Problem Set #2 - Solutions

Giacomo Negri (3155287),
Matteo Omizzolo (3316152),
Marek Chadim (3348781)

March 17, 2025

Problem 1

Overview Solution

The main idea is to use a **reverse graph**: for every edge $(u, v) \in E$ in the original graph the reverse graph include (v, u). The algorithm works by processing the nodes in increasing order. For each node v (from 1 to n), we perform a BFS in the reverse graph starting at v. Every node u reached in this BFS satisfies that there is a path from u to v in the original graph. If the answer for u has not yet been set, we update it to v. Since nodes are processed in increasing order, the first update guarantees that the value assigned is the smallest possible.

Algorithm Description

1. Reverse the original graph to create `reverse_graph`, where each edge (u, v) becomes (v, u). So the edges point in the opposite direction of the original one.
2. Given that we have n nodes and m edges we represent it in an adjacency list, then we create a list result of size n initialized with infinity. The array `ans`[u] will eventually store the smallest numbered node v that is reachable from u.
3. For each candidate node v from 1 to n:
   (a) Perform a BFS in the `reverse_graph` starting from v.
   (b) For every node u reached by the BFS, if `ans`[u] is still ∞, update it to v.

   Each node is updated only once and every edge is considered at most once overall.

Pseudocode

```
function solve(graph, n, m):

    // Build the reverse_graph
    reverse_graph = [ [] for i = 1 to n ]
    for u = 1 to n:
        for each v in graph[u]:
            reverse_graph[v].append(u)

    // Initialize answer
    ans = [float('inf')] * n

    // Process nodes
    for v = 1 to n:
        Queue Q = new empty queue
        Q.enqueue(v)
        while Q is not empty:
            u = Q.dequeue()
            if ans[u] == infinity:
                ans[u] = v
                for each neighbor in reverse_graph[u]:
                    if ans[neighbor] == infinity:
                        Q.enqueue(neighbor)
    return ans
```

Since we process nodes in increasing order (from 1 to n), when a node u is first reached in a BFS from v, it is guaranteed that v is the smallest numbered node such that there is a path from u to v in the original graph. Once `ans`[u] is set, it is not updated again.

Time Complexity

- Reversing the graph: $O(m)$ because we need to pass through all the edges.
- The BFS across all iterations will process each node and edge at most once. Therefore, the total time for the BFS steps is $O(n + m)$.

Thus, the overall running time of the algorithm is $O(n + m)$.

Problem 2

Overview Solution

The key idea is to use a **segment tree** that supports range maximum queries (RMQ). Each leaf node in the segment tree stores a tuple (value, index) corresponding to the array element and its index. The internal nodes store the maximum tuple among their two children, comparing first by value and then by index (to choose the leftmost occurrence in case of ties).

Algorithm Description

1. Build a segment tree where each leaf node stores a tuple $(value, index)$ for each element in the array. The internal nodes of the tree store the maximum value tuple among their children, preferring the leftmost index when values are equal. The tree is built in such a way:

```
function buildSegmentTree(array, n):
tree = new array of size 2n

for i = 0 to n-1:
    tree[i+n] = (array[i], i)

for i in range(n-1, 0,-1):
    tree[i]=max(tree[i*2], tree[i*2+1])

return tree
```

2. For each query (s_i, q_i):

   (a) Restrict the search range to identify the segment of the tree corresponding to indices $[s_i, n]$. The actual set of candidate solutions will differ in the case that s_i is odd, since we would be dealing with a right-child. In this case, if tree$[s_i + n]$ does not represent a valid solution, then the candidate solutions range of the tree becomes: $[s_i + n + 1, 2n]$.

```
if si%2!=0:
    if tree[si+n][0]>=qi:
        return si
    else:
        range=[si+n+1:]
else:
    range=[si+n:] #stays the same
```

(b) Ascend the segment tree over the restricted range to obtain the maximum value, which will be the *head* of the range. If this maximum is less than q_i, then no valid index exists and the answer is -1.

(c) If a valid index exists, we need to locate the leftmost index j in the range with $a_j \geq q_i$. First, recover the *head*, then *descend* the tree from this node:

- If the left child of the current node has a maximum value $\geq q_i$, then move to the left child.
- Otherwise, move to the right child.

When a leaf is reached, its index is the answer.

The descent pseudocode is as follows:

```
j = head
while j < n:  // while j is not a leaf
    if tree[2*j][0] >= q_i:
        j = 2*j
    else:
        j = 2*j + 1
return j - n
```

Time Complexity

- Building a Segment Tree: $O(n)$
- Restrict possible candidate solutions: $O(1)$
- Finding and checking the head of the interval: upper limit $O(\log_2 n)$ for finding and $O(1)$ for checking.
- Descending the tree: upper limit $O(\log_2 n)$.

Thus, for m queries the total query time is $O(m \log_2 n)$, and the overall time complexity is:

$$O(n) + O(m \log_2 n) = O(n + m \log_2 n)$$