325. Maximum Size Subarray Sum Equals k

Given an integer array nums and an integer k, return *the maximum length of a subarray that sums to* k. If there isn't one, return 0 instead.

Example 1:

```
Input: nums = [1,-1,5,-2,3], k = 3

Output: 4

Explanation: The subarray [1, -1, 5, -2] sums to 3 and is the longest.
```

Example 2:

```
Input: nums = [-2,-1,2,1], k = 1

Output: 2

Explanation: The subarray [-1, 2] sums to 1 and is the longest.
```

Constraints:

- $1 <= nums.length <= 2 * 10^5$
- $-10^4 <= nums[i] <= 10^4$
- $-10^9 <= k <= 10^9$

Algorithm (Prefix Sum + Hash Map)

1. Initialize three variables:
   - An integer prefixSum that keeps track of the prefix sum of nums as 0.
   - An integer longestSubarray that will keep track of the longest subarray with sum k as 0.
   - A hash map indices that has keys of prefix sums seen so far and values of the first index that each key was seen.
2. Iterate through nums. At each index i, add nums[i] to prefixSum. Then, make the following checks:
   - If prefixSum == k, that means the sum of the array up to this index is equal to k. Update longestSubarray = i + 1 (because i is 0-indexed)
   - If prefixSum - k exists in indices, that means there is a subarray with sum k ending at the current i. The length will be i - indices[prefixSum - k]. If this length is greater than longestSubarray, update longestSubarray.
   - If the current prefixSum does not yet exist in indices, then set indices[prefixSum] = i. Only do this if it does not already exist because we only want the earliest instance of this presum.
3. Return longestSubarray.

```
class Solution:
    def maxSubArrayLen(self, nums: List[int], k: int) -> int:
        prefix_sum = longest_subarray = 0
        indices = {}

        for i, num in enumerate(nums):
            prefix_sum += num

            # Check if all of the numbers seen so far sum to k.
            if prefix_sum == k:
                longest_subarray = i + 1

            # If any subarray seen so far sums to k, then
            # update the length of the longest_subarray.
            if prefix_sum - k in indices:
                longest_subarray = max(longest_subarray, i - indices[prefix_sum - k])

            # Only add the current prefix_sum index pair to the
            # map if the prefix_sum is not already in the map.
            if prefix_sum not in indices:
                indices[prefix_sum] = i

        return longest_subarray
```

1151. Minimum Swaps to Group All 1's Together

Given a binary array data, return the minimum number of swaps required to group all 1's present in the array together in **any place** in the array.

Example 1:

Input: data = [1,0,1,0,1] **Output:** 1 **Explanation:** There are 3 ways to group all 1's together:[1,1,1,0,0] using 1 swap.[0,1,1,1,0] using 2 swaps.[0,0,1,1,1] using 1 swap.The minimum is 1.

Example 2:

Input: data = [0,0,0,1,0] **Output:** 0 **Explanation:** Since there is only one 1 in the array, no swaps needed.

Example 3:

Input: data = [1,0,1,0,1,0,0,1,1,0,1] **Output:** 3 **Explanation:** One possible solution that uses 3 swaps is [0,0,0,0,0,1,1,1,1,1,1].

Example 4:

Input: data = [1,0,1,0,1,0,1,1,1,0,1,0,0,1,1,1,0,0,1,1,1,0,1,0,1,1,0,0,0,1,1,1,1,0,0,1] **Output:** 8

Constraints:

- $1 <= data.length <= 10^5$
- data[i] is 0 or 1.

```
class Solution: # Sliding Window with Two Pointers
    def minSwaps(self, data: List[int]) -> int:
        ones = sum(data)
        cnt_one = max_one = 0
        left = right = 0
        while right < len(data):
            # updating the number of 1's by adding the new element
            cnt_one += data[right]
            right += 1
            # maintain the length of the window to ones
            if right - left > ones:
                # updating the number of 1's by removing the oldest element
                cnt_one -= data[left]
                left += 1
            # record the maximum number of 1's in the window
            max_one = max(max_one, cnt_one)
        return ones – max_one
```

```
class Solution: # Sliding Window with Deque (Double-ended Queue)
    def minSwaps(self, data: List[int]) -> int:
        ones = sum(data)
        cnt_one = max_one = 0
        # maintain a deque with the size = ones
        deque = collections.deque()
        for i in range(len(data)):
            # we would always add the new element into the deque
            deque.append(data[i])
            cnt_one += data[i]
            # when there are more than ones elements in the deque,
            # remove the leftmost one
            if len(deque) > ones:
                cnt_one -= deque.popleft()
            max_one = max(max_one, cnt_one)
        return ones – max_one
```

1588. Sum of All Odd Length Subarrays

Given an array of positive integers arr, calculate the sum of all possible odd-length subarrays.

A subarray is a contiguous subsequence of the array.

Return *the sum of all odd-length subarrays of* arr.

Example 1:

Input: arr = [1,4,2,5,3]

Output: 58

Explanation: The odd-length subarrays of arr and their sums are:[1] = 1[4] = 4[2] = 2[5] = 5[3] = 3[1,4,2] = 7[4,2,5] = 11[2,5,3] = 10[1,4,2,5,3] = 15If we add all these together we get 1 + 4 + 2 + 5 + 3 + 7 + 11 + 10 + 15 = 58

Example 2:

Input: arr = [1,2]

Output: 3

Explanation: There are only 2 subarrays of odd length, [1] and [2]. Their sum is 3.

Example 3:

Input: arr = [10,11,12]

Output: 66

Constraints:

- 1 <= arr.length <= 100
- 1 <= arr[i] <= 1000

```
class Solution:
    def sumOddLengthSubarrays(self, arr: List[int]) -> int:
        curr_sum = 0
        total_sum = 0
        for l in range(len(arr)):
            for r in range(l, len(arr)):
                curr_sum += arr[r]
                if (r - l + 1) % 2 == 1:
                    total_sum += curr_sum
            curr_sum = 0
        return  total_sum
```

452. Minimum Number of Arrows to Burst Balloons

There are some spherical balloons taped onto a flat wall that represents the XY-plane. The balloons are represented as a 2D integer array points where points[i] = [x_{start}, x_{end}] denotes a balloon whose **horizontal diameter** stretches between x_{start} and x_{end}. You do not know the exact y-coordinates of the balloons.

Arrows can be shot up **directly vertically** (in the positive y-direction) from different points along the x-axis. A balloon with x_{start} and x_{end} is **burst** by an arrow shot at x if $x_{start} <= x <= x_{end}$. There is **no limit** to the number of arrows that can be shot. A shot arrow keeps traveling up infinitely, bursting any balloons in its path.

Given the array points, return *the **minimum** number of arrows that must be shot to burst all balloons*.

Example 1:

Input: points = [[10,16],[2,8],[1,6],[7,12]]

Output: 2

Explanation: The balloons can be burst by 2 arrows:- Shoot an arrow at x = 6, bursting the balloons [2,8] and [1,6].- Shoot an arrow at x = 11, bursting the balloons [10,16] and [7,12].

Example 2:

Input: points = [[1,2],[3,4],[5,6],[7,8]] **Output:** 4

Explanation: One arrow needs to be shot for each balloon for a total of 4 arrows.

Example 3:

Input: points = [[1,2],[2,3],[3,4],[4,5]] **Output:** 2

Explanation: The balloons can be burst by 2 arrows:- Shoot an arrow at x = 2, bursting the balloons [1,2] and [2,3].- Shoot an arrow at x = 4, bursting the balloons [3,4] and [4,5].

Constraints:

- $1 <= points.length <= 10^5$
- points[i].length == 2
- $-2^{31} <= x_{start} < x_{end} <= 2^{31} - 1$

Algorithm (Greedy)

Now the algorithm is straightforward :

- Sort the balloons by end coordinate x_end.
- Initiate the end coordinate of a balloon which ends first : first_end = points[0][1].
- Initiate number of arrows: arrows = 1.
- Iterate over all balloons:
    - If the balloon starts after first_end:
        - Increase the number of arrows by one.
        - Set first_end to be equal to the end of the current balloon.
- Return arrows.

```
class Solution:
    def findMinArrowShots(self, points: List[List[int]]) -> int:
        if not points:
            return 0
        # sort by x_end
        points.sort(key = lambda x : x[1])
        arrows = 1
        first_end = points[0][1]
        for x_start, x_end in points:
            # if the current balloon starts after the end of another one,
            # one needs one more arrow
            if first_end < x_start:
                arrows += 1
                first_end = x_end
        return arrows
```

128. Longest Consecutive Sequence

Given an unsorted array of integers `nums`, return *the length of the longest consecutive elements sequence.*

You must write an algorithm that runs in $O(n)$ time.

Example 1:

Input: nums = [100,4,200,1,3,2]

Output: 4

Explanation: The longest consecutive elements sequence is [1, 2, 3, 4]. Therefore its length is 4.

Example 2:

Input: nums = [0,3,7,2,5,8,4,6,0,1]

Output: 9

Constraints:

- $0 <= nums.length <= 10^5$
- $-10^9 <= nums[i] <= 10^9$

```
class Solution:
    def longestConsecutive(self, nums):
        longest_streak = 0
        num_set = set(nums)

        for num in num_set:
            if num - 1 not in num_set:
                current_num = num
                current_streak = 1

                while current_num + 1 in num_set:
                    current_num += 1
                    current_streak += 1

                longest_streak = max(longest_streak, current_streak)

        return longest_streak
```

454. 4Sum II

Given four integer arrays nums1, nums2, nums3, and nums4 all of length n, return the number of tuples (i, j, k, l) such that:

- $0 <= i, j, k, l < n$
- nums1[i] + nums2[j] + nums3[k] + nums4[l] == 0

Example 1:

Input: nums1 = [1,2], nums2 = [-2,-1], nums3 = [-1,2], nums4 = [0,2]**Output:** 2**Explanation:**The two tuples are:1. (0, 0, 0, 1) -> nums1[0] + nums2[0] + nums3[0] + nums4[1] = 1 + (-2) + (-1) + 2 = 02. (1, 1, 0, 0) -> nums1[1] + nums2[1] + nums3[0] + nums4[0] = 2 + (-1) + (-1) + 0 = 0

Example 2:

Input: nums1 = [0], nums2 = [0], nums3 = [0], nums4 = [0]**Output:** 1

Constraints:

- n == nums1.length
- n == nums2.length
- n == nums3.length
- n == nums4.length
- $1 <= n <= 200$
- $-2^{28} <=$ nums1[i], nums2[i], nums3[i], nums4[i] $<= 2^{28}$

Algorithm

4. For each a in A.
   - For each b in B.
     - If a + b exists in the hashmap m, increment the value.
     - Else add a new key a + b with the value 1.
5. For each c in C.
   - For each d in D.
     - Lookup key -(c + d) in the hashmap m.
     - Add its value to the count cnt.
6. Return the count cnt.

```
class Solution:
  def fourSumCount(self, A: List[int], B: List[int], C: List[int], D: List[int]) -> int:
    cnt = 0
    m = {}
    for a in A:
      for b in B:
        m[a + b] = m.get(a + b, 0) + 1
    for c in C:
      for d in D:
        cnt += m.get(-(c + d), 0)
    return cnt
```

448. Find All Numbers Disappeared in an Array

Given an array nums of n integers where nums[i] is in the range [1, n], return *an array of all the integers in the range* [1, n] *that do not appear in* nums.

Example 1:

Input: nums = [4,3,2,7,8,2,3,1]

Output: [5,6]

Example 2:

Input: nums = [1,1]

Output: [2]

Constraints:

- n == nums.length
- 1 <= n <= 10^5
- 1 <= nums[i] <= n

Follow up: Could you do it without extra space and in O(n) runtime? You may assume the returned list does not count as extra space.

```
class Solution(object):  # using an extra hashmap can produce cleaner code with extra O(n) space
    def findDisappearedNumbers(self, nums: List[int]) -> List[int]:
        # Iterate over each of the elements in the original array
        for i in range(len(nums)):

            # Treat the value as the new index
            new_index = abs(nums[i]) - 1

            # Check the magnitude of value at this new index. If the magnitude is positive, make it negative
            # thus indicating that the number nums[i] has appeared or has been visited.
            if nums[new_index] > 0:
                nums[new_index] *= -1

         result = []  # Response array that would contain the missing numbers

        # Iterate over the numbers from 1 to N and add all those that have positive magnitude in the array
        for i in range(1, len(nums) + 1):
            if nums[i - 1] > 0:
                result.append(i)

        return result
#O(N) time and O(1) space
```

1427. Perform String Shifts

You are given a string s containing lowercase English letters, and a matrix shift, where shift[i] = [$direction_i$, $amount_i$]:

- $direction_i$ can be 0 (for left shift) or 1 (for right shift).
- $amount_i$ is the amount by which string s is to be shifted.
- A left shift by 1 means remove the first character of s and append it to the end.
- Similarly, a right shift by 1 means remove the last character of s and add it to the beginning.

Return the final string after all operations.

Example 1:

Input: s = "abc", shift = [[0,1],[1,2]]
Output: "cab"**Explanation:** [0,1] means shift to left by 1. "abc" -> "bca"[1,2] means shift to right by 2. "bca" -> "cab"

Example 2:

Input: s = "abcdefg", shift = [[1,1],[1,1],[0,2],[1,3]]
Output: "efgabcd"
Explanation: [1,1] means shift to right by 1. "abcdefg" -> "gabcdef"[1,1] means shift to right by 1. "gabcdef" -> "fgabcde"[0,2] means shift to left by 2. "fgabcde" -> "abcdefg"[1,3] means shift to right by 3. "abcdefg" -> "efgabcd"

Constraints:

- 1 <= s.length <= 100
- s only contains lower case English letters.
- 1 <= shift.length <= 100
- shift[i].length == 2
- $direction_i$ is either 0 or 1.
- 0 <= $amount_i$ <= 100

```
class Solution:
  def stringShift(self, string: str, shift: List[List[int]]) -> str:
    for direction, amount in shift:
      amount %= len(string)
      if direction == 0:
        # Move necessary amount of characters from start to end
        string = string[amount:] + string[:amount]
      else:
        # Move necessary amount of characters from end to start
        string = string[-amount:] + string[:-amount]
    return string
```

Let *LL* be the length of the string and *NN* be the length of the shift array.

- Time complexity : *O(N \cdot L)*$O(N \cdot L)$.
- Space complexity : *O(L)*$O(L)$.

409. Longest Palindrome

Given a string s which consists of lowercase or uppercase letters, return *the length of the **longest palindrome*** that can be built with those letters.

Letters are **case sensitive**, for example, "Aa" is not considered a palindrome here.

Example 1:

Input: s = "abccccdd"
Output: 7
Explanation: One longest palindrome that can be built is "dccaccd", whose length is 7.

Example 2:

Input: s = "a"
Output: 1

Example 3:

Input: s = "bb"
Output: 2

Constraints:

- 1 <= s.length <= 2000
- s consists of lowercase **and/or** uppercase English letters only.

Algorithm

For each letter, say it occurs v times. We know we have v // 2 * 2 letters that can be partnered for sure. For example, if we have 'aaaaa', then we could have 'aaaa' partnered, which is 5 // 2 * 2 = 4 letters partnered.

At the end, if there was any v % 2 == 1, then that letter could have been a unique center. Otherwise, every letter was partnered. To perform this check, we will check for v % 2 == 1 and ans % 2 == 0, the latter meaning we haven't yet added a unique center to the answer.

Time O(N), Space O(1)

```
class Solution(object):
    def longestPalindrome(self, s):
        """
        :type s: str
        :rtype: int
        """
        ans = 0
        for v in collections.Counter(s).itervalues():
            ans += v / 2 * 2
            if ans % 2 == 0 and v % 2 == 1:
                ans += 1
        return ans
```

187. Repeated DNA Sequences

The **DNA sequence** is composed of a series of nucleotides abbreviated as 'A', 'C', 'G', and 'T'.

- For example, "ACGAATTCCG" is a **DNA sequence**.

When studying **DNA**, it is useful to identify repeated sequences within the DNA.

Given a string s that represents a **DNA sequence**, return all the **10-letter-long** sequences (substrings) that occur more than once in a DNA molecule. You may return the answer in **any order**.

Example 1:

Input: s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT"
Output: ["AAAAACCCCC","CCCCCAAAAA"]

Example 2:

Input: s = "AAAAAAAAAAAAA"
Output: ["AAAAAAAAAA"]

Constraints:

- $1 <= s.length <= 10^5$
- s[i] is either 'A', 'C', 'G', or 'T'.

```
class Solution:
    def findRepeatedDnaSequences(self, s: str) -> List[str]:
        L, n = 10, len(s)
        seen, output = set(), set()

        # iterate over all sequences of length L
        for start in range(n - L + 1):
            tmp = s[start:start + L]
            if tmp in seen:
                output.add(tmp[:])
            seen.add(tmp)
        return output
```

Can also be solved using Rabin-Karp and Bit-Manipulation approach – all solutions take O(N) for time and space.

5. Longest Palindromic Substring

Given a string s, return *the longest palindromic substring* in s.

Example 1:

Input: s = "babad" **Output:** "bab"
Note: "aba" is also a valid answer.

Example 2:

Input: s = "cbbd" **Output:** "bb"

Example 3:

Input: s = "a" **Output:** "a"

Example 4:

Input: s = "ac" **Output:** "a"

Constraints:

- 1 <= s.length <= 1000 s consist of only digits and English letters.

```
class Solution(object):
    def longestPalindrome(self, s):
        self.maxlen = 0
        self.start = 0

        for i in range(len(s)):
            self.expandFromCenter(s,i,i)
            self.expandFromCenter(s,i,i+1)
        return s[self.start:self.start+self.maxlen]


    def expandFromCenter(self,s,l,r):
        while l > -1 and r < len(s) and s[l] ==s[r]:
            l -= 1
            r += 1

        if self.maxlen < r-l-1:
            self.maxlen = r-l-1
            self.start = l + 1
```

O(N^2) for both time and space complexity

44. Wildcard Matching

Given an input string (s) and a pattern (p), implement wildcard pattern matching with support for '?' and '*' where:

- '?' Matches any single character.
- '*' Matches any sequence of characters (including the empty sequence).

The matching should cover the **entire** input string (not partial).

Example 1:

Input: s = "aa", p = "a"**Output:** false**Explanation:** "a" does not match the entire string "aa".

Example 2:

Input: s = "aa", p = "*"**Output:** true**Explanation:** '*' matches any sequence.

Example 3:

Input: s = "cb", p = "?a"**Output:** false**Explanation:** '?' matches 'c', but the second letter is 'a', which does not match 'b'.

Example 4:

Input: s = "adceb", p = "*a*b"**Output:** true**Explanation:** The first '*' matches the empty sequence, while the second '*' matches the substring "dce".

Example 5:

Input: s = "acdcb", p = "a*c?b"**Output:** false

Constraints:

- 0 <= s.length, p.length <= 2000
- s contains only lowercase English letters.
- p contains only lowercase English letters, '?' or '*'.

```
class Solution: #44
    def isMatch(self, s: str, p: str) -> bool:
        s_len = len(s)
        p_len = len(p)

        # base cases
        if p == s or set(p) == {'*'}:
            return True
        if p == '' or s == '':
            return False

        # init all matrix except [0][0] element as False
        d = [[False] * (s_len + 1) for _ in range(p_len + 1)]
        d[0][0] = True

        for p_idx in range(1, p_len + 1):
            # the current character in the pattern is '*'
            if p[p_idx - 1] == '*':
                s_idx = 1

                # d[p_idx - 1][s_idx - 1] is a string-pattern match on the previous step, i.e. one character before.
                # Find the first idx in string with the previous math.
                while not d[p_idx - 1][s_idx - 1] and s_idx < s_len + 1:
                    s_idx += 1

                # If (string) matches (pattern),
                # when (string) matches (pattern)* as well
                d[p_idx][s_idx - 1] = d[p_idx - 1][s_idx - 1]

                # If (string) matches (pattern),
                # when (string)(whatever_characters) matches (pattern)* as well
                while s_idx < s_len + 1:
                    d[p_idx][s_idx] = True
                    s_idx += 1

            # the current character in the pattern is '?'
            elif p[p_idx - 1] == '?':
                for s_idx in range(1, s_len + 1):
                    d[p_idx][s_idx] = d[p_idx - 1][s_idx - 1]

            # the current character in the pattern is not '*' or '?'
            else:
                for s_idx in range(1, s_len + 1):
                    # Match is possible if there is a previous match and current characters are the same
                    d[p_idx][s_idx] = d[p_idx - 1][s_idx - 1] and p[p_idx - 1] == s[s_idx - 1]

        return d[p_len][s_len]
```

214. Shortest Palindrome

Hard

You are given a string s. You can convert s to a palindrome by adding characters in front of it.

Return *the shortest palindrome you can find by performing this transformation*.

Example 1:

Input: s = "aacecaaa"**Output:** "aaacecaaa"

Example 2:

Input: s = "abcd"**Output:** "dcbabcd"

Constraints:

- $0 <= s.length <= 5 * 10^4$
- s consists of lowercase English letters only.

```
string shortestPalindrome(string s)  // KMP
{
   int n = s.size();
   string rev(s);
   reverse(rev.begin(), rev.end());
   string s_new = s + "#" + rev;
   int n_new = s_new.size();
   vector<int> f(n_new, 0);
   for (int i = 1; i < n_new; i++) {
      int t = f[i - 1];
      while (t > 0 && s_new[i] != s_new[t])
         t = f[t - 1];
      if (s_new[i] == s_new[t])
         ++t;
      f[i] = t;
   }
   return rev.substr(0, n - f[n_new - 1]) + s;
}
```

1634. Add Two Polynomials Represented as Linked Lists

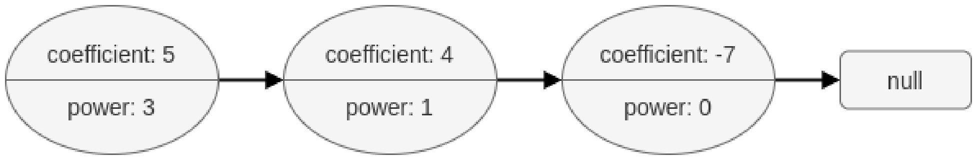
Medium

A polynomial linked list is a special type of linked list where every node represents a term in a polynomial expression.

Each node has three attributes:

- coefficient: an integer representing the number multiplier of the term. The coefficient of the term **9**x^4 is 9.
- power: an integer representing the exponent. The power of the term $9x^4$ is 4.
- next: a pointer to the next node in the list, or null if it is the last node of the list.

For example, the polynomial $5x^3 + 4x - 7$ is represented by the polynomial linked list illustrated below:



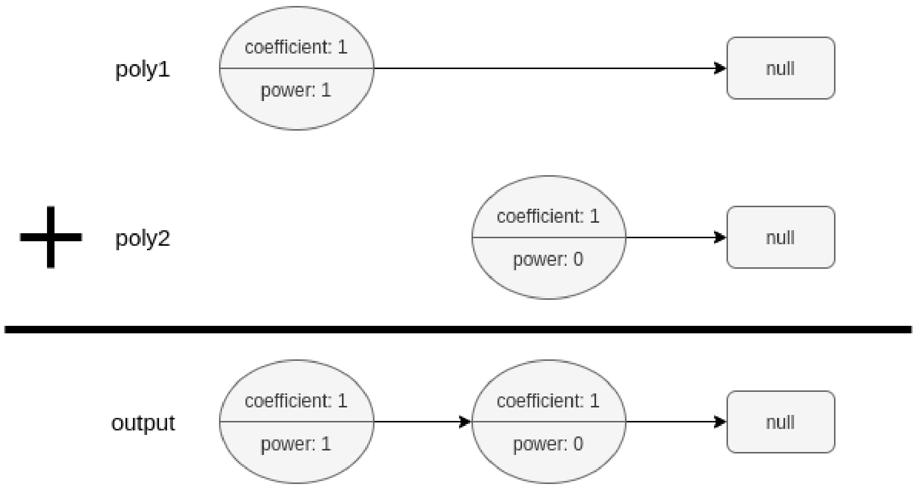
The polynomial linked list must be in its standard form: the polynomial must be in **strictly** descending order by its power value. Also, terms with a coefficient of 0 are omitted.

Given two polynomial linked list heads, poly1 and poly2, add the polynomials together and return *the head of the sum of the polynomials*.

PolyNode format:

The input/output format is as a list of n nodes, where each node is represented as its [coefficient, power]. For example, the polynomial $5x^3 + 4x - 7$ would be represented as: [[5,3],[4,1],[-7,0]].

Example 1:



Input: poly1 = [[1,1]], poly2 = [[1,0]]**Output:** [[1,1],[1,0]]**Explanation:** poly1 = x. poly2 = 1. The sum is x + 1.

Example 2:

Input: poly1 = [[2,2],[4,1],[3,0]], poly2 = [[3,2],[-4,1],[-1,0]]**Output:** [[5,2],[2,0]]**Explanation:** poly1 = $2x^2 + 4x + 3$. poly2 = $3x^2 - 4x - 1$. The sum is $5x^2 + 2$. Notice that we omit the "0x" term.

Example 3:
Input: poly1 = [[1,2]], poly2 = [[-1,2]]**Output:** []**Explanation:** The sum is 0. We return an empty list.

Constraints:

- $0 <= n <= 10^4$
- $-10^9 <= PolyNode.coefficient <= 10^9$
- PolyNode.coefficient != 0
- $0 <= PolyNode.power <= 10^9$
- PolyNode.power > PolyNode.next.power

```
# Definition for polynomial singly-linked list.
# class PolyNode:
#     def __init__(self, x=0, y=0, next=None):
#         self.coefficient = x
#         self.power = y
#         self.next = next
class Solution:
    def addPoly(self, poly1: 'PolyNode', poly2: 'PolyNode') -> 'PolyNode': # O(m+n)
        it1 = poly1
        it2 = poly2
        res = res_head = PolyNode(0, 0)
        while it1 or it2:
            co = 1
            if it1 and not it2:
                res.next = it1
                break
            elif it2 and not it1:
                res.next = it2
                break
            elif it1.power > it2.power:
                res.next = it1
                it1 = it1.next
            elif it1.power == it2.power:
                co = it1.coefficient + it2.coefficient
                if co  != 0:
                    it1.coefficient = co
                    res.next = it1
                it1 = it1.next
                it2 = it2.next
            else:
                res.next = it2
                it2 = it2.next
            if co != 0:
                res = res.next
                res.next = None
        return res_head.next
```

369. Plus One Linked List

Medium

Given a non-negative integer represented as a linked list of digits, *plus one to the integer*.

The digits are stored such that the most significant digit is at the head of the list.

Example 1:

Input: head = [1,2,3]**Output:** [1,2,4]

Example 2:

Input: head = [0]**Output:** [1]

Constraints:

- The number of nodes in the linked list is in the range [1, 100].
- 0 <= Node.val <= 9
- The number represented by the linked list does not contain leading zeros except for the zero itself.

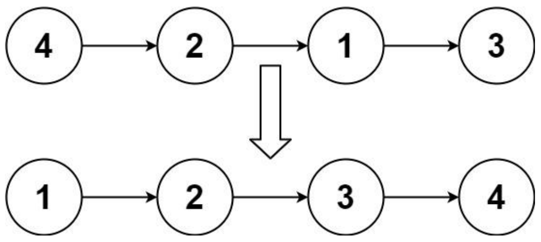
```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def plusOne(self, head: ListNode) -> ListNode: # Time (N), Space O(1)
        # sentinel head
        sentinel = ListNode(0)
        sentinel.next = head
        not_nine = sentinel

        # find the rightmost not-nine digit
        while head:
            if head.val != 9:
                not_nine = head
            head = head.next

        # increase this rightmost not-nine digit by 1
        not_nine.val += 1
        not_nine = not_nine.next

        # set all the following nines to zeros
        while not_nine:
            not_nine.val = 0
            not_nine = not_nine.next

        return sentinel if sentinel.val else sentinel.next
```

148. Sort List

Medium

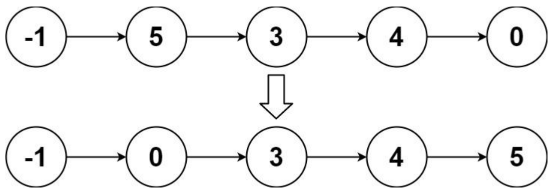
Given the head of a linked list, return *the list after sorting it in* ***ascending order***.

Example 1:



Input: head = [4,2,1,3]**Output:** [1,2,3,4]

Example 2:



Input: head = [-1,5,3,4,0]**Output:** [-1,0,3,4,5]

Example 3:

Input: head = []**Output:** []

Constraints:

- The number of nodes in the list is in the range $[0, 5 * 10^4]$.
- $-10^5 <= Node.val <= 10^5$

Follow up: Can you sort the linked list in $O(n \log n)$ time and $O(1)$ memory (i.e. constant space)?

Top down merge sort O(nlogn) time anda O(logn) space – cab be done in constant space with bottom up merge sort.

```
class Solution {
public:
   ListNode* sortList(ListNode* head) {
      if (!head || !head->next)
         return head;
      ListNode* mid = getMid(head);
      ListNode* left = sortList(head);
      ListNode* right = sortList(mid);
      return merge(left, right);
   }
```

```
    ListNode* merge(ListNode* list1, ListNode* list2) {
        ListNode dummyHead(0);
        ListNode* ptr = &dummyHead;
        while (list1 && list2) {
            if (list1->val < list2->val) {
                ptr->next = list1;
                list1 = list1->next;
            } else {
                ptr->next = list2;
                list2 = list2->next;
            }
            ptr = ptr->next;
        }
        if(list1) ptr->next = list1;
        else ptr->next = list2;

        return dummyHead.next;
    }

    ListNode* getMid(ListNode* head) {
        ListNode* midPrev = nullptr;
        while (head && head->next) {
            midPrev = (midPrev == nullptr) ? head : midPrev->next;
            head = head->next->next;
        }
        ListNode* mid = midPrev->next;
        midPrev->next = nullptr;
        return mid;
    }
};
```

138. Copy List with Random Pointer

Medium

A linked list of length n is given such that each node contains an additional random pointer, which could point to any node in the list, or null.

Construct a deep copy of the list. The deep copy should consist of exactly n **brand new** nodes, where each new node has its value set to the value of its corresponding original node. Both the next and random pointer of the new nodes should point to new nodes in the copied list such that the pointers in the original list and copied list represent the same list state. **None of the pointers in the new list should point to nodes in the original list**.

For example, if there are two nodes X and Y in the original list, where X.random --> Y, then for the corresponding two nodes x and y in the copied list, x.random --> y.

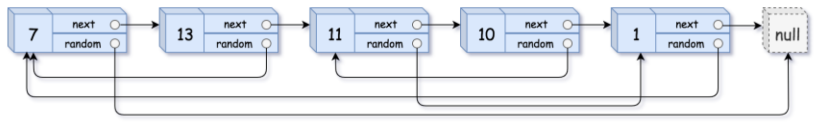
Return *the head of the copied linked list*.

The linked list is represented in the input/output as a list of n nodes. Each node is represented as a pair of [val, random_index] where:

- val: an integer representing Node.val
- random_index: the index of the node (range from 0 to n-1) that the random pointer points to, or null if it does not point to any node.

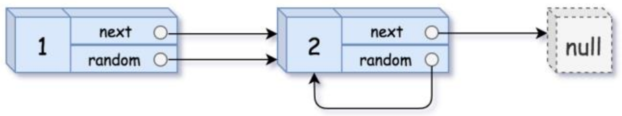
Your code will **only** be given the head of the original linked list.

Example 1:



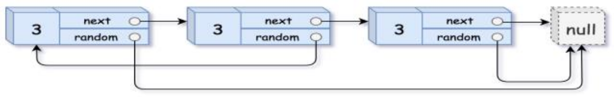
Input: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]**Output:** [[7,null],[13,0],[11,4],[10,2],[1,0]]

Example 2:



Input: head = [[1,1],[2,1]]**Output:** [[1,1],[2,1]]

Example 3:



Input: head = [[3,null],[3,0],[3,null]]**Output:** [[3,null],[3,0],[3,null]]

Example 4:

Input: head = []**Output:** []**Explanation:** The given linked list is empty (null pointer), so return null.

Constraints:

- 0 <= n <= 1000
- -10000 <= Node.val <= 10000
- Node.random is null or is pointing to some node in the linked list.

```
"""
# Definition for a Node.
class Node:
    def __init__(self, x: int, next: 'Node' = None, random: 'Node' = None):
        self.val = int(x)
        self.next = next
        self.random = random
"""

class Solution:
    def __init__(self):
        # Dictionary which holds old nodes as keys and new nodes as its values.
        self.visitedHash = {}

    def copyRandomList(self, head: 'Node') -> 'Node':

        if head == None:
            return None

        # If we have already processed the current node, then we simply return the cloned version of it.
        if head in self.visitedHash:
            return self.visitedHash[head]

        # create a new node
        # with the value same as old node.
        node = Node(head.val, None, None)

        # Save this value in the hash map. This is needed since there might be
        # loops during traversal due to randomness of random pointers and this would help us avoid them.
        self.visitedHash[head] = node

        # Recursively copy the remaining linked list starting once from the next pointer and then from the random pointer.
        # Thus we have two independent recursive calls.
        # Finally we update the next and random pointers for the new node created.
        node.next = self.copyRandomList(head.next)
        node.random = self.copyRandomList(head.random)

        return node
```

O(N) time and space

430. Flatten a Multilevel Doubly Linked List

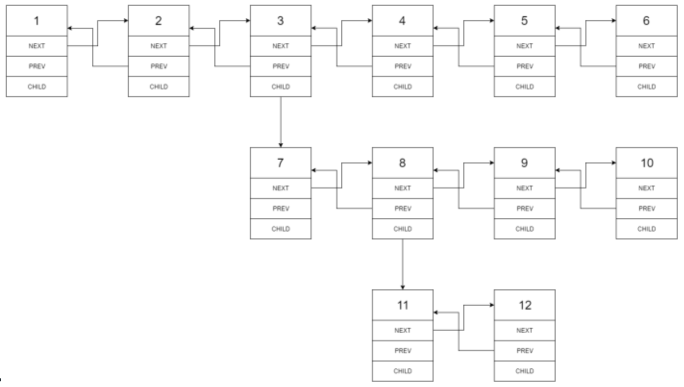
Medium

You are given a doubly linked list which in addition to the next and previous pointers, it could have a child pointer, which may or may not point to a separate doubly linked list. These child lists may have one or more children of their own, and so on, to produce a multilevel data structure, as shown in the example below.

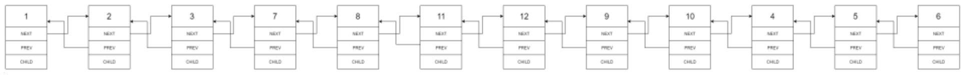
Flatten the list so that all the nodes appear in a single-level, doubly linked list. You are given the head of the first level of the list.

Example 1:

Input: head = [1,2,3,4,5,6,null,null,null,7,8,9,10,null,null,11,12]**Output:** [1,2,3,7,8,11,12,9,10,4,5,6]**Explanation:**The multilevel linked list in the input is as follows:



After flattening the multilevel linked list it becomes



Example 2:

Input: head = [1,2,null,3]**Output:** [1,3,2]**Explanation:**The input multilevel linked list is as follows: 1---2---NULL | 3---NULL

Example 3:

Input: head = []**Output:** []

How multilevel linked list is represented in test case:

We use the multilevel linked list from **Example 1** above:

1---2---3---4–-5-–6--NULL | 7---8---9---10--NULL | 11--12--NULL

The serialization of each level is as follows:

[1,2,3,4,5,6,null][7,8,9,10,null][11,12,null]

To serialize all levels together we will add nulls in each level to signify no node connects to the upper node of the previous level. The serialization becomes:

[1,2,3,4,5,6,null][null,null,7,8,9,10,null][null,11,12,null]

Merging the serialization of each level and removing trailing nulls we obtain:

[1,2,3,4,5,6,null,null,null,7,8,9,10,null,null,11,12]

Constraints:

- The number of Nodes will not exceed 1000.
- $1 <= Node.val <= 10^5$

```
"""
# Definition for a Node.
class Node(object):
  def __init__(self, val, prev, next, child):
    self.val = val
    self.prev = prev
    self.next = next
    self.child = child
"""
class Solution(object):

  def flatten(self, head: 'Node') -> 'Node':
    if not head:
      return head

    # pseudo head to ensure the `prev` pointer is never none
    pseudoHead = Node(None, None, head, None)
    self.flatten_dfs(pseudoHead, head)

    # detach the pseudo head from the real head
    pseudoHead.next.prev = None
    return pseudoHead.next


  def flatten_dfs(self, prev, curr):
    """ return the tail of the flatten list """
    if not curr:
      return prev

    curr.prev = prev
    prev.next = curr

    # the curr.next would be tempered in the recursive function
    tempNext = curr.next
    tail = self.flatten_dfs(curr, curr.child)
    curr.child = None
    return self.flatten_dfs(tail, tempNext)
```

O(N) time and space

281. Zigzag Iterator

Medium

Given two vectors of integers v1 and v2, implement an iterator to return their elements alternately.

Implement the ZigzagIterator class:

- ZigzagIterator(List<int> v1, List<int> v2) initializes the object with the two vectors v1 and v2.
- boolean hasNext() returns true if the iterator still has elements, and false otherwise.
- int next() returns the current element of the iterator and moves the iterator to the next element.

Example 1:

Input: v1 = [1,2], v2 = [3,4,5,6]**Output:** [1,3,2,4,5,6]
Explanation: By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: [1,3,2,4,5,6].

Example 2:

Input: v1 = [1], v2 = []**Output:** [1]

Example 3:

Input: v1 = [], v2 = [1]**Output:** [1]

Constraints:

- 0 <= v1.length, v2.length <= 1000
- 1 <= v1.length + v2.length <= 2000
- -2^{31} <= v1[i], v2[i] <= $2^{31} - 1$

Follow up: What if you are given k vectors? How well can your code be extended to such cases?

Clarification for the follow-up question:

The "Zigzag" order is not clearly defined and is ambiguous for k > 2 cases. If "Zigzag" does not look right to you, replace "Zigzag" with "Cyclic".

Example:

Input: v1 = [1,2,3], v2 = [4,5,6,7], v3 = [8,9]
Output: [1,4,8,2,5,9,3,6,7]

Algorithm

One could use the Iterator object (in Java or C++) as the pointer to the vector. Some of you might argue that we might be building a *iterator* with a built-in iterator. This has certain truth in it.

However, the key point here is that we could simply use some *index* and *integer* to implement the role of *pointer* in the above idea.

There are several advantages of using the *queue* of pointers, as one will see from the implementations later:

- First of all, we would achieve a constant time complexity for the next() function.
- The logics of implementation is much simplified and thus easy to read.

```
class ZigzagIterator: # 281
  def __init__(self, v1: List[int], v2: List[int]):
    self.vectors = [v1, v2]
    self.queue = deque()
    for index, vector in enumerate(self.vectors):
      # <index_of_vector, index_of_element_to_output>
      if len(vector) > 0:
        self.queue.append((index, 0))

  def next(self) -> int:

    if self.queue:
      vec_index, elem_index = self.queue.popleft()
      next_elem_index = elem_index + 1
      if next_elem_index < len(self.vectors[vec_index]):
        # append the pointer for the next round
        # if there are some elements left
        self.queue.append((vec_index, next_elem_index))

      return self.vectors[vec_index][elem_index]

    # no more element to output
    raise Exception

  def hasNext(self) -> bool:
    return len(self.queue) > 0

# Your ZigzagIterator object will be instantiated and called as such:
# i, v = ZigzagIterator(v1, v2), []
# while i.hasNext(): v.append(i.next())
```

Complexity Analysis

Let KK be the number of input vectors, although it is always two in the setting of the problem. This variable becomes relevant, once the input becomes KK vectors.

- Time Complexity: O(1)
  - For both the next() function and the hasNext() function, we have a constant time complexity, as we discussed before.
- Space Complexity: O(K)
  - We use a queue to keep track of the *pointers* to the input vectors in the variable self.vectors. As a result, we would need O(K) space for KK vectors.
  - Although the size of queue will reduce over time once we exhaust some shorter vectors, the space complexity for both functions is still O(K).

394. Decode String

Medium

Given an encoded string, return its decoded string.

The encoding rule is: k[encoded_string], where the encoded_string inside the square brackets is being repeated exactly k times. Note that k is guaranteed to be a positive integer.

You may assume that the input string is always valid; No extra white spaces, square brackets are well-formed, etc.

Furthermore, you may assume that the original data does not contain any digits and that digits are only for those repeat numbers, k. For example, there won't be input like 3a or 2[4].

Example 1:

Input: s = "3[a]2[bc]"**Output:** "aaabcbc"

Example 2:

Input: s = "3[a2[c]]"**Output:** "accaccacc"

Example 3:

Input: s = "2[abc]3[cd]ef"**Output:** "abcabccdcdcdef"

Example 4:

Input: s = "abc3[cd]xyz"**Output:** "abccdcdcdxyz"

Constraints:

- 1 <= s.length <= 30
- s consists of lowercase English letters, digits, and square brackets '[]'.
- s is guaranteed to be **a valid** input.
- All the integers in s are in the range [1, 300].

Algorithm

Iterate over the string s and process each character as follows:

Case 1) If the current character is a digit (0-9), append it to the number k.

Case 2) If the current character is a letter (a-z), append it to the currentString.

Case 3) If current character is a opening bracket [, push k and currentString intocountStack and stringStack respectively.

Case 4) Closing bracket]: We must begin the decoding process,

- We must decode the currentString. Pop currentK from the countStack and decode the pattern currentK[currentString]
- As the stringStack contains the previously decoded string, pop the decodedString from the stringStack. Update the decodedString = decodedString + currentK[currentString]

```
class Solution { //394
public:
    string decodeString(string s) {
        stack<int> countStack;
        stack<string> stringStack;
        string currentString;
        int k = 0;
        for (auto ch : s) {
            if (isdigit(ch)) {
                k = k * 10 + ch - '0';
            } else if (ch == '[') {
                // push the number k to countStack
                countStack.push(k);
                // push the currentString to stringStack
                stringStack.push(currentString);
                // reset currentString and k
                currentString = "";
                k = 0;
            } else if (ch == ']') {
                string decodedString = stringStack.top();
                stringStack.pop();
                // decode currentK[currentString] by appending currentString k times
                for (int currentK = countStack.top(); currentK > 0; currentK--) {
                    decodedString = decodedString + currentString;
                }
                countStack.pop();
                currentString = decodedString;
            } else {
                currentString = currentString + ch;
            }
        }
        return currentString;
    }
};
```

Complexity Analysis

Assume, *nn* is the length of the string *ss*.

- Time Complexity: *\mathcal{O}(\text{maxK} \cdot n)*O(maxK·*n*), where *\text{maxK}*maxK is the maximum value of *kk* and *nn* is the length of a given string *ss*. We traverse a string of size *nn* and iterate *kk* times to decode each pattern of form *\text{k[string]}*k[string]. This gives us worst case time complexity as *\mathcal{O}(\text{maxK} \cdot n)*O(maxK·*n*).
- Space Complexity: *\mathcal{O}(m+n)*O(*m*+*n*), where *mm* is the number of letters(a-z) and *nn* is the number of digits(0-9) in string *ss*. In worst case, the maximum size of *\text{stringStack}*stringStack and *\text{countStack}*countStack could be *mm* and *nn* respectively.

739. Daily Temperatures

Medium

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* i^{th} *day to get a warmer temperature*. If there is no future day for which this is possible, keep answer[i] == 0 instead.

Example 1:

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

Example 2:

Input: temperatures = [30,40,50,60]**Output:** [1,1,1,0]

Example 3:

Input: temperatures = [30,60,90]**Output:** [1,1,0]

Constraints:

- $1 <= temperatures.length <= 10^5$
- 30 <= temperatures[i] <= 100

Algorithm

1. Initialize an array answer with the same length as temperatures and all values initially set to 0. Also, initialize a stack as an empty array.
2. Iterate through temperatures. At each index currDay:
   - If the stack is not empty, that means there are previous days for which we have not yet seen a warmer day. While the current temperature is warmer than the temperature of prevDay (the index of the day at the top of the stack):
     - Set answer[prevDay] equal to the number of days that have passed between prevDay and the current day, that is, answer[prevDay] = currDay - prevDay.
   - Push the current index currDay onto the stack.
3. Return answer.

```
class Solution: # O(N) time and space
  def dailyTemperatures(self, temperatures: List[int]) -> List[int]:
    n = len(temperatures)
    answer = [0] * n
    stack = []

    for curr_day, curr_temp in enumerate(temperatures):
      # Pop until the current day's temperature is no warmer than the temperature at the top of the stack
      while stack and temperatures[stack[-1]] < curr_temp:
        prev_day = stack.pop()
        answer[prev_day] = curr_day - prev_day
      stack.append(curr_day)

    return answer
```

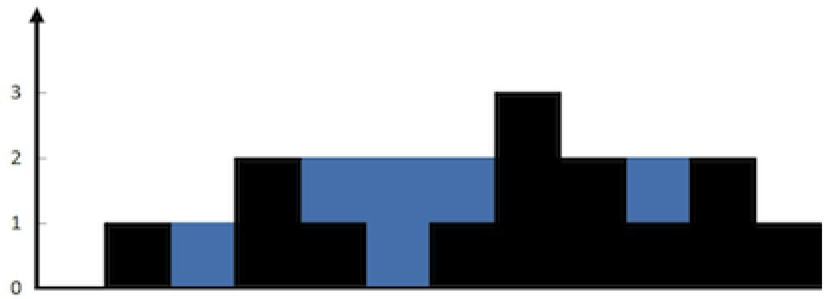
42. Trapping Rain Water

Hard

14902212Add to ListShare

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.

Example 1:



Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]**Output:** 6
Explanation: The above elevation map (black section) is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain wate r (blue section) are being trapped.

Example 2:

Input: height = [4,2,0,3,2,5]**Output:** 9

Constraints:

- $n == height.length$
- $1 <= n <= 2 * 10^4$
- $0 <= height[i] <= 10^5$

Approach 2: Dynamic Programming

Intuition

In brute force, we iterate over the left and right parts again and again just to find the highest bar size upto that index. B ut, this could be stored. Voila, dynamic programming.

The concept is illustrated as shown:

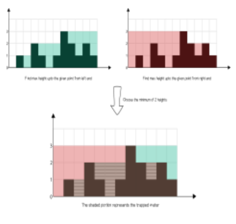

Find max height upto the given point from left end
Find max height upto the given point from right end
Choose the minimum of 2 heights
The shaded portion represents the trapped water

Fig: Dynamic Programming Approach

Algorithm

- Find maximum height of bar from the left end upto an index i in the array left_max.
- Find maximum height of bar from the right end upto an index i in the array right_max.
- Iterate over the height array and update ans:
    - Add min(left_max[*i*],right_max[*i*])−height[*i*] ->*ans*

```
int trap(vector<int>& height)
{
   if(height.empty())
      return 0;
   int ans = 0;
   int size = height.size();
   vector<int> left_max(size), right_max(size);
   left_max[0] = height[0];
   for (int i = 1; i < size; i++) {
      left_max[i] = max(height[i], left_max[i - 1]);
   }
   right_max[size - 1] = height[size - 1];
   for (int i = size - 2; i >= 0; i--) {
      right_max[i] = max(height[i], right_max[i + 1]);
   }
   for (int i = 1; i < size - 1; i++) {
      ans += min(left_max[i], right_max[i]) - height[i];
   }
   return ans;
}
```

Complexity analysis

- Time complexity: $O(n)$*O*(*n*).
    - We store the maximum heights upto a point using 2 iterations of $O(n)$*O*(*n*) each.
    - We finally update ans using the stored values in $O(n)$*O*(*n*).
- Space complexity: $O(n)$*O*(*n*) extra space.
    - Additional $O(n)$*O*(*n*) space for left_max and right_max arrays than in brute force approach.

There are many other ways to solve this- using stack, using two pointers etc.

402. Remove K Digits

Medium

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

Example 1:

Input: num = "1432219", k = 3**Output:** "1219"**Explanation:** Remove the three digits 4, 3, and 2 to form the new number 1219 which is the smallest.

Example 2:

Input: num = "10200", k = 1**Output:** "200"**Explanation:** Remove the leading 1 and the number is 200. Note that the output must not contain leading zeroes.

Example 3:

Input: num = "10", k = 2**Output:** "0"**Explanation:** Remove all the digits from the number and it is left with nothing which is 0.

Constraints:

- $1 <= k <= num.length <= 10^5$
- num consists of only digits.
- num does not have any leading zeros except for the zero itself.

```
class Solution: # greedy - O(N) time and space
    def removeKdigits(self, num: str, k: int) -> str:
        numStack = []

        # Construct a monotone increasing sequence of digits
        for digit in num:
            while k and numStack and numStack[-1] > digit:
                numStack.pop()
                k -= 1

            numStack.append(digit)

        # - Trunk the remaining K digits at the end
        # - in the case k==0: return the entire list
        finalStack = numStack[:-k] if k else numStack

        # trip the leading zeros
        return "".join(finalStack).lstrip('0') or "0"
```

456. 132 Pattern

Medium

Given an array of n integers nums, a **132 pattern** is a subsequence of three integers nums[i], nums[j] and nums[k] such that i < j < k and nums[i] < nums[k] < nums[j].

Return *true if there is a **132 pattern** in nums, otherwise, return false.*

Example 1:

Input: nums = [1,2,3,4]**Output:** false**Explanation:** There is no 132 pattern in the sequence.

Example 2:

Input: nums = [3,1,4,2]**Output:** true**Explanation:** There is a 132 pattern in the sequence: [1, 4, 2].

Example 3:

Input: nums = [-1,3,2,0]**Output:** true**Explanation:** There are three 132 patterns in the sequence: [-1, 3, 2], [-1, 3, 0] and [-1, 2, 0].

Constraints:

- n == nums.length
- $1 <= n <= 2 * 10^5$
- $-10^9 <= nums[i] <= 10^9$

```
class Solution:
    def find132pattern(self, nums: List[int]) -> bool:
        if len(nums) < 3:
            return False
        stack = []
        min_array = [-1] * len(nums)
        min_array[0] = nums[0]
        for i in range(1, len(nums)):
            min_array[i] = min(min_array[i - 1], nums[i])

        for j in range(len(nums) - 1, -1, -1):
            if nums[j] <= min_array[j]:
                continue
            while stack and stack[-1] <= min_array[j]:
                stack.pop()
            if stack and stack[-1] < nums[j]:
                return True
            stack.append(nums[j])
        return False
```

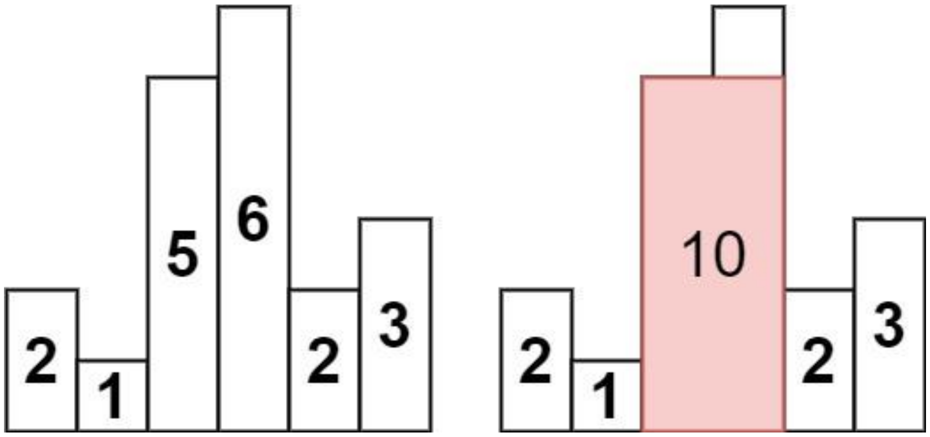
O(N) time and space complexity

84. Largest Rectangle in Histogram

Hard

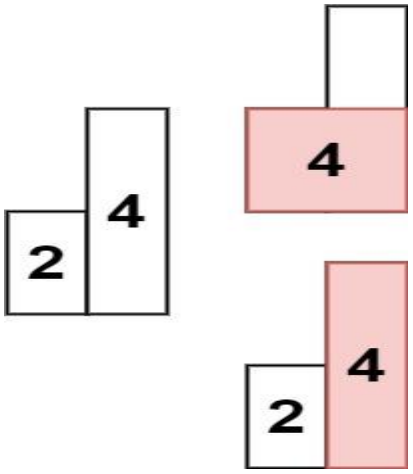
Given an array of integers heights representing the histogram's bar height where the width of each bar is 1, return *the area of the largest rectangle in the histogram*.

Example 1:



Input: heights = [2,1,5,6,2,3]**Output:** 10**Explanation:** The above is a histogram where width of each bar is 1.The largest rectangle is shown in the red area, which has an area = 10 units.

Example 2:



Input: heights = [2,4]**Output:** 4

Constraints:

- $1 <= heights.length <= 10^5$
- $0 <= heights[i] <= 10^4$
-

```
class Solution: # LC-84: using stack - O(N) time and space
    def largestRectangleArea(self, heights: List[int]) -> int:
        stack = [-1]
        max_area = 0
        for i in range(len(heights)):
            while stack[-1] != -1 and heights[stack[-1]] >= heights[i]:
                current_height = heights[stack.pop()]
                current_width = i - stack[-1] - 1
                max_area = max(max_area, current_height * current_width)
            stack.append(i)

        while stack[-1] != -1:
            current_height = heights[stack.pop()]
            current_width = len(heights) - stack[-1] - 1
            max_area = max(max_area, current_height * current_width)
        return max_area
```

862. Shortest Subarray with Sum at Least K

Hard

Given an integer array nums and an integer k, return *the length of the shortest non-empty* ***subarray*** *of* nums *with a sum of at least* k. If there is no such **subarray**, return -1.

A **subarray** is a **contiguous** part of an array.

Example 1:

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

Example 2:

Input: nums = [1,2], k = 4**Output:** -1

Example 3:

Input: nums = [2,-1,2], k = 3**Output:** 3

Constraints:

- $1 <= nums.length <= 10^5$
- $-10^5 <= nums[i] <= 10^5$
- $1 <= k <= 10^9$

```
class Solution(object): # O(N) time and space
  def shortestSubarray(self, A: List[int], K: int) -> int:
    N = len(A)
    P = [0]
    for x in A:
      P.append(P[-1] + x)

    #Want smallest y-x with Py - Px >= K
    ans = N+1 # N+1 is impossible
    monoq = collections.deque() #opt(y) candidates, represented as indices of P
    for y, Py in enumerate(P):
      #Want opt(y) = largest x with Px <= Py - K
      while monoq and Py <= P[monoq[-1]]:
        monoq.pop()

      while monoq and Py - P[monoq[0]] >= K:
        ans = min(ans, y - monoq.popleft())

      monoq.append(y)

    return ans if ans < N+1 else –1
```