PW SKILLS

Lesson Plan

Stack-3

PW SKILLS

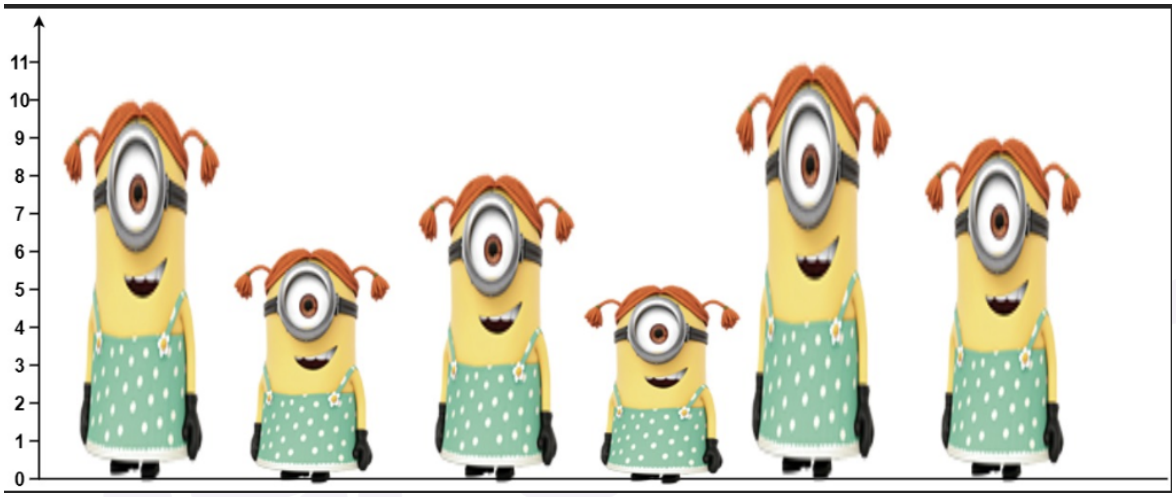
Q. Number of Visible People in a Queue (leetcode 1944)
There are n people standing in a queue, and they numbered from 0 to n - 1 in left to right order. You are given an array heights of distinct integers where heights[i] represents the height of the ith person.

A person can see another person to their right in the queue if everybody in between is shorter than both of them. More formally, the ith person can see the jth person if i < j and min(heights[i], heights[j]) > max(heights[i+1], heights[i+2], ..., heights[j-1]).

Return an array answer of length n where answer[i] is the number of people the ith person can see to their right in the queue.

Input: heights = [10,6,8,5,11,9]
Output: [3,1,2,1,1,0]

Explanation:



Person 0 can see person 1, 2, and 4.
Person 1 can see person 2.
Person 2 can see person 3 and 4.
Person 3 can see person 4.
Person 4 can see person 5.
Person 5 can see no one since nobody is to the right of them.

Code:

```
import java.util.ArrayList;
import java.util.List;

class Solution {
    public List<Integer> canSeePersonsCount(int[] heights) {
        List<Integer> stk = new ArrayList<>();
        List<Integer> ans = new ArrayList<>();

        for (int i = heights.length - 1; i ⩾ 0; i--) {
            int cnt = 0;
            while (!stk.isEmpty() && stk.get(stk.size() - 1) ⩽ heights[i]) {
```

```
                ++cnt;
                stk.remove(stk.size() - 1);
            }
            ans.add(cnt + (stk.size() > 0 ? 1 : 0)); //stk.size() > 0 means the current
person can see the first right person who is higher than him.
            stk.add(heights[i]);
        }

        // Reverse the answer list
        List<Integer> reversedAns = new ArrayList<>();
        for (int i = ans.size() - 1; i >= 0; i--) {
            reversedAns.add(ans.get(i));
        }
        return reversedAns;
    }
}
```

Ques: Sliding Window Maximum (LEETCODE 239)
You are given an array of integers nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position.

Return the max sliding window.

Input: nums = [1,3,-1,-3,5,3,6,7], k = 3
Output: [3,3,5,5,6,7]

Explanation:

Window position	Max
[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3 5] 3 6 7	5
1 3 -1 [-3 5 3] 6 7	5
1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7

Example 2:

Input: nums = [1], k = 1
Output: [1]

Code:

```
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.Vector;
```

```
class Solution {
    public Vector<Integer> maxSlidingWindow(Vector<Integer> nums, int k) {
        Deque<Integer> dq = new ArrayDeque<>();
        for (int i = 0; i < k - 1; i++) {
            while (!dq.isEmpty() && nums.get(dq.peekLast()) <= nums.get(i)) {
                dq.pollLast();
            }
            dq.offerLast(i);
        }
        int i = 0, j = k - 1;
        Vector<Integer> ans = new Vector<>(nums.size() + 1 - k);
        while (j < nums.size()) {
            while (!dq.isEmpty() && nums.get(dq.peekLast()) <= nums.get(j)) {
                dq.pollLast();
            }
            dq.offerLast(j);
            if (dq.peekFirst() < i) {
                dq.pollFirst();
            }
            ans.add(nums.get(dq.peekFirst()));
            i++;
            j++;
        }
        return ans;
    }
}
```

Q. Min Stack (leetcode 155)
Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the MinStack class:

MinStack() initializes the stack object.
void push(int val) pushes the element val onto the stack.
void pop() removes the element on the top of the stack.
int top() gets the top element of the stack.
int getMin() retrieves the minimum element in the stack.
You must implement a solution with O(1) time complexity for each function.

Example 1:

Input
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]

Output
[null,null,null,null,-3,null,0,-2]

Explanation

MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); // return -3
minStack.pop();
minStack.top(); // return 0
minStack.getMin(); // return -2

Code:

```
import java.util.Stack;

class MinStack {
    private Stack<Long> st;
    private long mini;

    /** initialize your data structure here. */
    public MinStack() {
        mini = Integer.MAX_VALUE;
        st = new Stack<>();
    }

    public void push(int val) {
        if (st.isEmpty()) {
            mini = val;
            st.push((long) val);
        } else {
            if (val < mini) {
                st.push(2L * val - mini);
                mini = val;
            } else {
                st.push((long) val);
            }
        }
    }

    public void pop() {
        if (st.peek() < mini) {
            mini = 2L * mini - st.pop();
        } else {
            st.pop();
        }
    }
```

```
    public int top() {
        if (st.peek() < mini) {
            return (int) mini;
        } else {
            return st.peek().intValue();
        }
    }

    public int getMin() {
        return (int) mini;
    }
}
```

PW SKILLS

THANK YOU !