Solutions Guide

Data Structures Problem Set

Problem 1: Building Heights

Problem Summary

For each building, count how many consecutive buildings to its right are shorter, stopping when you encounter a building that is taller or equal.

Approach

Key Insight: Use a **monotonic stack** to efficiently track buildings that might block the view for buildings we've already seen.

Algorithm:

1. Process buildings from **right to left** (reverse order)

2. Maintain a stack of building indices where heights are in increasing order

3. For each building at position i:

• Pop all buildings from the stack that are shorter than current building

• If stack is empty after popping: all remaining buildings to the right are shorter

• If stack is not empty: the top of stack is the first taller/equal building

• Calculate count = (first_taller_index - current_index - 1) or (n - current_index - 1)

• Push current building index onto stack

Complexity

Time Complexity: O(n) - Each element is pushed and popped at most once

Space Complexity: O(n) - Stack storage

C++ Solution

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;

int main() {
    int n;
    cin >> n;
    vector<int> heights(n);

    for (int i = 0; i < n; i++) {
```

```
        cin >> heights[i];
    }

    stack<int> st;
    vector<int> result(n);

    for (int i = n - 1; i >= 0; i--) {
        while (!st.empty() && heights[st.top()] < heights[i]) {
            st.pop();
        }

        if (st.empty()) {
            result[i] = n - i - 1;
        } else {
            result[i] = st.top() - i - 1;
        }

        st.push(i);
    }

    for (int i = 0; i < n; i++) {
        cout << result[i];
        if (i < n - 1) cout << " ";
    }
    cout << "\n";

    return 0;
}
```

Example Trace

Input: [4, 3, 5, 2, 1]

Processing from right to left:

• i=4 (height 1): stack empty → count = 0, push 4

• i=3 (height 2): pop 4 (1 < 2), stack empty → count = 1, push 3

• i=2 (height 5): pop 3 (2 < 5), stack empty → count = 2, push 2

• i=1 (height 3): 5 ≥ 3, so stop → count = 2 - 1 - 1 = 0, push 1

• i=0 (height 4): pop 1 (3 < 4), then 5 ≥ 4 → count = 2 - 0 - 1 = 1

Output: [1, 0, 2, 1, 0]

Problem 2: Printer Queue

Problem Summary

Simulate a priority-based printer queue where tasks with higher priority print first. Find how many tasks print before your specific task.

Approach

Key Insight: Use a **queue** to simulate the printer's behavior, tracking both priority and original position of each task.

Algorithm:

1. Create a queue storing pairs of (priority, original_index)

2. Keep track of the maximum priority in the current queue

3. Repeatedly:

• Dequeue the front task

• If its priority equals the maximum priority:

- Print it (increment counter)

- If this is our target task, return counter - 1

- Update maximum priority for remaining tasks

• Otherwise, move it to the back of the queue

Complexity

Time Complexity: $O(n^2)$ worst case - Each task might cycle through the queue multiple times

Space Complexity: O(n) - Queue storage

C++ Solution

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
using namespace std;

int main() {
    int n, k;
    cin >> n >> k;

    vector<int> priorities(n);
    queue<pair<int, int>> q;

    for (int i = 0; i < n; i++) {
        cin >> priorities[i];
        q.push({priorities[i], i});
    }

    int count = 0;
```

```
        while (!q.empty()) {
            auto front = q.front();
            q.pop();

            int maxPriority = front.first;
            queue<pair<int, int>> temp = q;
            while (!temp.empty()) {
                maxPriority = max(maxPriority, temp.front().first);
                temp.pop();
            }

            if (front.first == maxPriority) {
                if (front.second == k) {
                    cout << count << "\n";
                    return 0;
                }
                count++;
            } else {
                q.push(front);
            }
        }

        return 0;
    }
```

Example Trace

Input: n=5, k=2, priorities=[1, 2, 3, 4, 5]

Queue: [(1,0), (2,1), (3,2), (4,3), (5,4)]

• Front: (1,0), max=5 → 1≠5, move to back

• Front: (2,1), max=5 → 2≠5, move to back

• Front: (3,2), max=5 → 3≠5, move to back

• Front: (4,3), max=5 → 4≠5, move to back

• Front: (5,4), max=5 → 5=5, PRINT (count=0)

• Front: (1,0), max=4 → 1≠4, move to back

• Front: (2,1), max=4 → 2≠4, move to back

• Front: (3,2), max=4 → 3≠4, move to back

• Front: (4,3), max=4 → 4=4, PRINT (count=1)

• Front: (1,0), max=3 → 1≠3, move to back

• Front: (2,1), max=3 → 2≠3, move to back

• Front: (3,2), max=3 → 3=3, index=k=2, RETURN count=2

Output: 2

Key Takeaways

Stack Problem: Monotonic stacks are powerful for problems involving "next greater/smaller element" or visibility/blocking scenarios. Process in reverse when looking at elements to the right.

Queue Problem: Simulation problems often require tracking additional metadata (like original position) alongside the primary data. Use pairs or custom structures to maintain this information.