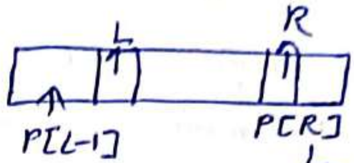
STL Application Using Stacks AND Maps

1. Find out no. of sub arrays with sum == k.

$$\sum_{i=L}^{R} a[i] = k$$



↳ It is the prefix sum up to index R

$\Rightarrow P[R] - P[L-1] = k$

$\Rightarrow P[L-1] = P[R] - k$

1. Maintain a map to count prefix sum.
2. ~~Initialize the count of subarrays~~ sum = 0
3. Iterate over the array for each index R :
   → Update prefix sum P[R].
   → Add to sum the count of prefix sums P[R] - k found in map.
   → Increment the count of prefix sum P[R] in the map.
4. Print sum

$\boxed{T.C : O(n \log n)}$

2. Largest Rectangle in Histogram

- Next smaller element - For every bar, find the ~~area~~ index of the ~~largest~~ next bar to the right that is shorter.
- Previous smaller element (PSE) - For every bar; find the index of nearest bar to the left that is shorter.

For NSE :-

1. Iterate from right to left.
2. Use a stack to keep indexes.
3. For each element pop stack until you find a smaller element.
4. If stack is empty, next smaller doesn't exist.
5. else, record top of stack.

For PSE :-

1. Iterate from left to right.
2. Use a stack similarly as above but in opp. dir^n.

Max Area

For each bar at index i :

$$Width = NSE(i) - PSE(i) - 1$$

$$Area = heights[i] \times Width$$

Take the max area among all bars.

```
class Solution {
public:
    int largestRectangleArea(vector<int>& heights) {

        stack<int> stk1,stk2;
        int n = heights.size();

        vector<int> pse(n), nse(n);

        //previous smaller element index
        pse[0] = -1;
        stk1.push(0);

        for(int i=1; i<n; i++){
            while(!stk1.empty() && heights[stk1.top()] >= heights[i]){
                stk1.pop();
            }

            if(stk1.empty())
            pse[i] = -1;

            else
            pse[i] = stk1.top();

            stk1.push(i);
        }

        //next smaller element index
        nse[n-1] = n;
        stk2.push(n-1);

        for(int i=n-2; i>=0; i--){

            while(!stk2.empty() && heights[stk2.top()] >= heights[i]){
                stk2.pop();
            }

            if(stk2.empty())
            nse[i] = n;

            else
            nse[i] = stk2.top();

            stk2.push(i);
```

```
    nse[n-1] = n;
    stk2.push(n-1);

    for(int i=n-2; i>=0; i--){

        while(!stk2.empty() && heights[stk2.top()] >= heights[i]){
            stk2.pop();
        }

        if(stk2.empty())
        nse[i] = n;

        else
        nse[i] = stk2.top();

        stk2.push(i);
    }

    int ans = 0;

    for(int i=0; i<n; i++){
        ans = max(ans, (nse[i]-pse[i]-1)*heights[i]);
    }

    return ans;
}
```

3. Trapping Rain Water

1. For each bar at i, calculate
   - Left Max : Highest bar to the left (including itself)
   - Right Max : " " " " right "

2. Water level above bar i :

$$Water_i = max(min(left\_max, right\_max) - h[i], 0)$$

3. Total trapped water is sum over all indices.

```cpp
class Solution {
public:
    int trap(vector<int>& height) {
        int n = height.size();
        vector<int> pref(n);
        vector<int> suff(n);
        pref[0] = height[0];
        suff[n-1] = height[n-1];

        for(int i=1; i<n; i++)
        pref[i] = max(height[i], pref[i-1]);

        for(int i=n-2; i>=0; i--)
        suff[i] = max(suff[i+1], height[i]);

        int ans = 0;
        for(int i=1; i<n-1; i++){

            int m1 = pref[i-1];
            int m2 = suff[i+1];
            ans += max((min(m1,m2) - height[i]),0);
        }
        return ans;
    }
};
```