Stacks: Next Greater Elements

Relevel
by Unacademy

List of Concepts Involved (5 mins)

Stacks

Next Greater Elements with Example

Problems on Previous Greater/Smaller Elements with Example and time complexity analysis

Stacks

It is a linear data structure with the Last In First Out(LIFO) principle. This means the last element inserted in stack first will be removed first.

Example: Consider a Pile of plates placed one above the other. Now you want to put another plate on top. If you want that the top one remove the last plate and if you want the bottom one you have to remove all plates in the stack

Stack representation similar to a pile of plate

Next Greater Elements with Example

Ganesh is in a line, and the tall man in front of him is obstructing his view, which irritates him.
Ganesh now wants to know how tall the tall person in front of him is.
Ganesh wants to find a general solution, thus he wants to find the height of the person who is nearer to that person and obstruct the view for each individual.
He prints 0 for the final person or if no one is taller.
This is nothing but we need to find the next greater element for each element in the array

Let's divide the problem into two parts:

a. next bigger element
b. for every item in an array

consider an array

For 2

The first element on the right side which is larger than 2 is 7.

For 7, The next greater element is 8

For 3, The next greater element is 5

For 5, The next greater element is 6

For 4, The next greater element is 6

For 6, The next greater element is 8

For 8, the next greater element

So the final array would look like

7,8,5,6,6,8,0

```
Input:  [2, 7, 3, 5, 4, 6, 8]
Output: [7, 8, 5, 6, 6, 8, -1]


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


Note that the next greater element for the last array
element is always -1.
```

The two possible solutions are

1. Brute Force Approach
2. Using Stack

Brute Force Approach

Two nested loops are used in this example.
Each array element is traversed from left to right in the outer loop. All elements to the "right" of the element specified by the outer loop are considered by the inner loop. Terminate the inner loop as soon as the first larger element is discovered, which will be the next greater element for the outer loop's element.

```
let nextGreater_array = (arrs, num = arrs.length) => {
  for(let start = 0; start< num; start++){
    let next_ele = -1;


    for(let j = start; j < num; j++){
      if(arrs[start] < arrs[j]){
        next_ele = arrs[j];
        break;
```

```
      }
    }

    console.log(next_ele);
  }
}
```

```
Input:
nextGreater_array([11, 13, 21, 3]);

Output: 13,21, -1, 1
```

Complexity analysis

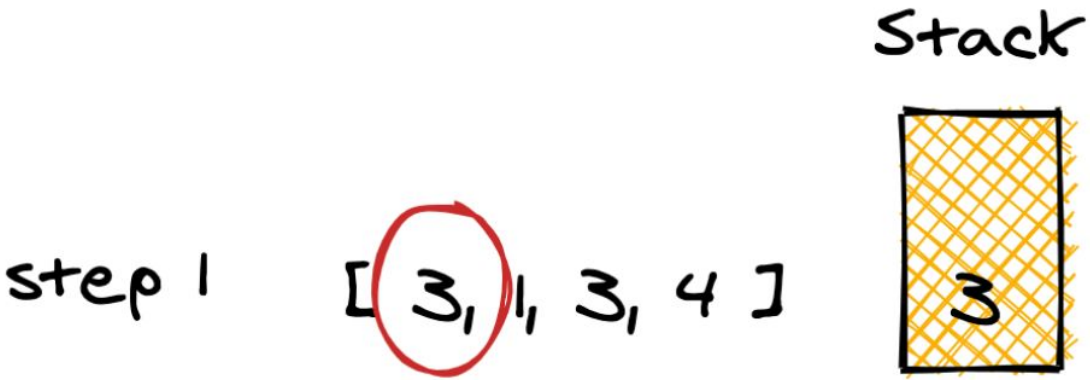
Time Complexity is O(n^2)

Space Complexity is O(n)

Using Stack Data structure

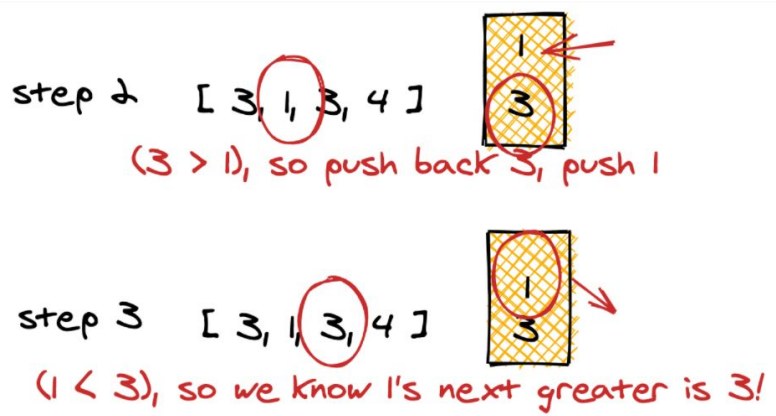
Using extra space, the time complexity can be easily reduced to linear. The stack data structure can be used here
We can reduce to one loop instead of two loops by using stack

Step 1.
Push the first element to the top of the stack.



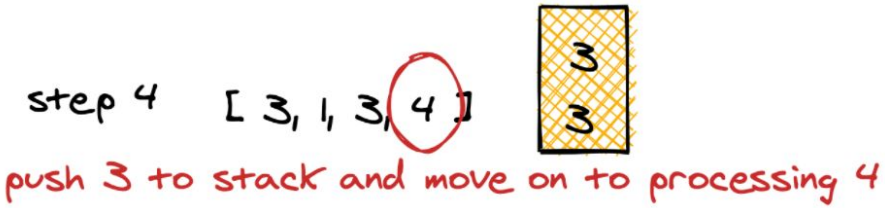
Steps 2 and 3 In a for-loop, iterate through the remaining elements.

- The iterated on/current element should be marked as the next element.
- If the stack is not empty, compare the top element with the next.
- If next is greater than the top, remove the element from the stack. This signifies that we've discovered that the next bigger element for the popped number is next
- Continue popping off the top of the stack when the popped element is smaller than the next. As a result of the logic, next becomes the next greater element for all of the popped items.



Step 3 Push the next variable onto the stack after iterating through all of the existing items.

Step 4 When the loop in step 2 is finished, remove all the elements from the stack and replace them with 0 as the next greater element.



```
class Stack_a {
    constructor() {
        this.stack_a = [];
    }

    isEmpty() {
        return this.stack_a.length === 0;
    }

    push(num_a) {
        this.stack_a.push(num_a);
    }

    pop() {
        if (this.isEmpty())
            throw 'Stack Underflow';
        return this.stack_a.pop();
    }
    peek() {
        if (this.isEmpty())
            throw null;
        return
this.stack_a[this.stack_a.length-1]
    }
}

let nextGreaterWithStackArray = (arrs,
num = arrs.length) => {
  let next_ele, elements;
let stack_data = new Stack_a();

  stack_data.push(arrs[0]);

  for(let start = 0; start < num;
start++){
    next_ele = arrs[start];
```

```
if(!stack_data.isEmpty()){
  elements = stack_data.pop();


  while(elements < next_ele){
    console.log(next_ele);
    if(stack_data.isEmpty()){
      break;
    }

    elements = stack_data.pop();
  }

  if(elements > next_ele){
    stack_data.push(elements)
  }
}
```

```
        elements = stack_data.pop();
      }

      if(elements > next_ele){
        stack_data.push(elements)
      }
    }

    stack_data.push(next_ele);
  }

while(!stack_data.isEmpty()){
    elements = stack_data.pop();
    next_ele = 0;
    console.log(next_ele);
  }
}
```

```
Input:
nextGreaterWithStackArray ([11, 13, 21,
3]);
Output:
13
21
0
0
```

Complexity Analysis

Time Complexity is O(n)

Space Complexity is O(n)

```
Example:


Input: "abbaca"
Output: "ca"
Explanation:
For example, in "abbaca" we could remove "bb" since the letters are
adjacent and equal, and this is the only possible move.  The result of
this move is that the string is "aaca", of which only "aa" is
possible, so the final string is "ca".
```

This can be solved by two ways

- Recursion
- Stack

Recursion

Remove adjacent duplicates recursively in the string till there are no duplicates left.

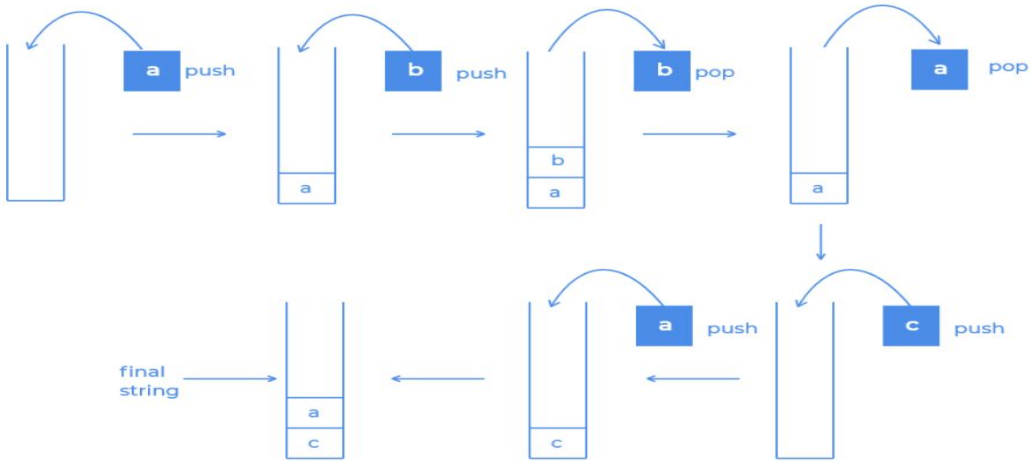
```
var removeDuplicates = function(S, K) {
     for (let i = 1, count = 1; i < S.length; i++) {
     S.charAt(i) === S.charAt(i-1) ? count++ : count = 1
     if (count === K)
          S = removeDuplicates(S.slice(0, i-K+1) + S.slice(i+1), K);
     }
     return S
};
console.log(removeDuplicates("pbbcggttciiippooaais", 2));
```

```
Output: "ps"
```

Stack

A stack can be used to overcome this problem. We can add all of the characters to the stack one by one. While adding one element to the stack, we check the top of the stack; if it is equal to our current character (i.e., two adjacent characters are the same), we pop that element and do not add our current character to the stack. As a result, we may ignore all of the neighboring characters.

Finally, we must reverse the string since they are returned in reverse order when we pop characters off the stack.



Below are the Steps

1. Create an empty stack array.
2. Iterate through the input string.
3. Check at each iteration if the top (last) element of the stack is equal to the current element of the input string.
4. If true, remove the top element of the stack.
5. If not, add the current element of the string on top of the stack.
6. Return all elements of the stack as a string after joining them.

```
var removeDuplicates = function(S) {

  // empty array

  let stack = []

  // iterate over S

  for (let char of S) {

     // check if the last element of
the stack is equal to the current
element of S

    if(stack[stack.length - 1] ===
char){

     // true, remove the last element
of the stack.

     stack.pop()

     } else {

     // false, add the current
element of S to the end of the
stack.

     stack.push(char)

     }

  }

  // return stack as a string

  return stack.join('');

};

console.log(removeDuplicates("pbbcgg
ttciiippooaais", 2));
```

```
Output: "ps"
```

Time Complexity: O(n)
Space Complexity: O(n)

2.Sum of Subarray Minimums

Given an array of integers arr, find the sum of min(b), where b ranges over every (contiguous) subarray of arr. Since the answer may be large, return the answer modulo 109 + 7

```
Input: arr = [3,1,2,4]
Output: 17
Explanation:
Subarrays are [3], [1], [2], [4], [3,1], [1,2], [2,4], [3,1,2], [1,2,4], [3,1,2,4].
Minimums are 3, 1, 2, 4, 1, 1, 2, 1, 1, 1.
Sum is 17.
```

```
Input: arr = [11,81,94,43,3]
Output: 444
```

Stack

For each number in a given array, we will count the number of subarrays in which the number appears as the minimum. Then, we will multiply it with the number's value and add it to a sum.
The number of subarrays in which the number appears as the
minimum = distance between current element and element less than current element on the left (Previous Less Element) * distance between current element and element less than current element on the right (Next Less Element)

We will fill two arrays PLE[] and NLE[], where PLE[i] is the index of previous less element of arr[i] and NLE[i] is the index of next less element of arr[i] using Stack. Then, we will
iterate through given array and at each index we will add (i-PLE[i]) * (NLE[i]-i) * arr[i] to a global sum. If there isn't PLE for arr[i], PLE[i] will be -1, and if there isn't NLE for arr[i] NLE[i] will be arr.length.

```
var sumSubarrayMins = function(arr) {
    let M = Math.pow(10, 9) + 7;
    let PLE = new Array(arr.length).fill(-1);
    let NLE = new Array(arr.length).fill(arr.length);
    findPLE(PLE, arr);
    findNLE(NLE, arr);
    let sum = 0;
    for (let i = 0; i < arr.length; i++) {
        sum = (sum + (i-PLE[i]) * (NLE[i]-i) * arr[i]) % M;
    }
    return sum;
    // T.C: O(N)
    // S.C: O(N)
};

function findPLE(PLE, arr) {
    let stack = [];
    for (let i = 0; i < arr.length; i++) {
        while (stack.length > 0
               && arr[stack[stack.length-1]] >= arr[i]) {
```

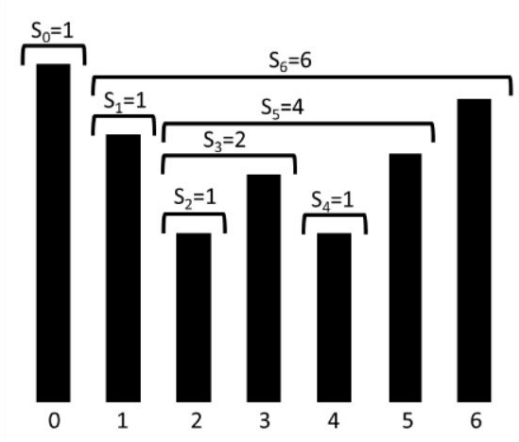
```
function findPLE(PLE, arr) {
    let stack = [];
    for (let i = 0; i < arr.length; i++) {
        while (stack.length > 0
               && arr[stack[stack.length-1]] >= arr[i]) {
                // we use less than or equal to
                // in either PLE or NLE in order to not miss out on subarrays
                // where there are duplicates of minimum element
            stack.pop();
        }
        if (stack.length > 0) {
            PLE[i] = stack[stack.length-1];
        }
        stack.push(i);
    }
}
```

```
function findNLE(NLE, arr) {
    let stack = [];
    for (let i = arr.length-1; i >= 0; i--) {
        while (stack.length > 0
               && arr[stack[stack.length-1]] > arr[i]) {
            stack.pop();
        }
        if (stack.length > 0) {
            NLE[i] = stack[stack.length-1];
        }
        stack.push(i);
    }
}

const arr = [3, 1, 2, 4]

console.log(sumSubarray(arr));
```

3.Stock Span Problem

The stock span problem is a sort of financial problem involving stocks in which the daily price of a stock is provided.

The stock's price today is defined as the max number of consecutive days (beginning today and working backwards) where the stock's price was less than or equal to today's price.



Example -

```
Input: [100, 80, 60, 70, 60, 75, 85]
Output: [1, 1, 1, 2, 1, 4, 6]

Input: [31, 27, 14, 21, 30, 22]
Output: [1, 1, 1, 2, 4, 1]
```

We can solve this using two approaches

1. Brute-force method
2. Stack

Brute-Force Method

Two loops will be used in this strategy. The outer loop will run over all of the array items, while the inner loop will iterate over all of the previous elements of the outer loop's current point. The inner loop will check to see if there is a greater element, and if so, the loop will be ended.

```
function stockSpanProblem(arrs) {
    const result_array = [];
      for (var start = 0; start < arrs.length;
start++) {
        var flag = true;
        var count = 0;
        for (var j = start; j >= 0; j--) {
            if (arrs[j] <= arrs[start]) {
                count++;
            }
            else {
                flag = false;
```

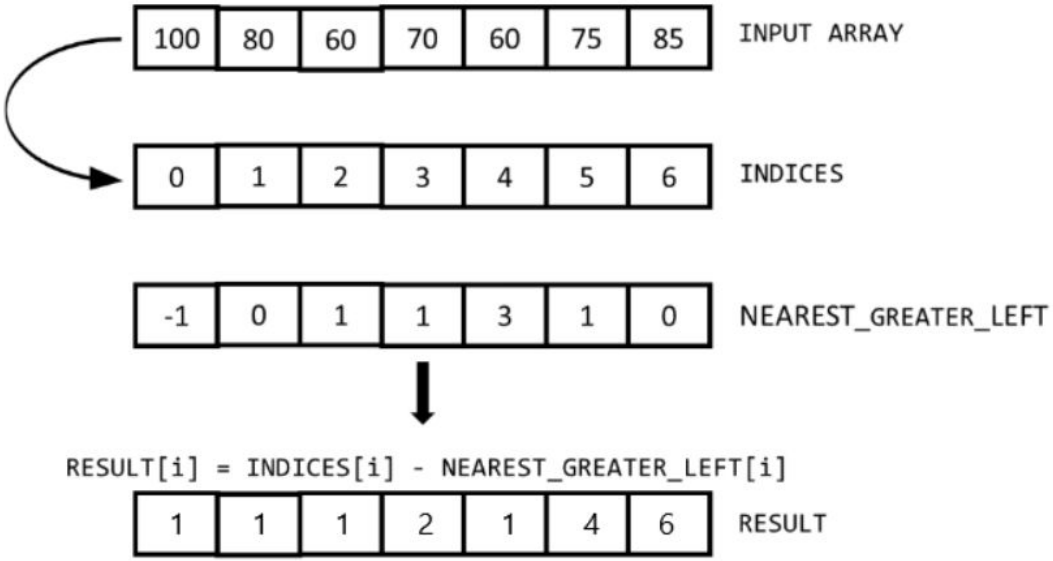
```
Output : [1, 1, 1, 2, 1, 4, 6]
```

Time Complexity: O(n^2)

Stack

We will be using the nearest greater element to left nothing but previous greater element

Nevertheless, instead of saving the greater element, we are storing the index of the greater element, and then subtracting the index of the greater element from the index of the present element.


100 80 60 70 60 75 85
INPUT ARRAY
0 1 2 3 4 5 6
INDICES
-1 0 1 1 3 1 0
NEAREST_GREATER_LEFT
RESULT[i] = INDICES[i] - NEAREST_GREATER_LEFT[i]
1 1 1 2 1 4 6
RESULT

```
class Stack_a {
    constructor() {
        this.stack_a = [];
    }

    isEmpty() {
        return this.stack_a.length === 0;
    }

    push(nums) {
        this.stack_a.push(nums);
    }

    pop() {
        if (this.isEmpty())
            throw 'Stack underflow';
        return this.stack_a.pop();
    }
```

```
}

function stockSpanProblem(arrs) {
    const stack = new Stack_a();
    const result_array = [];

      for (let start = 0; start <
arrs.length; start++) {
        if (stack.isEmpty()) {
            result_array.push(-1);
             stack.push([arrs[start],
start])
         } else if (!stack.isEmpty())
{
              while (!stack.isEmpty()
&& arrs[start] > stack.peek()[0]) {
                  stack.pop();
            }
            if (stack.isEmpty())
```

```
result_array.push(stack.peek()[1]);
                    stack.push([arrs[start],
[start]]);
        }
    }

    const output = [];
        for (let start = 0; start <
result_array.length; start++)
                        output.push(start -
result_array[start]);

    return output;
}

const data = [100, 80, 60, 70, 60, 75, 85];
console.log(stockSpanProblem(data));
```

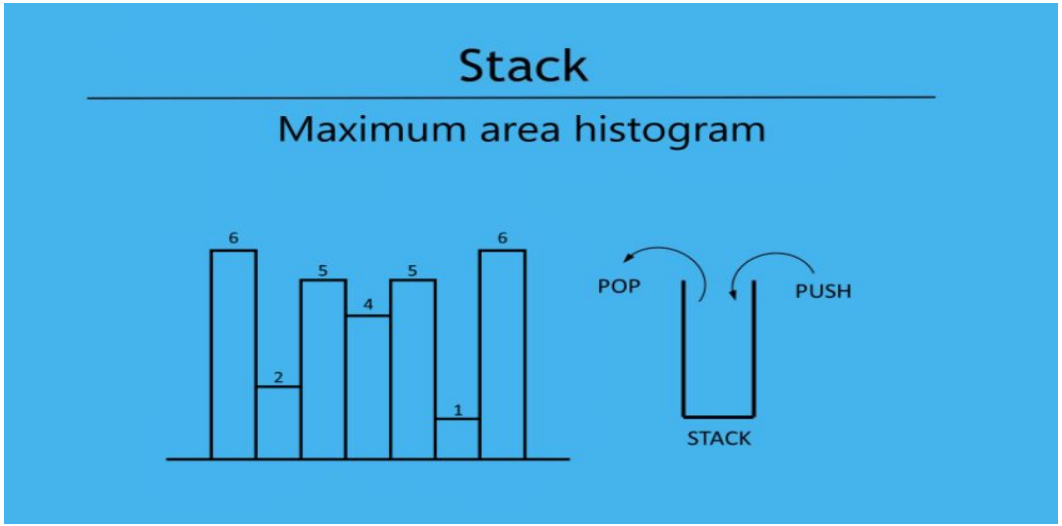
```
Output: [1, 1, 1, 2, 1, 4, 6]
```

Time Complexity: O(n)

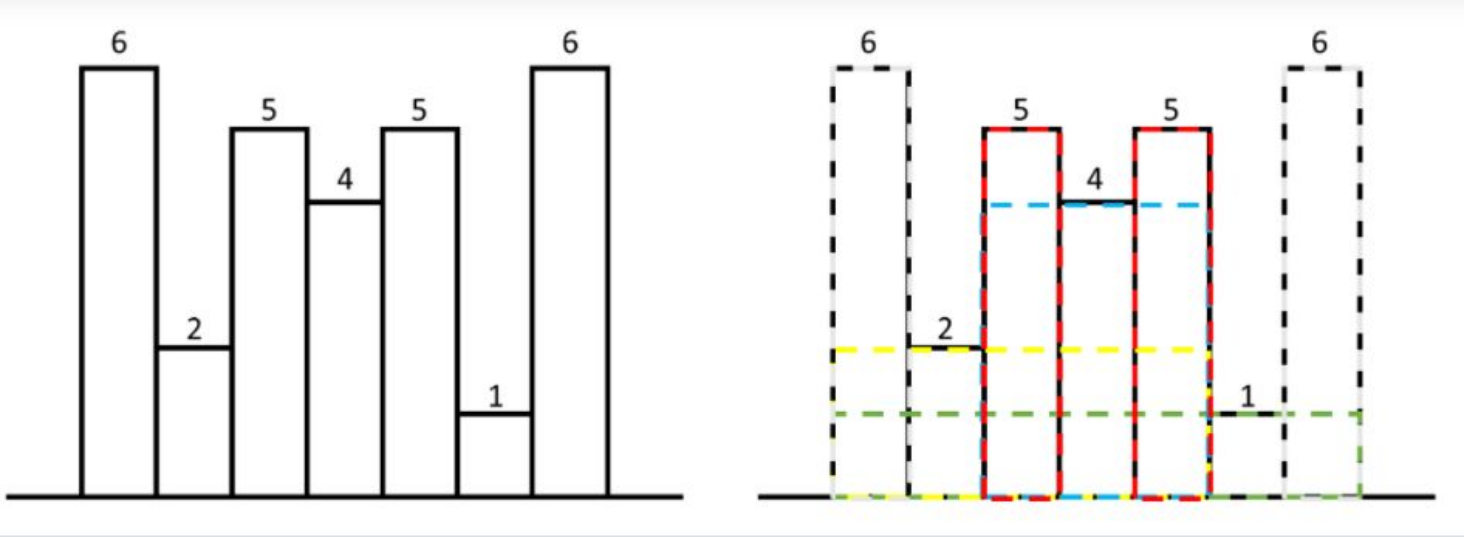
4.Maximum area histogram

In a given histogram, find the greatest rectangular area that can be made up of a number of contiguous bars. Assume that all bars are the same width and have a width of one unit.

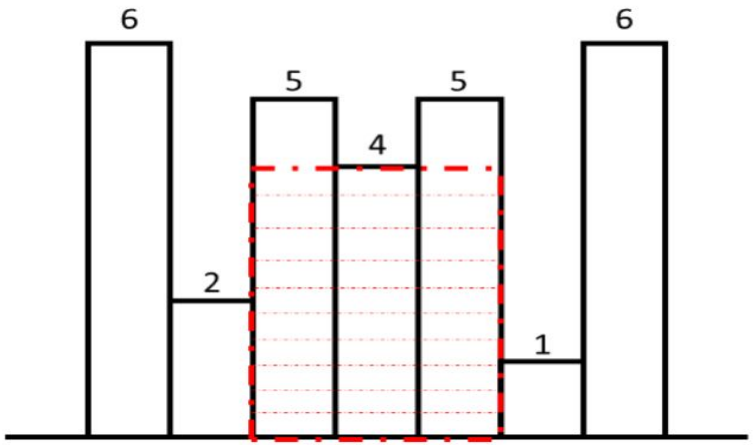
Find the area of the greatest rectangle in the histogram given n non-negative numbers indicating the histogram's bar height and the width of each bar being 1.



For Example, Area can be estimated over a large number of bars in this case..



The maximum area, however, is under the highlighted bars, i.e. 4 * 3 = 12.



Example -

Input: [2, 1, 5, 6, 2, 3]
Output: 10

Input: [6, 2, 5, 4, 5, 1, 6]
Output: 12

There are two approaches for this

- Brute Force
- Stacks

Brute-force

We shall use nested loops in this method. The outer loop will run over all of the array items, while the inner loops will iterate over all of the next and previous elements of the presently pointed-to element in the outer loop. Inner loops will be checked, smaller elements on the left and right will be checked, and the area of the rectangle may then be calculated and compared.

Please find the code here

```
function maxAreaHistogramProblem(arrs) {
    let maxAreas = 0;

    for (let i = 0; i < arrs.length; i++) {
        for (let j = i; j < arrs.length; j++) {
            minHeights = Math.min(arrs[i], arrs[j]);
            for (let k = i; k < j; k++) {
                minHeights = Math.min(minHeights, arrs[k]);
            }
            maxAreas = Math.max(maxAreas, minHeights * ((j - i) + 1));
        }
    }
    return maxAreas;
}

data = [6, 2, 5, 4, 5, 1, 6];
console.log(maxAreaHistogramProblem(data));
```

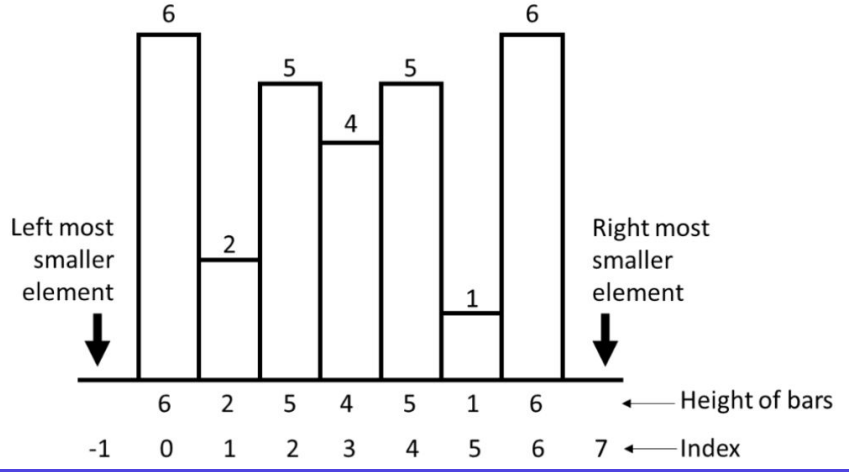
Output :12

Stack

If we look at the problem, we can see that for each bar, we must check the index of the nearest smaller to the left and nearest smaller to the right because if we can find these two parameters, we can get the width of the area under which the maximum area resides by simply subtracting these parameters.

This can be solved using Nearest smaller element to right and left

For example



- To simplify our computation, assume that the leftmost smallest element is at index -1 and the rightmost smallest number is at index equal to the length of the array (here 7), then we can merely remove the width and get the proper value.
- For index 3 with a height of 4, the index of the nearest smaller to the left is 1 and the index of the nearest smaller to the right is 5.
- Subtract the nearest smaller to the left from the nearest smaller to the right, plus one, because we are taking two extra bars from each sides, i.e., 5 - 1 - 1 = 3.
- since we assumed that the width of each bar is one unit, 3 will be the width of total bars or the area that can be covered by the index 3 bar with height 4.
- We can now calculate the area beneath the span of this bar, which is 4 * 3 = 12.

```
WIDTH = NEAREST SMALLER TO RIGHT - NEAREST SMALLER TO LEFT - 1
```

```
class Stack_a {
    constructor() {
        this.stack_a = [];
    }

    isEmpty() {
        return this.stack_a.length === 0;
    }

    push(nums) {
        this.stack_a.push(nums);
    }

    pop() {
        if (this.isEmpty())
            throw 'Stack underflow';
        return this.stack_a.pop();
    }
    peek() {
        if (this.isEmpty())
            return null;
        return
this.stack_a[this.stack_a.length -
1];
    }
}

function
nearestSmallerToRightProblem(arrs) {
    const stack = new Stack_a();
    const result_array = [];

    for (let i = arrs.length; i >= 0;
i--) {
        if (stack.isEmpty()) {
```

```
result_array.push(arrs.length);
            stack.push([arrs[i], i]);
        } else if (!stack.isEmpty()) {
            while (!stack.isEmpty() &&
arrs[i] <= stack.peek()[0]) {
                stack.pop();
            }
            if (stack.isEmpty()) {

result_array.push(arr.length);
            } else {

result_array.push(stack.peek()[1]);
            }
            stack.push([arrs[i], i]);
        }
    }
```

```
    result_array.reverse();
    return result_array;
}

function
nearestSmallerToLeftProblem(arrs) {
    const stack = new Stack_a();
    const result_array = [];

    for (let i = 0; i < arrs.length;
i++) {
        if (stack.isEmpty()) {
            result_array.push(-1);
            stack.push([arrs[i], i]);
        } else if (!stack.isEmpty())
{
```

```
            while (!stack.isEmpty() &&
arrs[i] <= stack.peek()[0]) {
                stack.pop();
            }
            if (stack.isEmpty()) {
                result_array.push(-1);
            } else {

result_array.push(stack.peek()[1]);
            }
            stack.push([arrs[i], i]);
        }
    }

    return result_array;
}
```

```
function
max_area_histogramProblem(arrs) {
    const NSLs =
nearestSmallerToLeftProblem(arrs);
    const NSRs =
nearestSmallerToRightProblem(arrs);

    const WIDTHs = [];

    for (let i = 0; i < arrs.length;
i++)
        WIDTHs.push(NSRs[i] - NSLs[i]
- 1);

    const AREAs = []
```

```
        AREAs.push(arrs[i] * WIDTHs[i]);

    return Math.max(...AREAs);
}

const data = [6, 2, 5, 4, 5, 1, 6];

console.log(max_area_histogramProblem(dat
a));
```

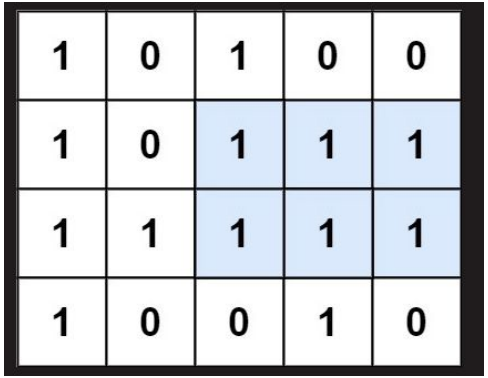
```
Output: 12
```

Time Complexity: O(n)

Space Complexity: O(n)

5. Maximal Rectangle

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




```
Input: matrix = [["1","0","1","0","0"],["1","0","1","1","1"],
["1","1","1","1","1"],["1","0","0","1","0"]]
Output: 6
Explanation: The maximal rectangle is shown in the above picture.
```

If the height of bars of the histogram is given then the largest area of the histogram can be found. This way in each row, the largest area of bars of the histogram can be found. To get the largest rectangle full of 1's, update the next row with the previous row and find the largest area under the histogram, i.e. consider each 1's as filled squares and 0's with an empty square and consider each row as the base.

```
Input :
0 1 1 0
1 1 1 1
1 1 1 1
1 1 0 0
Step 1:
0 1 1 0  maximum area  - 2
Step 2:
row 1  1 2 2 1  area - 4, maximum area becomes 4
row 2  2 3 3 2  area - 8, maximum area becomes 8
row 3  3 4 0 0  area - 6, maximum area remains 8
```

Algorithm:

1. Run a loop to traverse through the rows.
2. Now If the current row is not the first row then update the row as follows, if matrix[i][j] is not zero then matrix[i][j] = matrix[i-1][j] + matrix[i][j].
3. Find the maximum rectangular area under the histogram, consider the ith row as heights of bars of a histogram.
4. Do the previous two steps for all rows and print the maximum area of all the rows.

```
function maxHist(R, C, row)
{

      let result = [];

      let top_val; // Top of stack

      let max_area = 0; // Initialize max area in current row (or histogram)

      let area = 0; // Initialize area with current top

      // Run through all bars of given histogram (or row)
      let i = 0;
      while (i < C) {
            // If this bar is higher than the
            // bar on top stack, push it to stack
            if (result.length == 0
                  || row[result[result.length - 1]] <= row[i]) {
                  result.push(i++);
            }
```

```
            else {

                top_val = row[result[result.length - 1]];
                result.pop();
                area = top_val * i;

                if (result.length > 0) {
                    area = top_val * (i - result[result.length - 1] - 1);
                }
                max_area = Math.max(area, max_area);
            }
        }


        while (result.length > 0) {
            top_val = row[result[result.length - 1]];
            result.pop();
            area = top_val * i;
            if (result.length > 0) {
                area = top_val * (i - result[result.length - 1] - 1);
            }
```

```
                max_area = Math.max(area, max_area);
            }
            return max_area;
        }

        function maxRectangle(R, C, A)
        {

            for (let i = 1; i < R; i++) {
                for (let j = 0; j < C; j++) {

                    // if A[i][j] is 1 then
                    // add A[i -1][j]
                    if (A[i][j] == 1) {
                        A[i][j] += A[i - 1][j];
                    }
                }

                // Update result if area with current row (as last row of rectangle) is
more
```

```
            result = Math.max(result, maxHist(R, C, A[i]));
        }

        return result;
    }

    let R = 4;
    let C = 4;

    let A = [ [ 0, 1, 1, 0 ],
              [ 1, 1, 1, 1 ],
              [ 1, 1, 1, 1 ],
              [ 1, 1, 0, 0 ] ];
    console.log("Area of maximum rectangle is "
                + maxRectangle(R, C, A));
```

```
Output : Area of maximum rectangle is 8
```

MCQs

1. What is the time complexity of the next greater element using stack
   a. O(n^2)
   b. O(nlogn)
   c. O(n)
   d. O(n^3)
2. What is the previous greater element output for this array[4,5,2,25]
   a. [5,25,25,-1]
   b. [25,25,1,-5]
   c. [1,-5,25,5]
   d. None of these
3. What is the next greater element output for this array[10, 4, 2, 20, 40, 12, 30]
   a. [-1, 10, 4, -1, -1, 40, 40]
   b. [40, 10, 4, -1, -1, -1, 40]
   c. [40, 10, -1, 4, -1, -1, 40]
   d. None of these

Practice Problem (10 min)

1. Next Greater Element II
2. Sum of Subarray Ranges

   You are given an integer array `nums`. The range of a subarray of `nums` is the difference between the largest and smallest element in the subarray.

   Return *the sum of all subarray ranges of* `nums`.

   A subarray is a contiguous non-empty sequence of elements within an array.

```
Input: nums = [1,2,3]
Output: 4
Explanation: The 6 subarrays of nums are the following:
[1], range = largest - smallest = 1 - 1 = 0
[2], range = 2 - 2 = 0
[3], range = 3 - 3 = 0
[1,2], range = 2 - 1 = 1
[2,3], range = 3 - 2 = 1
[1,2,3], range = 3 - 1 = 2
So the sum of all ranges is 0 + 0 + 0 + 1 + 1 + 2 = 4.
```

3. Daily Temperatures

Given an array of integers temperatures represents the daily temperatures, return *an array* answer *such that* answer[i] *is the number of days you have to wait after the* ith *day to get a warmer temperature*. If there is no future day for which this is possible, keep answer[i] == 0 instead.

```
Input: temperatures = [73,74,75,71,69,72,76,73]
Output: [1,1,4,2,1,1,0,0]
```

4. What is the space complexity for the next greater element using stack?
    a. O(1)
    b. O(n logn)
    c. O(n)
    d. O(n^2)

5. Which optimal data structure used in solving next/previous greater elements?
    e. Stack
    f. Tree
    g. None of the above

Thank you