BRAC University Competitive Programming Workshop

Handouts for Day 2

Shafin Alam, Shehran Rahman

Department of CSE
BRAC University
13 April 2025

Monotonic Stack – Concepts and Applications

Overview

Monotonic stacks are special stacks that maintain a consistent increasing or decreasing order. They are widely used in problems that require comparisons with "nearest" elements – like nearest smaller or greater values to the left/right.

Problem Statement

You are given an array of n integers. For each position in the array, your task is to find the nearest position to its left that contains a strictly smaller value.

Constraints

- $1 \leq n \leq 2 \cdot 10^5$
- $1 \leq x_i \leq 10^9$

Solution

Naive Approach

For each element in the array, check all the elements to its left one by one, moving backward, until you find a smaller element.

Time Complexity

Worst-case: $\mathcal{O}(n^2)$ — for each element, you may have to scan all previous elements.

Efficient Approach

This is where monotonic stacks come in. To find the Nearest Smaller Value to the Left, we use a stack to keep track of potential candidates.

- We maintain a monotonic stack, where elements in the stack are in increasing order of value (from bottom to top).
- For each index `i`, we pop from the stack as long as the top of the stack has a value greater than or equal to the current value `arr[i]`.

Why do we pop greater values from the stack?

- Because we're looking for the nearest smaller value, and in the future (for upcoming elements), we will encounter the current index first.
- So, any greater value at a lower index becomes irrelevant — if there's already a smaller value available, the greater one is no longer useful.
- That's why we can safely pop all greater values from the stack.

Once we've removed all larger or equal elements:

- If the stack is empty $\rightarrow$ there is no smaller element to the left, so we add `0`.
- If the stack is not empty $\rightarrow$ the top of the stack is the nearest smaller, and we take its index.

After processing the current element, we push it's index onto the stack for future comparisons.

Time Complexity (Explanation)

We process the array from left to right, one element at a time.

For each element:

- We remove elements from the stack that are greater than or equal to the current value.
- Then we push the current index onto the stack.

Even though there's a loop inside a loop, every element is:

- Pushed to the stack only once
- Popped from the stack at most once

So the total number of stack operations (push + pop) across the whole array is at most $2n$.
Time Complexity: $\mathcal{O}(n)$

```cpp
stack<int> st;
for(int i = 1; i <= n; i++) {
    cin >> a[i];
    while(!st.empty() && a[st.top()] > a[i]) {
        st.pop();
    }
    ans[i] = st.empty() ? 0 : st.top();
    st.push(i);
}
```

The Histogram Problem

Problem Statement

You are given a histogram consisting of `N` bars, where each bar has a width of `1` and a positive integer height. Your task is to find the area of the largest rectangle that can be formed using one or more consecutive bars of the histogram.

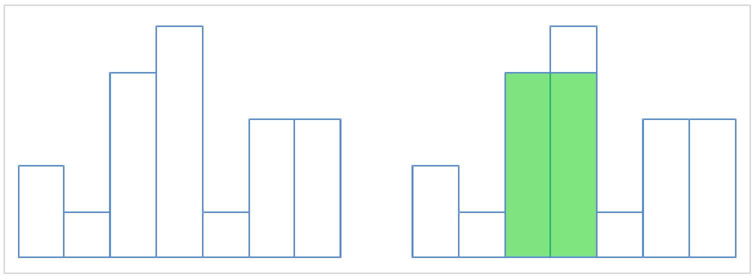


Figure 1: histogram

For example, if we choose a group of consecutive bars (like the green bars in the figure), the height of the rectangle is determined by the shortest bar in that group. This is because all bars in a rectangle must be at least as tall as the rectangle itself — otherwise, it wouldn't form a proper rectangle.

Constraints

- $1 \leq N \leq 2 \cdot 10^5$
- $1 \leq h_i \leq 10^9$

Solution

To find the largest rectangle, we fix each bar `i` as the height of the rectangle and try to extend it as much as possible to both sides — while keeping the height at least `h[i]`.

There are two key observations:

1. **The height of the largest rectangle will always be the height of a peak bar.** — Suppose, for contradiction, that the height h of the largest rectangle is not the height of any peak (i.e., not equal to any bar in the histogram). This means:

   - All bars within the selected range must have heights greater than or equal to `h`.
   - But if none of those bars are exactly equal to `h`, they must all be strictly greater than `h`.

   In that case, we could actually increase the height of our rectangle to the minimum height in that range — and that would give us a larger area than before. This contradicts our assumption that `h` gave the largest area. So, the optimal height must match the height of at least one bar — making it a valid peak to consider.

2. **To maximize the area, we need to maximize the width**.

   - Once the height is fixed as `h[i]`, we can only include **adjacent bars with height** $\geq$ `h[i]`.
   - So, we extend the rectangle **to the left and right** as far as we can, until we hit a bar that is **strictly smaller than** `h[i]`.

This is where Nearest Smaller to Left and Nearest Smaller to Right come in:

- `NSL[i]` gives us the nearest bar to the left that is smaller — so we can extend left up to `NSL[i]+1`.
- `NSR[i]` gives us the nearest bar to the right that is smaller — so we can extend right up to `NSR[i]-1`.

Then we compute:

- `width = NSR[i] - NSL[i] - 1`
- `area = heights[i] * width`

We repeat this for all bars and take the maximum area.

Time Complexity

We already know:

- Finding the **Previous Smaller Element** for all bars takes $\mathcal{O}(n)$.
- Finding the **Next Smaller Element** for all bars also takes $\mathcal{O}(n)$.

After that:

- We calculate the area for each bar using the precomputed NSL and NSR arrays, which takes another $\mathcal{O}(n)$ pass.

Time Complexity: $\mathcal{O}(n)$

CPP Code

```cpp
int largest_Rectangle(){
    int n;
    cin >> n;
    int h[n+5];
    for(int i = 1; i <= n; i++) cin >> h[i];

    int NSL[n+5];
    int NSR[n+5];
    h[n+1] = 0;

    stack<int>st;
    st.push(n+1);

    for(int i = n; i>=1; i--)
    {
        while(!st.empty() && h[st.top()]>=h[i]) st.pop();
        NSR[i] = st.top();
        st.push(i);
    }
    while(!st.empty()) st.pop();

    h[0] = 0;
    st.push(0);

    for(int i = 1; i <= n; i++)
    {
        while(!st.empty() && h[st.top()]>=h[i]) st.pop();
        NSL[i] = st.top();
        st.push(i);
    }
    int ans = 0;
    for(int i = 1; i <= n; i++)
    {
        ll res = (NSR[i]-NSL[i]-1)*h[i];
        ans = max(ans, res);
    }
    return ans;
}
```

Largest Sub-Rectangle in Binary Grid

Problem Statement

Given a binary matrix filled with 0's and 1's, find the largest rectangle containing only 1's and return its area.

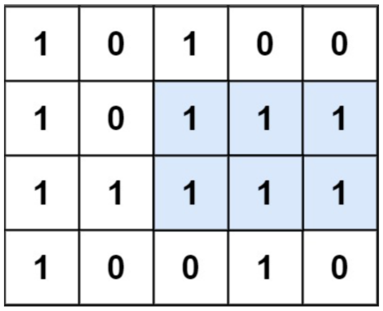
Input/Output Format

Input:

```
matrix = [
  ["1","0","1","0","0"],
  ["1","0","1","1","1"],
  ["1","1","1","1","1"],
  ["1","0","0","1","0"]
]
```

Output: 6

Visualization



1	0	1	0	0
1	0	1	1	1
1	1	1	1	1
1	0	0	1	0

Source: https://leetcode.com/problems/maximal-rectangle/description/

Solution Approaches

Naive Approach: Brute Force Checking All Rectangles

- Iterate through all possible rectangles in the matrix
- For each rectangle, check if it contains only 1's
- Track the maximum area found

- **Time Complexity:** $O((m \times n)^3)$ - Extremely inefficient

Optimized Approach Using Histograms

Key Observations

- Each row can be treated as the base of a histogram where the height at each column is the number of consecutive 1's above it (including current row)
- For each row, we can use the histogram algorithm (with stack) to find the largest rectangle

Implementation

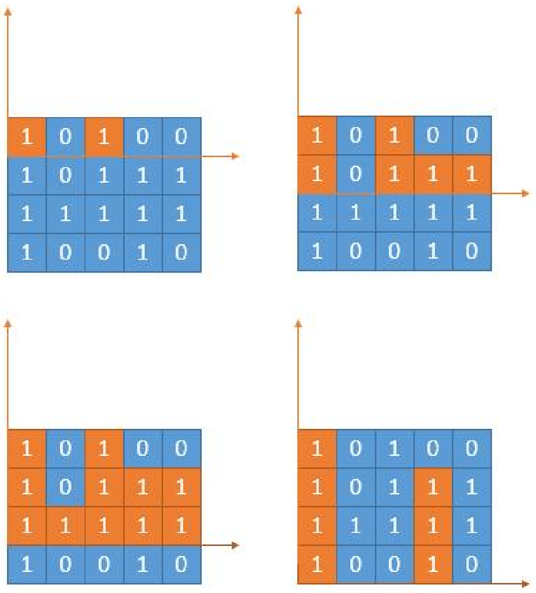
- Build a matrix or modify the given one such that:

$$\text{matrix}[row][col] += \text{matrix}[row-1][col] \quad \text{if} \quad \text{matrix}[i][j] \neq 0$$

- This provides the bar heights of the histogram for each row
- Globally track the largest rectangle found across all rows
- **Complexity:** $O(m \times n)$

CPP Code

```cpp
int maximalRectangle(vector<vector<char>>& matrix) {
    int row_len = matrix.size(), col_len = matrix[0].size();
    vector<vector<int>> mat(row_len, vector<int> (col_len));

    for(int row = 0; row < row_len; ++row) {
        for(int col = 0; col < col_len; ++col) {
            mat[row][col] = matrix[row][col] - '0';
        }
    }
    for(int col = 0; col < col_len; ++col) {
        for(int row = 1; row < row_len; ++row) {
            if(mat[row][col] != 0) {
                mat[row][col] += mat[row - 1][col];
            }
        }
    }

    int ans = 0;
    for(int row = 0; row < row_len; ++row) {

        // running the histogram algo for each row
        vector<int> heights = mat[row];
        stack<int> st;
        int n = heights.size();
        vector<int> nsr(n), nsl(n);
        for(int i = 0; i < n; ++i) {
            while(!st.empty() && heights[st.top()] >= heights[i
               ]) {
                st.pop();
            }
            if(!st.empty()) nsl[i] = st.top();
            else nsl[i] = -1;
            st.push(i);
        }

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

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

    return ans;
}
```



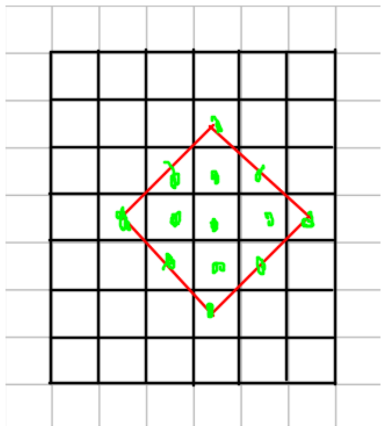
Source: `https://leetcode.wang/leetCode-85-Maximal-Rectangle.html`

Notice how we can find the maximal rectangle considering the heights with contiguous ”1”s. For efficiently finding the heights we process/add their sums as we go down the rows (until interrupted by a ”0”).

Maximum Diagonal Square Sum

Problem Statement

Given an **N×M** grid of integers, find the maximum possible sum of elements inside a square submatrix that is placed diagonally (i.e., its sides are aligned with the diagonals of the grid, not the rows/columns). The entire square must fit within the grid boundaries.



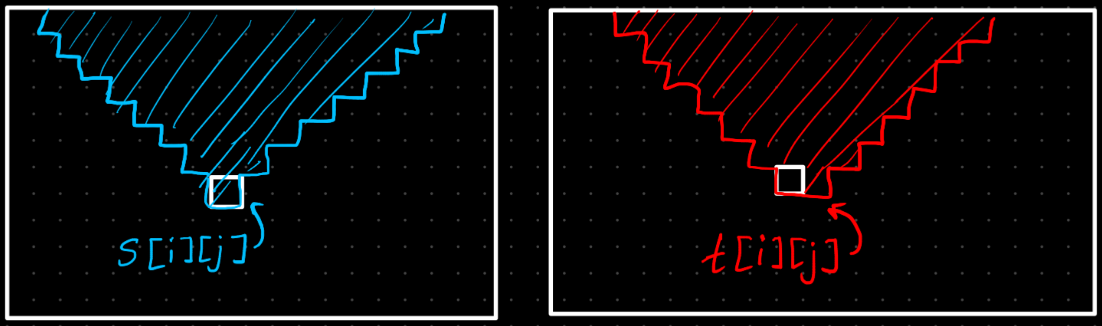
In the black grid, the red-bordered region is a valid diagonally placed square submatrix, as all its corners lie within the grid boundaries.

Constraints:

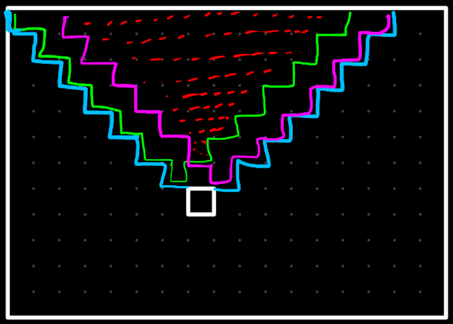
- $1 \leq N, M \leq 3 \times 10^6$
- $N \times M \leq 3 \times 10^6$
- $0 \leq F_{ij} \leq 10^9$

Solution

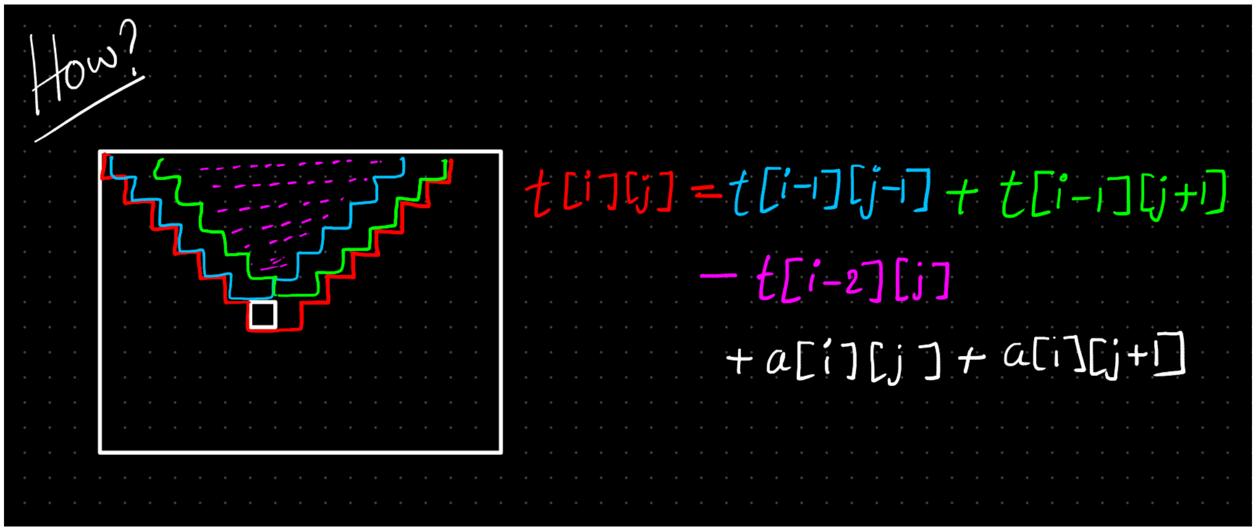
Preprocess two types of sums:

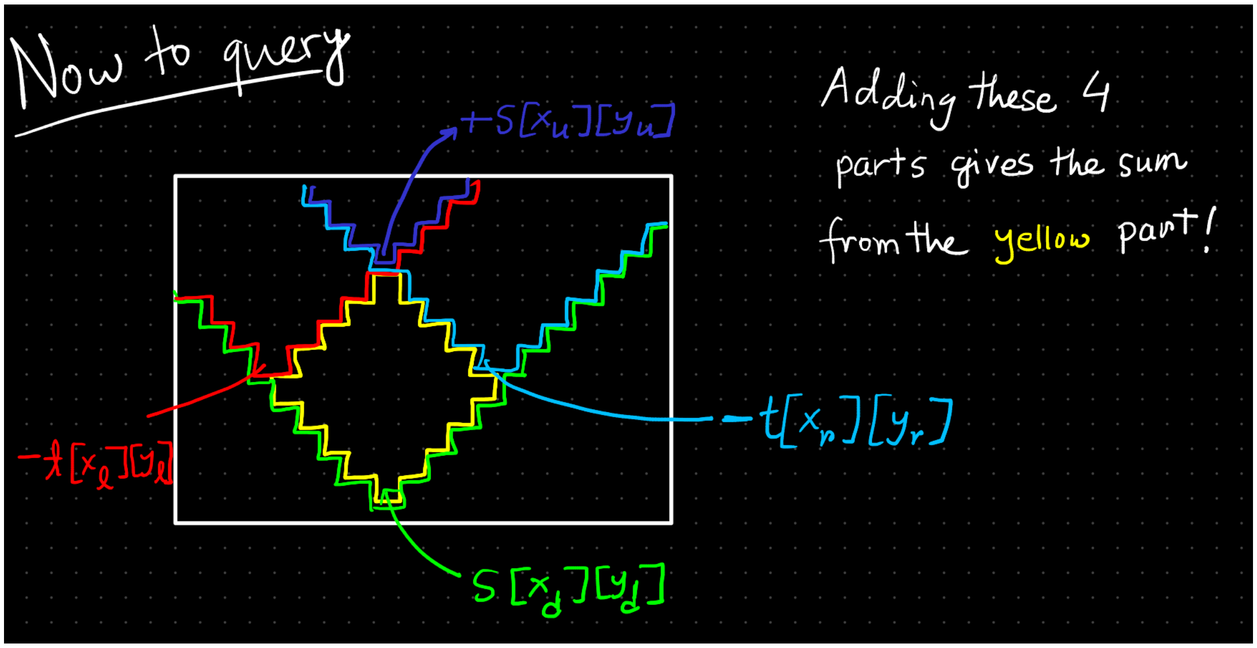


How?



$$s[i][j] = s[i-1][j-1] + s[i-1][j+1] - s[i-2][j] + a[i][j] + a[i-1][j]$$





Time Complexity: $\mathcal{O}(N * M)$