Stack_1

@kbbhatt04

@August 31, 2023

Check valid parenthesis

- Approach
  - Optimal
    - If any opening bracket is encountered “(” “{” “[”, push it onto the stack
    - If any closing bracket is encountered, check if stack is non-empty or not
    - Return False if stack is empty or the top element of stack is not opposite pair of closing bracket else move ahead
    - At last if the stack is empty, return True else return False
    - Time Complexity: $O(n)$
    - Space Complexity: $O(n)$

```
# Python3
# Optimal Solution
def isValid(s: str) -> bool:
    st = []
    for it in s:
        if it == '(' or it == '{' or it == '[':
            st.append(it)
        else:
            if len(st) == 0:
                return False
            ch = st[-1]
            st.pop()
            if (it == ')' and ch == '(') or
                            (it == ']' and ch == '[') or
```

```
                                (it == '}' and ch == '{'):
                continue
            else:
                return False
    return len(st) == 0
```

```
// C++
// Optimal Solution
class Solution {
    stack<char> stc;
public:
    bool isValid(string s) {
        int i;
        for(i = 0; i < s.size(); i++)
        {
            if(s[i] == '(' || s[i] == '[' || s[i] == '{')
            {
                stc.push(s[i]);
            }
            else
            {
                if(s[i] == ')' || s[i] == ']' || s[i] == '}')
                {
                    if(stc.empty())
                    {
                        return false;
                    }
                    else
                    {
                        char t = stc.top();
                        if(s[i] == ')' && t != '(')
                        {
                            return false;
                        }
                        else if(s[i] == ']' && t != '[')
```

```
                        {
                            return false;
                        }
                        else if(s[i] == '}' && t != '{')
                        {
                            return false;
                        }
                        else
                        {
                            stc.pop();
                        }
                    }
                }
            }

            if(stc.empty()){
                return true;
            }
            return false;
        }
};
```

Min Stack

- Approach
  - Optimal
    - Maintain pairs as elements in the stack that stores current value and minimum till now
    - Time Complexity: $O(1)$
    - Space Complexity: $O(1)$

```
# Python3
# Optimal Solution
```

```
class MinStack:

    def __init__(self):
        self.stack = []


    def push(self, val: int) -> None:
        if not self.stack:
            self.stack += (val, val),
        else:
            self.stack += (val, min(val, self.stack[-1][1])),


    def pop(self) -> None:
        if self.stack:
            self.stack.pop()


    def top(self) -> int:
        return self.stack[-1][0]


    def getMin(self) -> int:
        return self.stack[-1][1]
```

```
// C++
// Optimal Solution
class MinStack {
public:
    vector<pair<int, int>> s;
    MinStack() {
    }

    void push(int val) {
        if(s.empty()) {
```

```
            s.push_back({val, val});
        }
        else {
            s.push_back({val, min(val, s.back().second)});
        }
    }

    void pop() {
        s.pop_back();
    }

    int top() {
        return s.back().first;
    }

    int getMin() {
        return s.back().second;
    }
};
```

Next Greater Element - 1

- Approach
  - Brute-force
    - For each element, iterate over the right side of the array and find the next greater element and store it in a hashmap
    - Time Complexity: $O(n^2)$
    - Space Complexity: $O(1)$
  - Optimal
    - We traverse the array from right to left and maintain a stack
    - while stack is not empty and the stack top is less than curr_element, pop from stack

- if stack is not empty, then the stack top is the nge for curr_element else -1
- push curr_element onto stack
- Time Complexity: $O(n)$
- Space Complexity: $O(n)$

```
# Python3
# Brute-force Solution
class Solution:
    def nextGreaterElement(self, nums1, nums2):
        dic = {}
        for i in range(len(nums2)):
            nge = -1
            for j in range(i+1, len(nums2)):
                if nums2[j] > nums2[i]:
                    nge = nums2[j]
                    break
            dic[nums2[i]] = nge

        ans = []
        for i in nums1:
            ans.append(dic[i])
        return ans
```

```
# Python3
# Optimal Solution
class Solution:
    def nextGreaterElement(self, nums1, nums2):
        stack = []
        dic = {}
        for i in range(len(nums2)-1, -1, -1):
            while stack and stack[-1] <= nums2[i]:
                stack.pop()
```

```
            if stack:
                dic[nums2[i]] = stack[-1]
            else:
                dic[nums2[i]] = -1

            stack.append(nums2[i])

        ans = []
        for i in nums1:
            ans.append(dic[i])
        return ans
```

```
// C++
// Optimal Solution
class Solution {
public:
    vector<int> nextGreaterElement(vector<int>& nums1, vector<in
        stack<int> st;
        unordered_map<int,int> map;
        for(int i=nums2.size()-1;i>=0;i--) {
            while(!st.empty() && st.top()<=nums2[i])
                st.pop();
            if (st.empty()) {
                map[nums2[i]] = -1;
            }
            else {
                map[nums2[i]] = st.top();
            }
            st.push(nums2[i]);
        }
        vector<int> ans;
        for(int i=0;i<nums1.size();i++)
            ans.push_back(map[nums1[i]]);
        return ans;
```

```
    }
};
```

Next Greater Element - 2

Return array of nge's for each element in a circular array

- Approach
  - Brute-force
    - Concatenate the arr nums with itself
    - Using two nested loops, find the nge for each element
    - Time Complexity: $O(n^2)$
    - Space Complexity: $O(2n)$
  - Better
    - Instead of concatenating nums array with itself, we can traverse circularly using modulo operator
    - From each index i+1, we look for next n-1 elements circularly
    - Time Complexity: $O(n^2)$
    - Space Complexity: $O(1)$
  - Optimal
    - We traverse the array from right to left with twice the size of original array and maintain a stack
    - while stack is not empty and the stack top is less than curr_element, pop from stack
    - if i < arr.size() and if stack is not empty, then the stack top is the nge for curr_element else -1
    - push curr_element onto stack
    - Time Complexity: $O(n)$
    - Space Complexity: $O(n)$

```
# Python3
# Brute-force Solution
class Solution:
    def nextGreaterElements(self, nums: List[int]) -> List[int]
        arr = nums+nums
        ans = []
        for i in range(len(nums)):
            nge = -1
            for j in range(i+1, len(arr)):
                if arr[j] > arr[i]:
                    nge = arr[j]
                    break
            ans.append(nge)
        return ans
```

```
# Python3
# Better Solution
class Solution:
    def nextGreaterElements(self, nums: List[int]) -> List[int]
        n = len(nums)
        ans = []
        for i in range(len(nums)):
            nge = -1
            for j in range(1, n):
                if nums[(i + j) % n] > nums[i]:
                    nge = nums[(i + j) % n]
                    break
            ans.append(nge)
        return ans
```

```
# Python3
# Optimal Solution
class Solution:
    def nextGreaterElements(self, nums: List[int]) -> List[int]
```

```
        n = len(nums)
        stack = []
        ans = [-1]*n

        for i in range(2*n-1, -1, -1):
            while stack and stack[-1] <= nums[i%n]:
                stack.pop()

            if i < n:
                if stack:
                    ans[i] = stack[-1]
            stack.append(nums[i%n])

        return ans
```

```
// C++
// Optimal Solution
class Solution {
  public:
    vector<int> nextGreaterElements(vector<int>& nums) {
      int n = nums.size();
      vector<int> nge(n, -1);
      stack<int> st;
      for (int i = 2 * n - 1; i >= 0; i--) {
        while (!st.empty() && st.top() <= nums[i % n]) {
          st.pop();
        }

        if (i < n) {
          if (!st.empty()) nge[i] = st.top();
        }
        st.push(nums[i % n]);
      }
      return nge;
```

```
    }
};
```

Nearest Smaller Element to the Left

- Approach
  - Brute-force
    - A naive solution would be to take a nested loop, and for every element keep iterating back till we find a smaller element
    - Time Complexity: $O(n^2)$
    - Space Complexity: $O(1)$
  - Optimal
    - We traverse the array and maintain a stack
    - while stack is not empty and the stack top is greater than curr_element, pop from stack
    - if stack is not empty, then the stack top is the nse for curr_element else -1
    - push curr_element onto stack
    - Time Complexity: $O(n)$
    - Space Complexity: $O(n)$

```
# Python3
# Brute-force Solution
class Solution:
    def prevSmaller(self, A):
        ans = []

        for i in range(len(A)):
            nse = -1
            for j in range(i-1, -1, -1):
                if A[j] < A[i]:
```

```
                    nse = A[j]
                    break
            ans.append(nse)

        return ans
```

```
# Python3
# Optimal Solution
class Solution:
    def prevSmaller(self, A):
        stack = [A[0]]
        ans = [-1]

        for i in range(1, len(A)):
            while stack and stack[-1] >= A[i]:
                stack.pop()

            if stack:
                ans.append(stack[-1])
            else:
                ans.append(-1)

            stack.append(A[i])
        return ans
```

```
// C++
// Optimal Solution
vector<int> Solution::prevSmaller(vector<int> &A) {
    vector<int> ans;
    ans.resize(A.size());

    stack<int> st;

    for (int i = 0; i < A.size(); i++) {
```

```
        while (!st.empty() && st.top() >= A[i]) st.pop();
        if (st.empty()) {
            ans[i] = -1;
        } else {
            ans[i] = st.top();
        }
        st.push(A[i]);
    }
    return ans;
}
```

Sum of Subarray Minimums

- Approach
  - Brute-force
    - Generate all subarrays
    - Traverse all subarrays and take minimum from each
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(n^2)$
  - Better
    - Take minimum while generating all subarrays
    - Time Complexity: $O(n^2)$
    - Space Complexity: $O(1)$
  - Optimal
    - For each element, we find the index on its left and right up to which the element itself will be minimum and maintain theirs indexes in two separate arrays
    - We do this by finding Previous Smaller Element (PSE) and Next Smaller Element (NSE) using monotonic stack

- To handle duplicate elements: Set **strict less** and **non-strict less**(less than **or equal to**) for finding **NLE** and **PLE** respectively. The order doesn't matter.
    - Let's Take Example : `[1,5,6,5,4]`
    - Now lets fill the `left[ ]` and `right[ ]` array for the index 2 and 4 which is `arr[4]=arr[2]=5` //indexing starting from 1
    - We will do `non-strict less` condition for both left and right array
    - then `left[4]=2` //which is {5,6}
    - `right[4]=0` //no element right of index 4 is greater or equal to arr[4]=5
    - so possible subarray in this case which can be formed are - `{5,6,5},{6,5},{5}` //(left[4]+1)*(right[4]+1) elements
    - Now we will calculate `left[2]` and `right[2]`
    - `left[2]=0` //no element left of index 2 is greater or equal to arr[2]=5
    - `right[2]=2` //which is {6,5}
    - so possible subarray in this case which can be formed are - `{5}, {5,6},{5,6,5}`
    - Thus, you can see we are adding 2 times same set of sub array which
      is {5,6,5} which is extra and hence we have to do 1 Strict-less searching and 1 non-strict less
- Formula to find number of subarrays in which the element is minimum (num_subarrays) = `(idx_of_cur_elm - index_of_PSE) * (index_of_NSE - idx_of_cur_elm)`
- And each element will contribute `element * num_subarrays` to the final sum
- Time Complexity: $O(n^3)$
- Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
```

```
class Solution:
    def sumSubarrayMins(self, arr: List[int]) -> int:
        mod = 10**9 + 7
        n = len(arr)
        sub_arrays = []

        for i in range(n):
            for j in range(i, n):
                temp = []
                for k in range(i, j+1):
                    temp += arr[k],
                sub_arrays += temp,

        sum_ = 0
        for i in sub_arrays:
            sum_ += min(i)
        return sum_
```

```
# Python3
# Better Solution
class Solution:
    def sumSubarrayMins(self, arr: List[int]) -> int:
        mod = 10**9 + 7
        n = len(arr)
        sum_ = 0

        for i in range(n):
            mini = arr[i]
            for j in range(i, n):
                mini = min(mini, arr[j])
                sum_ += mini
        return sum_ % mod
```

```
# Python3
# Optimal Solution
class Solution:
    def sumSubarrayMins(self, arr: List[int]) -> int:
        n = len(arr)
        st1, st2 = [], []
        nse, pse = [0 for _ in range(n)], [0 for _ in range(n)]

        for i in range(n):
            nse[i] = n-i-1
            pse[i] = i

        for i in range(n):
            while st1 and arr[st1[-1]] > arr[i]:
                nse[st1[-1]] = i - st1[-1] - 1
                st1.pop()
            st1.append(i)

                # notice ">="
        for i in range(n-1, -1, -1):
            while st2 and arr[st2[-1]] >= arr[i]:
                pse[st2[-1]] = st2[-1] - i - 1
                st2.pop()
            st2.append(i)

        ans = 0
        mod = 10**9 + 7
        for i in range(n):
            ans += (arr[i] * (nse[i] + 1) * (pse[i] + 1))
            ans %= mod

        return ans
```

```
// C++
// Optimal Solution
int sumSubarrayMins(vector<int>& arr) {
        int n = arr.size();
        vector<int> left(n,-1), right(n,n);
        // for every i find the Next smaller element to left and

        // Left
        stack<int> st;
        for(int i=0; i<n; i++)
        {
            while(st.size() && arr[i] < arr[st.top()]) st.pop()
            if(st.size()) left[i] = i - st.top();
            else left[i] = i+1;
            st.push(i);
        }

        while(st.size()) st.pop();

        // Right
        for(int i=n-1; i>=0; i--)
        {
                        // notice "<="
            while(st.size() && arr[i] <= arr[st.top()]) st.pop()
            if(st.size()) right[i] = st.top() - i;
            else right[i] = n - i;
            st.push(i);
        }

        // for each i, contribution is (Left * Right) * Element
        int res = 0;
        for(int i=0; i<n; i++)
        {
            long long prod = (left[i]*right[i])%MOD;
            prod = (prod*arr[i])%MOD;
```

```
            res = (res + prod)%MOD;
        }
        return res%MOD;
    }
```

Asteroid Collision

- Approach
  - Optimal
    - Apply concept of relative velocity s.t. positive values remain fixed and only negative values move
    - Whenever we encounter a positive value, we will simply push it to the stack (or negative asteroid onto another negative asteroid e.g. [-2,-1,1,2])
    - The moment we encounter a negative value, we know some or all or zero positive values will be knocked out of the stack
    - We have to take care of the case when `s.top() == asteroids[i]`. In this case one positive element will pop out and negative asteroid won't enter the stack.
    - Time Complexity: $O(n)$
    - Space Complexity: $O(n)$ if using stack explicitly along side ans vector else $O(1)$

```
# Python3
# Optimal Solution
class Solution(object):
    def asteroidCollision(self, asteroids):
        res = []
        for asteroid in asteroids:
            while len(res) and asteroid < 0 and res[-1] > 0:
                if res[-1] == -asteroid:
                    res.pop()
                    break
```

```
                elif res[-1] < -asteroid:
                    res.pop()
                    continue
                elif res[-1] > -asteroid:
                    break
            else:
                res.append(asteroid)
        return res
```

```
// C++
// Optimal Solution
class Solution {
public:
    vector<int> asteroidCollision(vector<int>& asteroids) {
        vector<int> st;
        int n = asteroids.size();

        for (int i = 0; i < n; i++) {
            bool destroyed = false;
            while (st.size() != 0 && asteroids[i] < 0 && st.back
                // same size
                if (-asteroids[i] == st.back()) {
                    destroyed = true;
                    st.pop_back();
                    break;
                }
                // -(-5) > 4
                else if (-asteroids[i] > st.back()) {
                    st.pop_back();
                    continue;
                }
                // -(-3) < 4
                else {
                    destroyed = true;
                    break;
```

```
                }
            }
            if (!destroyed) {st.push_back(asteroids[i]);}
        }
        return st;
    }
};
```

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.

- Approach
  - Brute-force
    - Generate all subarrays
    - Traverse all subarrays and take difference of maximum and minimum from each subarray and add it to total
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$
  - Better
    - Take difference of maximum and minimum from each subarray and add it to total while traversing each element
    - Time Complexity: $O(n^2)$
    - Space Complexity: $O(1)$
  - Optimal
    - Same logic as Sum of Subarray Minimum
    - Using monotonic stacks, we find nse, pse and nge, pge
    - Time Complexity: $O(n)$
    - Space Complexity: $O(n)$

```python
# Python3
# Brute-force Solution
class Solution:
    def subArrayRanges(self, nums: List[int]) -> int:
        n = len(nums)
        sum = 0
        for i in range(n):
            for j in range(i, n):
                mini = float("inf")
                maxi = float("-inf")
                for k in range(i, j+1):
                    mini = min(mini, nums[k])
                    maxi = max(maxi, nums[k])
                sum += (maxi - mini)
        return sum
```

```python
# Python3
# Better Solution
class Solution:
    def subArrayRanges(self, nums: List[int]) -> int:
        n = len(nums)
        sum = 0
        for i in range(n):
            mini = nums[i]
            maxi = nums[i]
            for j in range(i, n):
                mini = min(mini, nums[j])
                maxi = max(maxi, nums[j])
                sum += (maxi - mini)
        return sum
```

```python
# Python3
# Optimal Solution
class Solution:
```

```
    def subArrayRanges(self, nums: List[int]) -> int:
        n = len(nums)

        ## minimum sum
        left, right = [0 for _ in range(n)], [0 for _ in range(n
        st_l, st_r = [], []

        # left
        for i in range(n):
            while st_l and nums[i] < nums[st_l[-1]]:
                st_l.pop()
            if st_l:
                left[i] = i - st_l[-1]
            else:
                left[i] = i + 1
            st_l.append(i)

        # right
        for i in range(n-1, -1, -1):
            while st_r and nums[i] <= nums[st_r[-1]]:
                st_r.pop()
            if st_r:
                right[i] = st_r[-1] - i
            else:
                right[i] = n - i
            st_r.append(i)

        min_sum = 0
        for num, i, j in zip(nums, left, right):
            min_sum += num * i * j

        ## maximum sum
        left, right = [0 for _ in range(n)], [0 for _ in range(n
        st_l, st_r = [], []

        # left
```

```
        for i in range(n):
            while st_l and nums[i] > nums[st_l[-1]]:
                st_l.pop()
            if st_l:
                left[i] = i - st_l[-1]
            else:
                left[i] = i + 1
            st_l.append(i)

        # right
        for i in range(n-1, -1, -1):
            while st_r and nums[i] >= nums[st_r[-1]]:
                st_r.pop()
            if st_r:
                right[i] = st_r[-1] - i
            else:
                right[i] = n - i
            st_r.append(i)

        max_sum = 0
        for num, i, j in zip(nums, left, right):
            max_sum += num * i * j

        return max_sum - min_sum
```

```
// C++
// Optimal Solution
class Solution {
public:
    long long subArrayRanges(vector<int>& nums) {
        int n = nums.size();

        // minimum sum
        vector<int> left(n, 0), right(n, 0);
        stack<int> l, r;
```

```
        // left
        for (int i = 0; i < n; i++) {
            while (l.size() && nums[i] < nums[l.top()]) {
                l.pop();
            }
            if (l.size()) {
                left[i] = i - l.top();
            }
            else {
                left[i] = i + 1;
            }
            l.push(i);
        }

        // right
        for (int i = n-1; i >= 0; i--) {
            while (r.size() && nums[i] <= nums[r.top()]) {
                r.pop();
            }
            if (r.size()) {
                right[i] = r.top() - i;
            }
            else {
                right[i] = n - i;
            }
            r.push(i);
        }

        long long min_sum = 0;
        for (int i = 0; i < n; i++) {
            min_sum += long(long(nums[i]) * long(left[i]) * long
        }

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

```
            left[i] = 0;
            right[i] = 0;
        }
        while (l.size()) {l.pop();}
        while (r.size()) {r.pop();}

        // left
        for (int i = 0; i < n; i++) {
            while (l.size() && nums[i] > nums[l.top()]) {
                l.pop();
            }
            if (l.size()) {
                left[i] = i - l.top();
            }
            else {
                left[i] = i + 1;
            }
            l.push(i);
        }

        // right
        for (int i = n-1; i >= 0; i--) {
            while (r.size() && nums[i] >= nums[r.top()]) {
                r.pop();
            }
            if (r.size()) {
                right[i] = r.top() - i;
            }
            else {
                right[i] = n - i;
            }
            r.push(i);
        }

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

```
            max_sum += long(long(nums[i]) * long(left[i]) * long
        }

        return max_sum - min_sum;
    }
};
```

Remove K digits

Given string num representing a non-negative integer `num`, and an integer `k`, return *the smallest possible integer after removing* `k` *digits from* `num`

- Approach
  - Optimal
    - If we are given a set of digits from which we have to create a number, then the smallest possible number that can be formed is a number of increasing sequence (e.g. 5324 → 2345)
    - Time Complexity: $O(n)$
    - Space Complexity: $O(n)$

```
# Python3
# Optimal Solution
class Solution:
    def removeKdigits(self, num: str, k: int) -> str:
        if len(num) <= k:   return "0"
        if k == 0:  return num

        stack = []
        for i in range(len(num)):
            while k and stack and num[i] < stack[-1]:
                stack.pop()
                k -= 1
            stack.append(num[i])

                # if still k > 0, then remove larger digits fro
```

```
        while k and stack:
            stack.pop()
            k -= 1

        if len(stack) == 1 and stack[0] == "0": return "0"

                # removing 0s from front
        while stack:
            if stack[0] == "0":
                stack.pop(0)
            else:
                break

        if not stack:   return "0"
        return "".join(stack)
```

```
// C++
// Optimal Solution
class Solution {
public:
    string removeKdigits(string num, int k) {
        if (num.size() <= k) return "0";
        if (k == 0) return num;

        stack<char> st;
        for (int i = 0; i < num.size(); i++) {
            while (k && st.size() && num[i] < st.top())
            {
                st.pop();
                k--;
            }
            st.push(num[i]);
        }

        while (k && st.size()) {
```

```
            st.pop();
            k--;
        }

        if (st.size() == 1 && st.top() == '0') return "0";

        stack<char> temp;
        while (!st.empty()) {
            temp.push(st.top());
            st.pop();
        }

        while (!temp.empty() && temp.top() == '0') temp.pop();

        if (temp.empty())   return "0";
        string ans = "";
        while (!temp.empty()) {
            ans += temp.top();
            temp.pop();
        }
        return ans;
    }
};
```

Largest Rectangle in Histogram

- Approach
  - Brute-force
    - For each element, loop over left and right side of element to find index of its pse and nse
    - Time Complexity: $O(n^2)$
    - Space Complexity: $O(1)$
  - Better

        - Find nse and pse indices for every element using stack (with 2 separate loops)
        - Time Complexity: $O(n)$
        - Space Complexity: $O(n)$
    - Optimal
        - https://takeuforward.org/data-structure/area-of-largest-rectangle-in-histogram/
        - Time Complexity: $O(n)$
        - Space Complexity: $O(n)$

```
# Python3
# Brute-force Solution
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        n = len(heights)
        ans = 0

        for i in range(n):
            pse = i
            nse = i

            # pse
            for j in range(i-1, -1, -1):
                if heights[j] >= heights[i]:
                    pse = j
                else:
                    break

            # nse
            for j in range(i+1, n):
                if heights[j] >= heights[i]:
                    nse = j
                else:
```

```
                    break

            ans = max(ans, heights[i] * (nse - pse + 1))
        return ans
```

```
# Python3
# Better Solution
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        n = len(heights)
        pse, nse = [0 for _ in range(n)], [n-1 for _ in range(n)
        stack = []

        # pse
        for i in range(n):
            while stack and heights[i] <= heights[stack[-1]]:
                stack.pop()
            if stack:
                pse[i] = stack[-1]+1
            stack.append(i)

        stack.clear()

        # nse
        for i in range(n-1, -1, -1):
            while stack and heights[i] <= heights[stack[-1]]:
                stack.pop()
            if stack:
                nse[i] = stack[-1]-1
            stack.append(i)

        ans = 0
        for i in range(n):
```

```
            ans = max(ans, (nse[i] - pse[i] + 1) * heights[i])
        return ans
```

```
# Python3
# Optimal Solution
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        n = len(heights)
        ans = 0
        stack = []
        for i in range(n+1):
            while stack and (i==n or heights[stack[-1]] >= heigh
                height = heights[stack.pop()]
                width = i if not stack else i - stack[-1] - 1
                ans = max(ans, width * height)
            stack.append(i)
        return ans
```

```
// C++
// Optimal Solution
class Solution {
  public:
    int largestRectangleArea(vector<int>& histo) {
      stack<int> st;
      int maxA = 0;
      int n = histo.size();
      for (int i = 0; i <= n; i++) {
        while (!st.empty() && (i == n || histo[st.top()] >= hist
          int height = histo[st.top()];
          st.pop();
          int width;
          if (st.empty())
            width = i;
          else
```

```
            width = i - st.top() - 1;
          maxA = max(maxA, width * height);
        }
        st.push(i);
      }
      return maxA;
    }
};
```

Template

- Approach
  - Brute-force
    -
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$
  - Better
    -
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$
  - Optimal
    -
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
```

```
# Python3
# Better Solution
```

```
# Python3
# Optimal Solution
```

```
// C++
// Optimal Solution
```