Given a sequence of n real numbers $a_1, \cdots, a_n$ where n is even, design a polynomial time algorithm to partition these numbers into $n/2$ pairs in the following way: For each pair we compute the sum of its numbers. Denote $s_1, \cdots, s_{n/2}$ these $n/2$ sums. Your algorithm should find the partition which minimizes the maximum sum. For example, given numbers 3, -1, 2, 7 you should output the following partition: $\{3, 2\}, \{-1, 7\}$. In such a case the maximum sum is $7 + (-1) = 6$.
Note: Give the pseudo-code for your algorithm and analyze its time complexity to show that it has polynomial time complexity.

Solution

Algorithm: Given a sequence $a_1, a_2, \ldots, a_n$ (with n even), we wish to partition the numbers into $n/2$ pairs so that the maximum sum among the pairs is minimized. The following greedy algorithm works:

1. Sort the numbers into non-decreasing order: let $b_1 \leq b_2 \leq \cdots \leq b_n$.
2. For $i = 1$ to $n/2$, pair b_i with b_{n-i+1}.

Pseudo-code:

```
MinimizeMaxPairSum(a[1..n]):
    sort(a)           // O(n log n)
    maxSum = -infinity
    for i = 1 to n/2:
        sum = a[i] + a[n-i+1]
        if sum > maxSum:
            maxSum = sum
    return maxSum
```

Analysis: The sort takes $O(n \log n)$ time and the pairing loop takes $O(n)$. Therefore, the overall time complexity is $O(n \log n)$, which is polynomial.

Problem 5 (10 pts)

Given two edge disjoint spanning trees T_1, T_2 (**NOT** necessarily minimum) on n vertices, prove that for every edge $e \in T_1$ there exists an edge $f \in T_2$ that satisfies both of the following criteria:

- $T_1 - e + f$ is a spanning tree (on n vertices).
- $T_2 - f + e$ is a spanning tree (on n vertices).

Note: Two edge-disjoint spanning trees in a graph G are two spanning trees T_1 and T_2 such that they share **NO** common edges.

Solution

Claim: Let T_1 and T_2 be two edge-disjoint spanning trees on the same vertex set V (with $|V| = n$). Then for every edge $e \in T_1$, there exists an edge $f \in T_2$ such that both

$$T_1 - e + f \quad \text{and} \quad T_2 - f + e$$

are spanning trees.

Proof: Fix an edge $e \in T_1$. Removing e from T_1 disconnects it into two components C_1 and C_2. Since T_2 is a spanning tree (and is connected) and T_1 and T_2 are edge-disjoint, there must exist at least one edge $f \in T_2$ that has one endpoint in C_1 and the other in C_2. Adding f to $T_1 - e$ reconnects the two components (and does not create a cycle), so $T_1 - e + f$ is a spanning tree. Similarly, removing f from T_2 disconnects T_2 into two components; since e connects C_1 and C_2 (by the same partition induced in T_1), adding e to $T_2 - f$ yields a spanning tree. □

Problem 6 (Bonus 10 pts)

Solve the following problem with a program which should use one of the greedy algorithms we discussed. Please submit your **source code** to Canvas and attach a **screenshot** of the running output of your code here.

There are n houses in a village. We want to supply water for all the houses by building wells and laying pipes.

For each house i, we can either build a well inside it directly with cost wells$[i-1]$ (note the -1 due to 0-indexing), or pipe in water from another well to it. The costs to lay pipes between houses are given by the array pipes where each pipes$[j] = [\text{house1}_j, \text{house2}_j, \text{cost}_j]$ represents the cost to connect house1$_j$ and house2$_j$ together using a pipe. Connections are bidirectional, and there could be multiple valid connections between the same two houses with different costs.

Return the minimum total cost to supply water to all houses.

The function template:

minCostSupplyWater(n: int, wells: List[int], pipes: List[List[int]]) → int

Test your code with the following cases:

Testing Case 1

- **Input**: n = 3, wells = [1,2,2], pipes = [[1,2,1],[2,3,1]]
- **Output**: 3

Testing Case 2

- **Input**: n = 4, wells = [1, 2, 2, 1], pipes = [[1, 2, 1], [1, 2, 3], [2, 3, 2], [3, 4, 3], [1, 4, 2]]
- **Output**: 5

Testing Case 3

- **Input**: n = 5, wells = [10, 2, 2, 10, 2], pipes = [1, 2, 1], [2, 3, 1], [3, 4, 1], [1, 4, 2], [2, 5, 2]]
- **Output**: 7

Solution

Water Supply Problem: There are n houses. For each house i one may build a well at cost wells$[i-1]$ or connect house i to another house via a pipe (with cost given in the list pipes). We wish to minimize the total cost.

Algorithm Idea: Construct a new graph G' with $n+1$ vertices, where vertex 0 is a *virtual node*. For each house i add an edge $(0, i)$ with weight wells$[i-1]$. For every pipe $[u, v, c]$ in pipes, add an edge (u, v) with weight c. Then, compute a minimum spanning tree of G' (using, e.g., Kruskal's or Prim's algorithm). The total cost of this MST is the minimum cost to supply water.

Pseudo-code:

```
function minCostSupplyWater(n, wells, pipes):
    // Create graph with vertices 0,1,...,n where 0 is the virtual node.
    G = empty list of edges
    for i = 1 to n:
        add edge (0, i) with cost wells[i-1] to G
    for each [u, v, c] in pipes:
        add edge (u, v) with cost c to G

    // Compute MST of G (e.g., using Kruskal's algorithm)
    MST = Kruskal(G)
    totalCost = sum of costs of edges in MST
    return totalCost
```

Time Complexity: If there are m pipes, then the graph has $O(n+m)$ edges. Sorting the edges takes $O((n+m)\log(n+m))$ time. Therefore, the algorithm is polynomial.
