# ARRAY QNS NOTES (450 and extras)

## REVISION PURPOSE

collected by: KALEEM AHMED





1. Implement Dynamic Array (Vector) Class
2. Find the "Kth" max and min element of an array
3. Sort an Array of 0 1 2
4. Move all the negative elements to one side of the array
5. Find the Union and Intersection of the two sorted arrays.
6. Write a program to cyclically rotate an array by one.
7. Find the Largest sum contiguous Subarray [V. IMP]
8. Minimize the maximum difference between heights [V.IMP]
9. Minimum number of Jumps to reach the end of an array
10. Find a duplicate in an array of N+1 Integers
11. Merge 2 sorted arrays without using Extra space.
12. Prefix Sum Technique Achieves (imp)
13. Kadane's Algorithm [V.V.V.V.V IMP]
14. Max length Subarray with an even sum
15. Length of the longest Subarray with an equal number of odd and even elements
16. Maximize the sum Array by flipping the sign of all elements of a single subarray (read)
17. Equilibrium point
18. Max Circular Subarray sum, longest alternating subarray (v imp)
19. Merge Intervals, Non-overlapping intervals, Insert Interval
20. Min operation to make the array increasing
21. Next Permutation
22. Count Inversion, Reverse Pairs Optimized
23. Max Points on a Line
24. Maximum Index (can be applied to solve many problems)
25. Best time to buy and Sell stock
26. Find all pairs on an integer array whose sum is equal to a given number
27. Find common elements In 3 sorted arrays
28. Rearrange the array in alternating +ve and -ve items without space
29. Find if there is any subarray with a sum equal to 0
30. Find the factorial of a large number
31. Find the maximum product subarray
32. Find the longest consecutive subsequence
33. Longest Fibonacci Sequence Leetcode
34. Find all elements that appear more than "n/k" times.
35. Maximum profit by buying and selling a share at most twice
36. Find whether an array is a subset of another array
37. Find the triplet that sums to a given value
38. Remove Duplicates from a Tricky Array LC
39. Trapping Rainwater problem
40. Smallest Subarray with a sum greater than a given value
41. Three-way partitioning of an array around a given value
42. Minimum swaps required to bring elements less than or equal to K together
43. Min pair merge operations required to make Array non-increasing
44. Maximize K to array Palindrome, each element replaced by its remainder with K
45. Min number of merge operations to make an array palindrome
46. Collecting Chocolates (great ad-hoc), Movement of Robots
47. Median of 2 sorted arrays of equal size, different size (log N imp)
48. Minimum swaps to sort the array (imp)
49. Skyline Problem (optional now)
50. Minimum Time to Make Rope Colorful
51. Count subarray with a product less than K (VVVVV imp *(to count subarray))
52. Longest subarray of ones with 1 deletion
53. Longest Substring without repeating character
54. K radius subarray average, Fruits into a basket
55. maximum-points-you-can-obtain-from-cards
56. No of substring containing all three chars (vvvvvv Imp)
57. Binary subarray with sum
58. Longest repeating char replacement
59. Maximize-the-confusion-of-an-exam
60. Frequency of max frequent element (vvvvv imp)
61. Substr with the largest variance

```cpp
class Vector {
    int* arr;
    int capacity;
    int current;
public:
    Vector() {
        arr = new int[1];
        capacity = 1;
        current = 0;
    }
    ~Vector() {
        delete[] arr;
    }
    void push(int data) {
        if (current == capacity) {
            int* temp = new int[2 * capacity];
            for (int i = 0; i < capacity; i++)
                temp[i] = arr[i];
            delete[] arr;
            capacity *= 2;
            arr = temp;
        }
        arr[current] = data;
        current++;
    }
    int get(int index) {
        if (index < current)
            return arr[index];
    }
    void pop() {
        if (current > 0)
            current--;
    }
    int size() {
        return current;
    }
    int getCapacity() {
        return capacity;
    }
};

int main() {
    Vector v;   if using Vector arr = new vector();  // it returns pointer to that obj
                so further use arr->push_back(x); syntax for all methods
    v.push(10);
    v.push(40);
    v.push(40);

    cout << "Vector capacity: " << v.getCapacity() << endl;
    cout << "Vector size: " << v.size() << endl;
    v.pop();
    cout << "Vector capacity: " << v.getCapacity() << endl;
    cout << "Vector size: " << v.size() << endl;

    return 0;
```

```cpp

#include <bits/stdc++.h>
using namespace std;
template <typename T> class vectorClass {
    T* arr;
    int capacity;
    int current;
public:
    vectorClass() {
        arr = new T[1];
        capacity = 1;
        current = 0;
    }
    ~ vectorClass() {
        delete [] arr;
    }
    void push(T data) {
        if (current == capacity) {
            T* temp = new T[2 * capacity];
            for (int i = 0; i < capacity; i++)
                temp[i] = arr[i];
            delete[] arr;
            capacity *= 2;
            arr = temp;
        }
        arr[current] = data;
        current++;
    }
    void push(T data, int index) {
        if (index == capacity)
            push(data);
        else
            arr[index] = data;
    }
    T get(int index) {
        if (index < current)
            return arr[index];
    }
    void pop() { current--; }
    int size() { return current; }
    int getcapacity() { return capacity; }
    void print() {
        for (int i = 0; i < current; i++)
            cout << arr[i] << " ";
        cout << endl;
    }
};
```

reverse array iter

```
void reverse(int arr[], int n)
{
    for (int low = 0, high = n - 1; low < high; low++, high--) {
        swap(arr[low], arr[high]);
    }
}
```

reverse array using recursion

```
void reverse(int arr[], int low, int high)
{
    if (low < high)
    {
        swap(arr[low], arr[high]);
        reverse(arr, low + 1, high - 1);
    }
}
```

rotate by one position clockwise

```
void rotate(int arr[], int n) {
    int last = arr[n-1];
    for(int i=n-2; i>=0; i--) {
        arr[i+1] = arr[i];
    }
    arr[0] = last;
}
```

rotate array by k position:

```
class Solution {
    public:
    void rotateArr(int arr[], int d, int n) {
        d = d % n;
        reverse(arr, arr+n);
        reverse(arr, arr+(n-d));
        reverse(arr+(n-d), arr+n);
    }
};
```

Kth smallest element (partition method)

```
class Solution {
public:
    int partition(int arr[], int low, int high) {
        int pivot = arr[low];
        int i = low, j = high + 1;

        while (i < j) {
            do {
                i++;
            } while (arr[i] <= pivot);
            do {
                j--;
            } while (arr[j] > pivot);

            if (i < j) swap(arr[i], arr[j]);
        }

        swap(arr[low], arr[j]);
        return j;
    }
    int kthSmallest(int arr[], int l, int r, int k) {
        int low = l;
        int high = r;

        while (low <= high) {
            int pivotIndex = partition(arr, low, high);

            if (pivotIndex == k - 1)
                return arr[pivotIndex];
            else if (pivotIndex > k - 1)
                high = pivotIndex - 1;
            else
                low = pivotIndex + 1;
        }

        return -1; // Return -1 for invalid inputs
    }
};
```

**Time complexity:** O(N)

**Auxiliary Space:** O(1)

**Two Pointer Approach:** The idea is to solve this problem with constant space and linear time is by using a two-pointer or two-variable approach where we simply take two variables like left and right which hold the 0 and N-1 indexes. Just need to check that :

1. Check If the left and right pointer elements are negative then simply increment the left pointer.
2. Otherwise, if the left element is positive and the right element is negative then simply swap the elements, and simultaneously increment and decrement the left and right pointers.
3. Else if the left element is positive and the right element is also positive then simply decrement the right pointer.
4. Repeat the above 3 steps until the left pointer ≤ right pointer.

```
void rearrange(int arr[], int n)
{
    int j = 0;
    for (int i = 0; i < n; i++) {
        if (arr[i] < 0) {
            if (i != j)
                swap(arr[i], arr[j]);
            j++;
        }
    }
}
```

```
// Using Dutch National Flag Algorithm.
void reArrange(int arr[],int n){
      int low =0,high = n-1;
      while(low<high){
      if(arr[low]<0){
          low++;
      }else if(arr[high]>0){
          high--;
      }else{
        swap(arr[low],arr[high]);
      }
    }
}
```

Union of two arrays

```
class Solution {
public:
    // Function to return the count of the number of elements
    int doUnion(int a[], int n, int b[], int m) {
        int res[m + n];
        int i = 0, j = 0, k = 0;

        while (i < n && j < m) {
            if (a[i] == b[j]) {
                res[k++] = a[i++];
                j++;
            } else if (a[i] < b[j]) {
                res[k++] = a[i++];
            } else {
                res[k++] = b[j++];
            }
        }

        while (i < n) {
            res[k++] = a[i++];
        }

        while (j < m) {
            res[k++] = b[j++];
        }

        return k;
    }
};
```

accepted

```
class Solution{
    public:
    //Function to return the count of number of eleme
    int doUnion(int a[], int n, int b[], int m)  {
        //code here
        unordered_set<int>s;
        for(int i=0;i<n;i++){
            s.insert(a[i]);
        }
        for(int i=0;i<m;i++){
            s.insert(b[i]);
        }
        int count=0;
        for(auto it:s){
            count++;
        }
        return count;

    }
};
```

Kadane's Algo:

```
class Solution{
    public:
    long long maxSubarraySum(int arr[], int n){

        long long currSum = 0;
        long long maxSum = INT_MIN;

        for(int i=0; i<n; i++) {
            currSum += arr[i];
            maxSum = max(maxSum, currSum);

            currSum = (currSum < 0) ? 0 : currSum;
            // only -ve matters else keep adding
        }

         return maxSum;
    }
};
```

Given an array **arr[]** denoting heights of **N** towers and a positive integer **K.**

For **each** tower, you must perform **exactly one** of the following operations **exactly once**.

- **Increase** the height of the tower by **K**
- **Decrease** the height of the tower by **K**

Find out the **minimum** possible difference between the height of the shortest and tallest towers after you have modified each tower.

You can find a slight modification of the problem here.
**Note:** It is **compulsory** to increase or decrease the height by K for each tower. **After** the operation, the resultant array should **not** contain any **negative integers**.

**Example 1:**

**Input:**
K = 2, N = 4
Arr[] = {1, 5, 8, 10}
**Output:**
5
**Explanation:**
The array can be modified as
{1+k, 5-k, 8-k, 10-k} = {3, 3, 6, 8}.
The difference between
the largest and the smallest is 8-3 = 5.

**Example 2:**

**Input:**
K = 3, N = 5
Arr[] = {3, 9, 12, 16, 20}
**Output:**
11
**Explanation:**
The array can be modified as
{3+k, 9+k, 12-k, 16-k, 20-k} -> {6, 12, 9, 13, 17}.
The difference between
the largest and the smallest is 17-6 = 11.

```cpp
class Solution {
  public:
    int getMinDiff(int arr[], int n, int k) {
        // sort for better adjacent candidate
        sort(arr, arr+n);
        int initial_mx = arr[n-1]-k, initial_mn = arr[0]+k;

        int minDiff = arr[n-1] - arr[0];
        int _max, _min;

        for(int i=0; i<n-1; i++){
            _max = max(arr[i]+k, initial_mx);
            _min = min(arr[i+1]-k, initial_mn);
            if(_min<0) continue; // becz -- will increase ans
            minDiff = min(minDiff, (_max - _min));
        }
        return minDiff;
    }
};
```



# Minimum number of jumps

Medium Accuracy: 11.91% Submissions: 639K+ Points: 4



Given an array of **N** integers **arr[]** where each element represents the **maximum** length of the jump that can be made forward from that element. This means if arr[i] = x, then we can jump any distance y such that $y \leq x$.

Find the minimum number of jumps to reach the end of the array (starting from the first element). If an element is **0**, then you cannot move through that element.

**Note:** Return -1 if you can't reach the end of the array.

**Example 1:**

**Input:**
N = 11
arr[] = {1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9}
**Output:** 3
**Explanation:**
First jump from 1st element to 2nd element with value 3. Now, from here we jump to 5th element with value 9, and from here we will jump to the last.

**Example 2:**

**Input :**
N = 6
arr = {1, 4, 3, 2, 6, 7}
**Output:** 2
**Explanation:**
First we jump from the 1st to 2nd element and then jump to the last element.

```cpp
class Solution {
public:
    int minJumps(int arr[], int n) {
        if (n == 0 || arr[0] == 0)
            return -1;

        int jumps = 1;
        int steps = arr[0];
        int maxReach = arr[0];


        for (int i = 1; i < n; i++) {
            if (steps == 0)     // for 2 1 0 3
                return -1;

            if (i >= n - 1)
                return jumps;

            maxReach = max(maxReach, i + arr[i]);
            steps--;

            if (steps == 0) {
                jumps++;
                steps = maxReach - i;
            }
        }

        return jumps;
    }
};
```

## Find dups (cycle sort)

```cpp
class Solution {
public:
    void sort(vector<int>& nums){
        int i=0;
        while(i<nums.size()){
            int correct = nums[i]-1;
            if(nums[i]!=nums[correct]){
                swap(nums[i],nums[correct]);
            }else{
                i++;
            }
        }
    }
    int findDuplicate(vector<int>& nums) {
        sort(nums);
        int val;
        for(int i=0;i<nums.size();i++){
            if(nums[i]-1!=i)
                val = nums[i];
        }
        return val;
    }
};
```

## Find dups (slow fast pointer)



```
public int findDuplicate_fastSlow(int[] nums) {
    int slow = 0;
    int fast = 0;
    do {
        slow = nums[slow];
        fast = nums[nums[fast]];
    } while (slow != fast);

    slow = 0;
    while (slow != fast) {
        slow = nums[slow];
        fast = nums[fast];
    }

    return slow;
}
```



## Find dups (-ve mark elm approach)



```
// Visited
public static int findDuplicate_mark(int[] nums) {
    int len = nums.length;
    for (int num : nums) {
        int idx = Math.abs(num);
        if (nums[idx] < 0) {
            return idx;
        }
        nums[idx] = -nums[idx];
    }

    return len;
}
```

using elm as visited itself mark it -ve first time and when second time that sends at that idx and its elm is negative that means its repeated

## Merge two sorted arrays const space (N^2)

```cpp
// O(N^2) approach fixArray (O(N)* min(N, M))
class Solution {
public:
    void fixArray(long long arr[], int n) {
        int i = 0;
        while(i < n-1) {
            if(arr[i] > arr[i+1])
                swap(arr[i], arr[i+1]);
            i++;
        }
    }

    void merge(long long arr1[], long long arr2[], int n, int m) {

        int i = 0, j = 0;
        while(i < n && j < m) {
            if(arr1[i] > arr2[j]) {
                swap(arr1[i], arr2[j]);
                i++;
                fixArray(arr2, m);
                // Pass the size of arr2 to the fixArray function imp
            }
            else {
                i++, j++;
            }
        }
    }
};

```

## Merge two sorted arrays const space (NlogN)

APPROACH: To merge the arrays in-place without extra spaces,
we can start from the end of arr1 and the beginning of arr2.
We compare the elements at these positions and swap them if necessary.
By repeatedly doing this, we ensure that the larger elements are moved
to the end of arr1 and the smaller elements are moved to the beginning of arr2.
Finally, we sort both arrays to obtain the merged sorted order.

```cpp
// Optimised approach (min(N, M) + O(NLOGN + MLOGM))
class Solution{
public:
    //Function to merge the arrays
    void merge(long long arr1[], long long arr2[], int n, int m)
    {
        int i = n-1;
        int j = 0;

        while(i >= 0 && j < m) {
            if(arr1[i] >= arr2[j]) {
                swap(arr1[i], arr2[j]);
                i--;
                j++;
            }
            else {
                break;
            }
        }

        sort(arr1, arr1+n);
        sort(arr2, arr2+m);
    }
};
```

Max subarray size such that sum is less than k

```cpp
// C++ program for the above approach
#include <bits/stdc++.h>
using namespace std;
// Maximum subarray size, such that all subarrays of that size have sum less than k
void func(vector<int> arr, int k, int n) {
    int ans = n;
    int sum = 0, start = 0;

    for (int end = 0; end < n; end++) {
        sum += arr[end];
        while (sum > k) {
            // Sliding window from right
            sum -= arr[start];
            start++;
            // Storing sub-array size - 1 for which sum was greater than k
            ans = min(ans, end - start + 1);
            // Sum will be 0 if start>end  because all elements are positive
            // start>end only when arr[end]>k i.e, there is an array element with
            // value greater than k, so sub-array sum cannot be less than k.
            // Input : arr[] = {1, 2, 10, 4} and k = 8. Output : -1
            if (sum == 0)
                break;
        }
        if (sum == 0) {
            ans = -1;
            break;
        }
    }
    cout << ans;
}
```

# Prefix Sum Array – Implementation and Applications in Competitive Programming

Read | Discuss | Courses | Practice | Video

Given an array **arr[]** of size **N**, find the prefix sum of the array. A prefix sum array is another array **prefixSum[]** of the same size, such that the value of **prefixSum[i]** is **arr[0] + arr[1] + arr[2] . . . arr[i]**.

**Examples:**

***Input:*** *arr[] = {10, 20, 10, 5, 15}*
***Output:*** *prefixSum[] = {10, 30, 40, 45, 60}*
***Explanation:*** *While traversing the array, update the element by adding it with its previous element.*
*prefixSum[0] = 10,*
*prefixSum[1] = prefixSum[0] + arr[1] = 30,*
*prefixSum[2] = prefixSum[1] + arr[2] = 40 and so on.*

Recommended Practice
**Adding Ones** — Try It!

**Approach:** To solve the problem follow the given steps:

- Declare a new array **prefixSum[]** of the same size as the input array
- Run a for loop to traverse the input array
- For each index add the value of the current element and the previous value of the prefix sum array

## Sum of an array between indexes L and R using Prefix Sum:

*Given an array arr[] of size **N**. Given **Q** queries and in each query given L and R, Print the sum of array elements from index L to R.*

Follow the given steps to solve the problem:

- Create the prefix sum array of the given input array
- Now for every query (1-based indexing)
  - If L is greater than 1, then print prefixSum[R] – prefixSum[L-1]
  - else print prefixSum[R]

### Example Problem:

Consider an array of size N with all initial values as 0. Perform the given 'm' add operations from index 'a' to 'b' and evaluate the highest element in the array. An add operation adds 100 to all the elements from a to b (both inclusive).

**Example:**

***Input:*** *n = 5, m = 3*
*a = 2, b = 4.*
*a = 1, b = 3.*
*a = 1, b = 2.*
***Output:*** *300*
***Explanation:***
*After I operation – A[] = {0, 100, 100, 100, 0}*
*After II operation – A[] = {100, 200, 200, 100, 0}*
*After III operation – A[] = {200, 300, 200, 100, 0}*
*Highest element: 300*

*The efficient approach is to use **Prefix Sum Array***



Follow the given steps to solve the problem:

- Run a loop for '**m**' times, inputting '**a**' and '**b**'.
- Add **100** at index '**a-1**' and subtract **100** from index '**b**'.
- After completion of '**m**' operations, compute the prefix sum array.
- Scan the largest element and we're done.

**Explanation:** We added 100 at 'a' because this will add 100 to all elements while taking the prefix sum array. Subtracting 100 from 'b+1' will reverse the changes made by adding 100 to elements from 'b' onward.

Below is the illustration of the above approach:

*After I operation –*
*A[] = {0, 100, 0, 0, -100}*

*After II operation –*
*A[] = {100, 100, 0, -100, -100}*

*After III operation –*
*A[] = {200, 100, -100, -100, -100}*

*Final Prefix Sum Array : 200 300 200 100 0*
*The required highest element : 300*

```
int find(int m, vector<pair<int, int> > q) {
    int mx = 0;
    vector<int> pre(5, 0);
    for (int i = 0; i < m; i++) {
        int a = q[i].first, b = q[i].second;
        // add 100 at first index and subtract 100 from last index
        // pre[1] becomes 100
        pre[a - 1] += 100;
        // pre[4] becomes -100 and this
        pre[b] -= 100;
        // continues m times as we input diff. values of a and b
    }
    for (int i = 1; i < 5; i++) {
        // add all values in a cumulative way
        pre[i] += pre[i - 1];
        // keep track of max value
        mx = max(mx, pre[i]);
    }

    return mx;
}
```





## Applications of Prefix Sum:

- Equilibrium index of an array: The equilibrium index of an array is an index such that the sum of elements at lower indexes is equal to the sum of elements at higher indexes.
- Find if there is a subarray with 0 sums: Given an array of positive and negative numbers, find if there is a subarray (of size at least one) with 0 sum.
- Maximum subarray size, such that all subarrays of that size have a sum less than k: Given an array of n positive integers and a positive integer k, the task is to find the maximum subarray size such that all subarrays of that size have the sum of elements less than k.
- Find the prime numbers which can be written as sum of most consecutive primes: Given an array of limits. For every limit, find the prime number which can be written as the sum of the most consecutive primes smaller than or equal to the limit.
- Longest Span with same Sum in two Binary arrays: Given two binary arrays, arr1[] and arr2[] of the same size n. Find the length of the longest common span (i, j) where j >= i such that arr1[i] + arr1[i+1] + .... + arr1[j] = arr2[i] + arr2[i+1] + .... + arr2[j].
- Maximum subarray sum modulo m: Given an array of n elements and an integer m. The task is to find the maximum value of the sum of its subarray modulo m i.e find the sum of each subarray mod m and print the maximum value of this modulo operation.
- Maximum subarray size, such that all subarrays of that size have sum less than k: Given an array of n positive integers and a positive integer k, the task is to find the maximum subarray size such that all subarrays of that size have the sum of elements less than k.
- Maximum occurred integer in n ranges : Given n ranges of the form L and R, the task is to find the maximum occurring integer in all the ranges. If more than one such integer exits, print the smallest one.
- Minimum cost for acquiring all coins with k extra coins allowed with every coin: You are given a list of N coins of different denominations. you can pay an amount equivalent to any 1 coin and can acquire that coin. In addition, once you have paid for a coin, we can choose at most K more coins and can acquire those for free. The task is to find the minimum amount required to acquire all the N coins for a given value of K.
- Random number generator in arbitrary probability distribution fashion: Given n numbers, each with some frequency of occurrence. Return a random number with a probability proportional to its frequency of occurrence.

# Find the prime numbers which can written as sum of most consecutive primes

Read Discuss Courses Practice

Given an array of limits. For every limit, find the prime number which can be written as the sum of the most consecutive primes smaller than or equal to limit.
The maximum possible value of a limit is 10^4.

**Example:**

```
Input  : arr[] = {10, 30}
Output : 5, 17
Explanation : There are two limit values 10 and 30.
Below limit 10, 5 is sum of two consecutive primes,
2 and 3. 5 is the prime number which is sum of largest
chain of consecutive below limit 10.

Below limit 30, 17 is sum of four consecutive primes.
2 + 3 + 5 + 7 = 17
```

Below are steps.

1. Find all prime numbers below a maximum limit (10^6) using Sieve of Sundaram and store them in primes[].
2. Construct a prefix sum array **prime_sum[]** for all prime numbers in primes[]
   prime_sum[i+1] = prime_sum[i] + primes[i].
   Difference between two values in prime_sum[i] and prime_sum[j] represents sum of consecutive primes from index i to index j.
3. Traverse two loops , outer loop from i (0 to limit) and inner loop from j (0 to i)
4. For every i, inner loop traverse (0 to i), we check if current sum of consecutive primes (**consSum** = prime_sum[i] – prime_sum[j]) is prime number or not (we search consSum in prime[] using Binary search).
5. If consSum is prime number then we update the result if the current length is more than length of current result.

```cpp
int LSCPUtil(int limit, vector<int>& prime, long long int sum_prime[]) {
    int max_length = -1, prime_number = -1;
    for (int i = 0; prime[i] <= limit; i++) {
        for (int j = 0; j < i; j++) {
            if (sum_prime[i] - sum_prime[j] > limit)
                break;
            long long int consSum = sum_prime[i] - sum_prime[j];
            if (binary_search(prime.begin(), prime.end(), consSum)) {
                if (max_length < i - j + 1) {
                    max_length = i - j + 1;
                    prime_number = consSum;
                }
            }
        }
    }
    return prime_number;
}
void LSCP(int arr[], int n) {
    vector<int> primes;
    sieveEratosthenes(primes);
    long long int sum_prime[primes.size() + 1];
    sum_prime[0] = 0;
    for (int i = 1; i <= primes.size(); i++)
        sum_prime[i] = primes[i - 1] + sum_prime[i - 1];
    for (int i = 0; i < n; i++)
        cout << LSCPUtil(arr[i], primes, sum_prime) << " ";
}
```

# Longest Span with same Sum in two Binary arrays





Given two binary arrays, arr1[] and arr2[] of the same size n. Find the length of the longest common span (i, j) where j >= i such that arr1[i] + arr1[i+1] + .... + arr1[j] = arr2[i] + arr2[i+1] + .... + arr2[j].

The expected time complexity is Θ(n).

**Examples:**

```
Input: arr1[] = {0, 1, 0, 0, 0, 0};
       arr2[] = {1, 0, 1, 0, 0, 1};
Output: 4
The longest span with same sum is from index 1 to 4.

Input: arr1[] = {0, 1, 0, 1, 1, 1, 1};
       arr2[] = {1, 1, 1, 1, 1, 0, 1};
Output: 6
The longest span with same sum is from index 1 to 6.

Input: arr1[] = {0, 0, 0};
       arr2[] = {1, 1, 1};
Output: 0

Input: arr1[] = {0, 0, 1, 0};
       arr2[] = {1, 1, 1, 1};
Output: 1
```

```cpp
#include<bits/stdc++.h>
using namespace std;
int longestCommonSum(bool arr1[], bool arr2[], int n) {
    int maxLen = 0;
    // One by one pick all possible starting points of subarrays
    for (int i=0; i<n; i++) {
        int sum1 = 0, sum2 = 0;
        for (int j=i; j<n; j++) {
            sum1 += arr1[j];
            sum2 += arr2[j];
            if (sum1 == sum2) {
                int len = j-i+1;
                if (len > maxLen)
                    maxLen = len;
            }
        }
    }
    return maxLen;
}
```

Optimised



```cpp
int longestCommonSum(bool arr1[], bool arr2[], int n) {
    // Find difference between the two now it is find max sun=barray len with zero sum
    int arr[n];
    for (int i=0; i<n; i++)
        arr[i] = arr1[i] - arr2[i];

    unordered_map<int, int> mp;
    int sum = 0,  max_len = 0; // Initialize result

    for (int i = 0; i < n; i++) {
        sum += arr[i];
        // To handle sum=0 at last index
        if (sum == 0)
            max_len = i + 1;
        // If this sum is seen before, then update max_len if required
        if (mp.find(sum) != mp.end())
            max_len = max(max_len, i - mp[sum]);
        else
            mp[sum] = i;
    }

    return max_len;
}
```

# Maximum subarray sum modulo m

Read Discuss Courses Practice

Given an array of **n** elements and an integer m. The task is to find the maximum value of the sum of its subarray modulo m i.e find the sum of each subarray mod m and print the maximum value of this modulo operation.

**Examples:**

```
Input : arr[] = { 3, 3, 9, 9, 5 }
        m = 7
Output : 6
All sub-arrays and their value:
{ 9 } => 9%7 = 2
{ 3 } => 3%7 = 3
{ 5 } => 5%7 = 5
{ 9, 5 } => 14%7 = 2
{ 9, 9 } => 18%7 = 4
{ 3, 9 } => 12%7 = 5
{ 3, 3 } => 6%7 = 6
{ 3, 9, 9 } => 21%7 = 0
{ 3, 3, 9 } => 15%7 = 1
{ 9, 9, 5 } => 23%7 = 2
{ 3, 3, 9, 9 } => 24%7 = 3
{ 3, 9, 9, 5 } => 26%7 = 5
{ 3, 3, 9, 9, 5 } => 29%7 = 1

Input : arr[] = {10, 7, 18}
        m = 13
Output : 12
The subarray {7, 18} has maximum sub-array
sum modulo 13.
```





```cpp
int maxSubarray(int arr[], int n, int m)
{
    int prefix = 0, maxim = 0;

    set<int> S;
    S.insert(0);

    // Traversing the array.
    for (int i = 0; i < n; i++)
    {
        // Finding prefix sum.
        prefix = (prefix + arr[i])%m;

        // Finding maximum of prefix sum.
        maxim = max(maxim, prefix);

        // Finding iterator pointing to the first
        // element that is not less than value
        // "prefix + 1", i.e., greater than or
        // equal to this value.
        auto it = S.lower_bound(prefix+1);

        if (it != S.end())
            maxim = max(maxim, prefix - (*it) + m );

        // Inserting prefix in the set.
        S.insert(prefix);
    }

    return maxim;
}
```

# Maximum occurring integer in given ranges





Given two arrays **L[]** and **R[]** of size **N** where L[i] and R[i] (**$0 \le L[i], R[i] < 10^6$**)denotes a range of numbers, the task is to find the maximum occurred integer in all the ranges. If more than one such integer exists, print the smallest one.

**Examples:**

***Input:*** *L[] = {1, 4, 3, 1}, R[] = {15, 8, 5, 4}*
***Output:*** *4*

***Input:*** *L[] = {1, 5, 9, 13, 21}, R[] = {15, 8, 12, 20, 30}*
***Output:*** *5*
***Explanation:*** *Numbers having maximum occurrence i.e 2 are 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15. The smallest number among all are 5.*

```
int maximumOccurredElement(int L[], int R[], int n) {
    int arr[MAX];
    memset(arr, 0, sizeof arr);
    // Adding +1 at Li index and subtracting 1 at Ri+1 index.
    int maxi = -1;
    for (int i = 0; i < n; i++) {
        arr[L[i]] += 1;
        arr[R[i] + 1] -= 1;
        if (R[i] > maxi) {
            maxi = R[i];
        }
    }

    // Finding prefix sum and index having maximum prefix sum.
    int msum = arr[0], ind;
    for (int i = 1; i < maxi + 1; i++) {
        arr[i] += arr[i - 1];
        if (arr[i] > msum)  {
            msum = arr[i];
            ind = i;
        }
    }

    return ind;
}
```

# Minimum cost for acquiring all coins with k extra coins allowed with every coin





You are given a list of N coins of different denominations. You can pay an amount equivalent to any 1 coin and can acquire that coin. In addition, once you have paid for a coin, we can choose at most K more coins and can acquire those for free. The task is to find the minimum amount required to acquire all the N coins for a given value of K.

**Examples :**

```
Input : coin[] = {100, 20, 50, 10, 2, 5},
        k = 3
Output : 7

Input : coin[] = {1, 2, 5, 10, 20, 50},
        k = 3
Output : 3
```

As per the question, we can see that at a cost of 1 coin, we can acquire at most K+1 coins. Therefore, in order to acquire all the n coins, we will be choosing ceil(n/(k+1)) coins and the cost of choosing coins will be minimum if we choose the smallest ceil(n/(k+1)) ( Greedy approach). The smallest ceil(n/(k+1)) coins can be found by simply sorting all the N values in increasing order.
If we should check for time complexity (n log n) is for sorting element and (k) is for adding the total amount. So, finally Time Complexity: O(n log n).

```
#include<bits/stdc++.h>
using namespace std;
// brute force nlogn + k
int minCost(int coin[], int n, int k) {
    sort(coin, coin + n);
    int coins_needed = ceil(1.0 * n / (k + 1));
    int ans = 0;
    for (int i = 0; i <= coins_needed - 1; i++)
        ans += coin[i];
    return ans;
}
// Converts coin[] to prefix sum array to remove O(K) loop as it can be huge
void preprocess(int coin[], int n) {
    sort(coin, coin + n);
    // Maintain prefix sum array
    for (int i = 1; i <= n - 1; i++)
        coin[i] += coin[i - 1];
}
int minCost(int coin[], int n, int k) {
    int coins_needed = ceil(1.0 * n / (k + 1));
    return coin[coins_needed - 1];
}

```

# Random number generator in arbitrary probability distribution fashion





Given n numbers, each with some frequency of occurrence. Return a random number with probability proportional to its frequency of occurrence.

**Example:**

```
Let following be the given numbers.
  arr[] = {10, 30, 20, 40}

Let following be the frequencies of given numbers.
  freq[] = {1, 6, 2, 1}

The output should be
  10 with probability 1/10
  30 with probability 6/10
  20 with probability 2/10
  40 with probability 1/10
```

```cpp
#include <bits/stdc++.h>
using namespace std;
int findCeil(int arr[], int r, int l, int h) {
    int mid;
    while (l < h) {
        mid = l + ((h - l) >> 1);
        r > arr[mid] ? (l = mid + 1) : (h = mid);
    }
    return (arr[l] >= r) ? l : -1;
}

int myRand(int arr[], int freq[], int n) {
    int prefix[n], i;
    prefix[0] = freq[0];
    for (i = 1; i < n; ++i)
        prefix[i] = prefix[i - 1] + freq[i];

    // prefix[n-1] is sum of all frequencies.
    // Generate a random number with value from 1 to this sum
    int r = (rand() % prefix[n - 1]) + 1;

    // Find index of ceiling of r in prefix array
    int indexc = findCeil(prefix, r, 0, n - 1);
    return arr[indexc];
}
```





# Maximum length of subarray such that sum of the subarray is even

souradeep



Given an array of N elements. The task is to find the length of the longest subarray such that sum of the subarray is even.

**Examples:**

```
Input : N = 6, arr[] = {1, 2, 3, 2, 1, 4}
Output : 5
Explanation: In the example the subarray
in range [2, 6] has sum 12 which is even,
so the length is 5.
Input : N = 4, arr[] = {1, 2, 3, 2}
Output : 4
```

Recommended: Please try your approach on *{IDE}* first, before moving on to the solution.

### Naive Approach

The idea is to find all subarrays and then find those subarrays whose sum of elements are even. After that choose the longest length of those subarrays.

Steps to implement-

- Declare a variable ans with value 0 to store the final answer
- Run two loops to find all subarrays
- Find the sum of all elements of the subarray
- When the sum of all elements of the subarray is even
  - Then update ans as the maximum of ans and the length of that subarray

```
<script>
function maxLength(a, n) {
    let sum = 0, len = 0;
    for (let i = 0; i < n; i++)
        sum += a[i];
    if (sum % 2 == 0) // total sum is already even
        return n;

    // Find an index i, a[i] is odd and compare length of both
    //  halfs excluding a[i] to find max length subarray
    for (let i = 0; i < n; i++) {
        if (a[i] % 2 == 1)
            len = Math.max(len, Math.max(n - i - 1, i));
// in 2, 3, 2, 1, 4, 1 (max(6-5-1, 5)) = 5 so this handles both sides
    }

    return len;
}

    // Driver Code
    let a = [ 1, 2, 3, 2 ];
    let n = a.length;
    document.write(maxLength(a, n) + "<br>");

</script>
```



# Maximum occurring integer in given ranges





Given two arrays **L[]** and **R[]** of size **N** where L[i] and R[i] **(0 ≤ L[i], R[i] < $10^6$)**denotes a range of numbers, the task is to find the maximum occurred integer in all the ranges. If more than one such integer exists, print the smallest one.

**Examples:**

***Input:*** *L[] = {1, 4, 3, 1}, R[] = {15, 8, 5, 4}*
***Output:*** *4*

***Input:*** *L[] = {1, 5, 9, 13, 21}, R[] = {15, 8, 12, 20, 30}*
***Output:*** *5*
***Explanation:*** *Numbers having maximum occurrence i.e 2 are 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15. The smallest number among all are 5.*

Follow the below steps to Implement the idea:

- The idea is similar to merge sort, divide the array into two equal or almost equal halves in each step until the base case is reached.
- Create a function merge that counts the number of inversions when two halves of the array are merged,
  - Create two indices i and j, i is the index for the first half, and j is an index of the second half.
  - If a[i] is greater than a[j], then there are (mid – i) inversions because left and right subarrays are sorted, so all the remaining elements in left-subarray (a[i+1], a[i+2] … a[mid]) will be greater than a[j].
- Create a recursive function to divide the array into halves and find the answer by summing the number of inversions in the first half, the number of inversions in the second half and the number of inversions by merging the two.
  - The base case of recursion is when there is only one element in the given half.
- Print the answer.

```
function mergeAndCount(arr, l, m, r){
    let left = [];
    for(let i = l; i < m + 1; i++)
        left.push(arr[i]);
    let right = [];
    for(let i = m + 1; i < r + 1; i++)
        right.push(arr[i]);
    let i = 0, j = 0, k = l, swaps = 0;
    while (i < left.length && j < right.length) {
        if (left[i] <= right[j])
            arr[k++] = left[i++];
        else {
            arr[k++] = right[j++];
            swaps += (m + 1) - (l + i);
        }
    }
    while (i < left.length)
        arr[k++] = left[i++];
    while (j < right.length)
        arr[k++] = right[j++];
    return swaps;
}
function mergeSortAndCount(arr, l, r) {
    let count = 0;
    if (l < r) {
        let m = Math.floor((l + r) / 2);
        count += mergeSortAndCount(arr, l, m);
        count += mergeSortAndCount(arr, m + 1, r);
        count += mergeAndCount(arr, l, m, r);
    }
    return count;
}
```

# Length of longest Subarray with equal number of odd and even elements

rupesh_rao



Read Discuss Courses Practice

Given an integer array **arr[]**, the task is to find the length of the longest subarray with an equal number of odd and even elements.

**Examples:**

***Input:*** *arr[] = {1, 2, 1, 2}*
***Output:*** *4*
***Explanation:***
*Subarrays in the given array are –*
*{{1}, {1, 2}, {1, 2, 1}, {1, 2, 1, 2}, {2}, {2, 1}, {2, 1, 2}, {1}, {1, 2}, {2}}*
*where the length of the longest subarray with an equal number of even and odd elements is 4 – {1, 2, 1, 2}*

***Input:*** *arr[] = {12, 4, 7, 8, 9, 2, 11, 0, 2, 13}*
***Output:*** *8*

Treat even as 1 and odd as 0 now this qns reduces to max len of subarray with equal 0s and 1s

```
<script>
function maxSubarrayLength(A, N) {
    let maxLen = 0;
    let curr_sum = 0;
    let hash = new Map();
    hash.set(0, -1);
    for (var i = 0; i < N; i++) {
        if (A[i] % 2 == 0)
            curr_sum -= 1;
        else
            curr_sum += 1;

        // If curr_sum already exists in map we have a subarray with 0 sum
        // i.e, equal number of odd and even number
        if (hash.has(curr_sum))
            maxLen = Math.max(maxLen, i - hash.get(curr_sum));
        else
            hash.set(curr_sum, i);
    }

    return maxLen;
}

// Driver Code
var arr = [12, 4, 7, 8, 9, 2, 11, 0, 2, 13 ];
var n = arr.length;
document.write( maxSubarrayLength(arr, n));
```

# Maximize sum of an Array by flipping sign of all elements of a single subarray

hemanthswarna1506





Given an array **arr[]** of **N** integers, the task is to find the **maximum sum** of the array that can be obtained by **flipping signs** of any subarray of the given array at most once.

**Examples:**

return Total sum - 2*minSubarraySum

eg 1, -1, 1 sum is 1 but if we remove -1
our sum would be 2 ie currsum - 2*removed

and invert -1 to +1 then its 3

***Input:*** *arr[] = {-2, 3, -1, -4, -2}*
***Output:*** *8*
***Explanation:***
*Flipping the signs of subarray {-1, -4, -2} modifies the array to {-2, 3, 1, 4, 2}. Therefore, the sum of the array = -2 + 3 + 1 + 4 + 2 = 8, which is the maximum possible.*

***Input:*** *arr[] = {1, 2, -10, 2, -20}*
***Output:*** *31*
***Explanation:***
*Flipping the signs of subarray {-10, 2, -20} modifies the array to {1, 2, 10, -2, 20}. Therefore, the sum of the array = 1 + 2 + 10 – 2 + 20 = 31, which is the maximum possible.*

```
<script>
function maxsum(l,n){
    let total_sum = 0;
    for (let i = 0; i < n; i++)
        total_sum += l[i];

    // kadane's algorithm to find the minimum subarray sum
    let current_sum=0
    let minimum_sum=0
    for(let i of l){
        current_sum += i
        minimum_sum = Math.min(minimum_sum,current_sum)
        current_sum = Math.min(0, current_sum);
    }
    return Math.max(total_sum, total_sum-2*minimum_sum)
}
    // driver code
    let l = [-2, 3, -1, -4, -2]
    let n = l.length
    document.write(maxsum(l,n))

</script>
```

# Equilibrium index of an array





Given a sequence **arr[]** of size **n**, Write a function ***int equilibrium(int[] arr, int n)*** that returns an equilibrium index (if any) or -1 if no equilibrium index exists.

*The **equilibrium index of an array** is an index such that the sum of elements at lower indexes is equal to the sum of elements at higher indexes.*

**Examples:**

***Input***: *A[] = {-7, 1, 5, 2, -4, 3, 0}*
***Output***: *3*
*3 is an equilibrium index, because:*
*A[0] + A[1] + A[2] = A[4] + A[5] + A[6]*

***Input***: *A[] = {1, 2, 3}*
***Output***: *-1*

Naive approach: To solve the problem follow the below idea:

*Use two loops. The Outer loop iterates through all the element and inner loop finds out whether the current index picked by the outer loop is equilibrium index or not*

```
<script>
function equilibrium(a, n) {
    if (n == 1)
        return (0);
    var forward = new Array(0);
    var rev = new Array(0);
    for (let i = 0; i < n; i++) {
        if (i)
          forward[i] = forward[i - 1] + a[i];
        else
          forward[i] = a[i];
    }
    for (let i = n - 1; i > 0; i--) {
        if (i <= n - 2)
            rev[i] = rev[i + 1] + a[i];
        else
            rev[i] = a[i];
    }

    for (let i = 0; i < n; i++) {
        if (forward[i] == rev[i]) {
            return i;
        }
    }
    return -1;
}
```

# Maximum circular subarray sum



Read Discuss(270+) Courses Practice

Given a **circular array** of size n, find the **maximum subarray sum** of the non-empty subarray.

**Examples:**

***Input:*** *arr[] = {8, -8, 9, -9, 10, -11, 12}*
***Output:*** *22*
***Explanation:*** *Subarray 12, 8, -8, 9, -9, 10 gives the maximum sum, that is **22**.*

***Input:*** *arr[] = {10, -3, -4, 7, 6, 5, -4, -1}*
***Output:*** *23*
***Explanation:*** *Subarray 7, 6, 5, -4, -1, 10 gives the maximum sum, that is **23**.*

***Input:*** *arr[] = {-1, 40, -14, 7, 6, 5, -4, -1}*
***Output:*** *52*
***Explanation:*** *Subarray 7, 6, 5, -4, -1, -1, 40 gives the maximum sum, that is **52**.*

```javascript
/**
 * @param {number[]} arr
 * @param {number} N
 * @returns {number}
*/
class Solution {
    //Function to find maximum circular subarray sum.
    circularSubarraySum(arr, N) {
        let sum = arr[0];
        let mxSum = arr[0];
        let mnSum = arr[0];
        let mx = arr[0];
        let mn = arr[0];

        for (let i = 1; i < N; i++) {
            mx = Math.max(arr[i], mx + arr[i]);
            mxSum = Math.max(mxSum, mx);
            mn = Math.min(arr[i], mn + arr[i]);
            mnSum = Math.min(mnSum, mn);
            sum += arr[i];
        }

        return mxSum > 0 ? Math.max(mxSum, sum - mnSum) : mxSum;
    }
}
```



Facebook 110 Amazon 87 Google 28



## 56. Merge Intervals

Medium 19923 674 Add to List Share

Given an array of `intervals` where `intervals[i] = [start_i, end_i]`, merge all overlapping intervals, and return *an array of the non-overlapping intervals that cover all the intervals in the input.*

**Example 1:**

```
Input: intervals = [[1,3],[2,6],[8,10],
[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: Since intervals [1,3] and [2,6]
overlap, merge them into [1,6].
```

**Example 2:**

```
Input: intervals = [[1,4],[4,5]]
Output: [[1,5]]
Explanation: Intervals [1,4] and [4,5] are
considered overlapping.
```



```javascript
/**
 * @param {number[][]} intervals
 * @return {number[][]}
 */
var merge = function(intervals) {
    intervals.sort((a, b) => a[0] - b[0]);
    let n = intervals.length;
    let ans = [];
    ans.push(intervals[0]);

    for (let i = 1; i < n; i++) {
        if (ans[ans.length-1][1] >= intervals[i][0]) {
            ans[ans.length-1][1] = Math.max(ans[ans.length-1][1], intervals[i][1]);
        } else {
            ans.push(intervals[i]);
        }
    }

    return ans;
};
```

Google 11 | LinkedIn 9 | Facebook 5

## 57. Insert Interval

Medium | 8663 | 617 | Add to List | Share

You are given an array of non-overlapping intervals `intervals` where `intervals[i] = [start_i, end_i]` represent the start and the end of the `i^th` interval and `intervals` is sorted in ascending order by `start_i`. You are also given an interval `newInterval = [start, end]` that represents the start and end of another interval.

Insert `newInterval` into `intervals` such that `intervals` is still sorted in ascending order by `start_i` and `intervals` still does not have any overlapping intervals (merge overlapping intervals if necessary).

Return `intervals` *after the insertion*.



**Example 1:**

```
Input: intervals = [[1,3],[6,9]],
newInterval = [2,5]
Output: [[1,5],[6,9]]
```

**Example 2:**

```
Input: intervals = [[1,2],[3,5],[6,7],
[8,10],[12,16]], newInterval = [4,8]
Output: [[1,2],[3,10],[12,16]]
Explanation: Because the new interval [4,8]
overlaps with [3,5],[6,7],[8,10].
```



```javascript
var merge = function(intervals) {
    intervals.sort((a, b) => a[0] - b[0]);
    let n = intervals.length;
    let ans = [];
    ans.push(intervals[0]);

    for (let i = 1; i < n; i++) {
        if (ans[ans.length - 1][1] >= intervals[i][0]) {
            ans[ans.length - 1][1] = Math.max(ans[ans.length - 1][1], intervals[i][1]);
        } else {
            ans.push(intervals[i]);
        }
    }

    return ans;
};

var insert = function(intervals, newInterval) {
    let i = 0, n = intervals.length;
    let res = [];
    // Pushing intervals that don't interfere
    while (i < n && newInterval[0] > intervals[i][1])
        res.push(intervals[i++]);

    res.push(newInterval);

    while (i < n)
        res.push(intervals[i++]);
    // merge(res); // aise modify nahi hoga because new array return ho raha hai

    res = merge(res)

    return res;
};
```

Optimised

```javascript
/**
 * @param {number[][]} intervals
 * @param {number[]} newInterval
 * @return {number[][]}
 */
// one brute force can be push newInterval where first it start interfare
// then call merge interval

var insert = function(intervals, newInterval) {
    let i = 0;
    let n = intervals.length;

    let res = [];
    // Pushing intervals that don't interfere
    while (i < n && newInterval[0] > intervals[i][1])
        res.push(intervals[i++]);

    // Merging [0] will be minfirst and [1] will be max sec
    while (i < n && newInterval[1] >= intervals[i][0]) {
        newInterval[0] = Math.min(newInterval[0], intervals[i][0]);
        newInterval[1] = Math.max(newInterval[1], intervals[i][1]);
        i++;
    }

    res.push(newInterval);
    while (i < n)
        res.push(intervals[i++]);       // Push remaining intervals

    return res;
};
```



**Intuition:**

1. Minimum number of intervals to remove .
2. Which is nothing but maximum number of intervals we can should keep.
3. Then it comes under Maximum Meeting we can attend.



▶ **In Detail**

**Explanation:**

Imagine we have a set of meetings, where each meeting is represented by an interval [start_time, end_time]. The goal is to find the maximum number of non-overlapping meetings we can attend.

1. **Sorting by end times ( `cmp` function):**
   The function first sorts the intervals based on their end times in ascending order using the custom comparator `cmp` . This sorting is crucial because it allows us to prioritize intervals that finish early, giving us more opportunities to accommodate additional meetings later on.
2. **Initializing variables:**
   The function initializes two variables, `prev` and `count` . The `prev` variable is used to keep track of the index of the last processed interval, and `count` is used to store the number of non-overlapping meetings found so far. We start `count` with 1 because the first interval is considered non-overlapping with itself.
3. **Greedy approach:**
   The function uses a greedy approach to find the maximum number of non-overlapping meetings. It iterates through the sorted intervals starting from the second interval (index 1) because we've already counted the first interval as non-overlapping. For each interval at index `i` , it checks if the start time of the current interval ( `intervals[i][0]` ) is greater than or equal to the end time of the previous interval ( `intervals[prev][1]` ). If this condition is true, it means the current interval does not overlap with the previous one, and we can safely attend this meeting. In that case, we update `prev` to the current index `i` and increment `count` to reflect that we have attended one more meeting.
4. **Return result:**
   Finally, the function returns the number of intervals that need to be removed to make the remaining intervals non-overlapping. Since we want to maximize the number of meetings we can attend, this value is calculated as `n - count` , where `n` is the total number of intervals.

```cpp
class Solution {
public:
    static bool cmp(vector<int>& a, vector<int>& b){
        return a[1] < b[1];
    }

    int eraseOverlapIntervals(vector<vector<int>>& intervals) {
        int n = intervals.size();
        sort(intervals.begin(), intervals.end(), cmp);

        int prev = 0;
        int count = 1;

        for(int i = 1; i < n; i++){
            if(intervals[i][0] >= intervals[prev][1]){
                prev = i;
                count++;
            }
        }
        return n - count;
    }
};
```

```cpp
class Solution {
public:
    static bool comp(vector<int> &a, vector<int> &b) {
        if(a[1] == b[1])
            return a[0] > b[0];
        return a[1] < b[1];
    }
    int eraseOverlapIntervals(vector<vector<int>>& intervals) {
        sort(intervals.begin(), intervals.end(), comp);
        int n = intervals.size();
        int count = 0;
        // [[1,3],[1,2],[2,3],[3,4]]
        vector<int> prev = intervals[0];
        for(int i=1; i<n; i++) {
            if(prev[1] > intervals[i][0])
                count++;
            else
                prev = intervals[i];

        }

        return count;
    }
};
```



# 1353. Maximum Number of Events That Can Be Attended



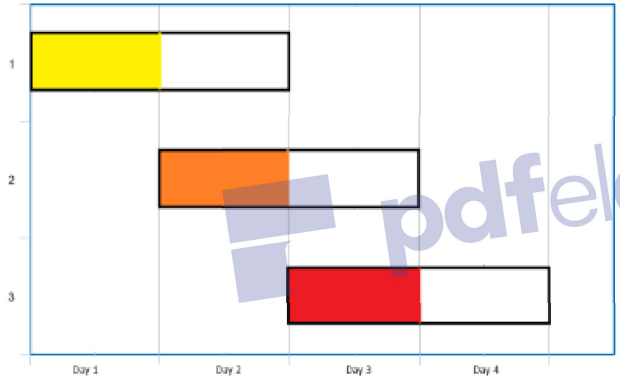
Medium 2682 360 Add to List Share

You are given an array of `events` where `events[i] = [startDay_i, endDay_i]`. Every event `i` starts at `startDay_i` and ends at `endDay_i`.

You can attend an event `i` at any day `d` where `startTime_i <= d <= endTime_i`. You can only attend one event at any time `d`.

Return *the maximum number of events you can attend.*

**Example 1:**




```
Input: events = [[1,2],[2,3],[3,4]]
Output: 3
Explanation: You can attend all the three events.
One way to attend them all is as shown.
Attend the first event on day 1.
Attend the second event on day 2.
Attend the third event on day 3.
```

**Example 2:**

```
Input: events= [[1,2],[2,3],[3,4],[1,2]]
Output: 4
```

we have total 4 days:
Attend [1,2] event on day 1
Attend [1,2] event on day 2
Attend [2,3] event on day 3
Attend [3,4 ] event on day 4

**Constraints:**

- `1 <= events.length <= 10^5`
- `events[i].length == 2`
- `1 <= startDay_i <= endDay_i <= 10^5`

#1. Sort the events based on starting day of the event
#2. Now once you have this sorted events, every day check what are the events that can start today
#3. for all the events that can be started today, keep their ending time in heap.
# - Wait why we only need ending times ?
# i) from today onwards, we already know this event started in the past and all we need to know is when this event will finish
# ii) Also, another key to this algorithm is being greedy, meaning I want to pick the event which is going to end the soonest.
# - So how do we find the event which is going to end the soonest?
# i) brute force way would be to look at all the event's ending time and find the minimum, this is probably ok for 1 day but as we can only attend 1 event a day,
# we will end up repeating this for every day and that's why we can utilize heap(min heap to be precise) to solve the problem of finding the event with earliest ending time
#4. There is one more house cleaning step, the event whose ending time is in the past, we no longer can attend those event
#5. Last but very important step, Let's attend the event if any event to attend in the heap.

```cpp
class Solution {
public:
    int maxEvents(vector<vector<int>>& events) {
        sort(events.begin(), events.end());
        int total_days = 0;
        for (const auto& event: events) {
            total_days = max(total_days, event[1]);
        }

        int ith = 0;
        int num_events_attended = 0;
        priority_queue<int, vector<int>, greater<int>> min_heap;

        for (int day = 1; day <= total_days; day++) {
            // pushing all the events that start today
            while (ith < events.size() && events[ith][0] == day) {
                min_heap.push(events[ith][1]);
                ith++;
            }

            // whose end date was before today cant be attended
            while (!min_heap.empty() && min_heap.top() < day) {
                min_heap.pop();
            }

            // If any event that can be attended today, let's attend it
            if (!min_heap.empty()) {
                min_heap.pop(); // 2 2 3 4
                num_events_attended++;
            }
        }

        return num_events_attended;
    }
};
```

# 1827. Minimum Operations to Make the Array Increasing

Easy 1018 51 Add to List Share

You are given an integer array `nums` (**0-indexed**). In one operation, you can choose an element of the array and increment it by `1`.

- For example, if `nums = [1,2,3]`, you can choose to increment `nums[1]` to make `nums = [1,3,3]`.

Return *the **minimum** number of operations needed to make* `nums` ***strictly increasing***.

An array `nums` is **strictly increasing** if `nums[i] < nums[i+1]` for all `0 <= i < nums.length - 1`. An array of length `1` is trivially strictly increasing.

**Example 1:**

```
Input: nums = [1,1,1]
Output: 3
Explanation: You can do the following operations:
1) Increment nums[2], so nums becomes [1,1,2].
2) Increment nums[1], so nums becomes [1,2,2].
3) Increment nums[2], so nums becomes [1,2,3].
```

**Example 2:**

```
Input: nums = [1,5,2,4,1]
Output: 14
```

1 5 6 7 8
4+3+7 opr

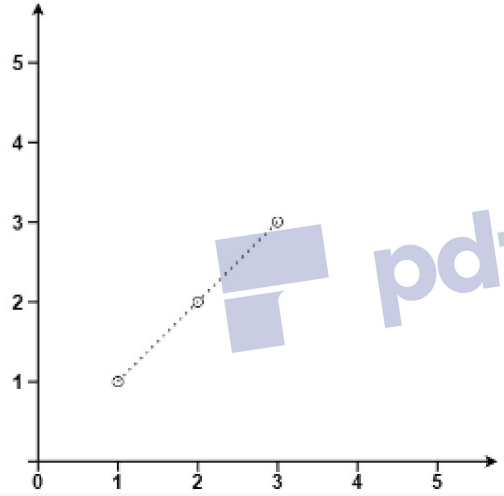
```cpp
class Solution {
public:
    int minOperations(vector<int>& arr) {
        int last = 0, opr = 0;
        // just track last elm
        for(int elm: arr) {
            opr += max(0, last - elm + 1);
            last = max(elm, last + 1);
        }

        return opr;
    }
};
```

# 149. Max Points on a Line

Hard 3758 420 Add to List

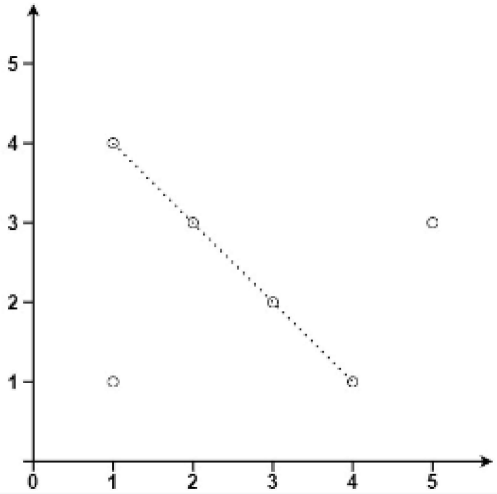
Given an array of `points` where `points[i] = [xi, yi]` represents a point on the **X-Y** plane, return *the maximum number of points that lie on the same straight line.*

**Example 1:**



```
Input: points = [[1,1],[2,2],
[3,3]]
Output: 3
```

**Example 2:**



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



Approach

Idea is very simple

1. We will select point and for every selected point we will calculate the slope wr.t to another point and store into the map
2. In map we will store <slope,that selected point>

- for example we have point :-[[1,1],[2,2],[3,3].....
  we pick point [1,1]and for that [1,1] we will calculate the slope
  w.r.t [2,2] ,[3,3]..... and store like this-
  for understanding purpose i'm writing
  mp[slope1,[1,1]]
  mp[slope2,[1,1]]
  mp[slope1,[1,1]].....
  This type we will store the slope for [2,2] , [3,3].....

3. Ater storing the slope into map we will calculate the max point for that slope
   For example after storing the slope map can look like this
   [2/3,3]
   [INF,1]//for perpendicular case
   [1/3,2]...
4. find the max number of points having the same slope w.r.t. that selected point :-means which slope have maximum point
   :- here 2/3 have 3 point so our answer will be 3+1(we are adding that point as well)

```
var maxPoints = function(points) {
    let maxi = 0;
    for ( let i = 0; i < points.length; i++ ) {
        const mp = new Map();
        const [x1, y1] = [...points[i]]; // ... JS spread
        let currMax = 1, slope = 0;

        for ( let j = i + 1; j < points.length; j++ ) {
            const [x2, y2] = [...points[j]];

            if ( x1 === x2 )
                slope = Number.MAX_VALUE; // denominator is 0
            else
                slope = (y2 - y1) / (x2 - x1);

            if (!mp.has(slope))
                mp.set(slope, 2); // new slope 2 pts

            else
                mp.set(slope, mp.get(slope) + 1); // if slope already exits

            currMax = Math.max(currMax, mp.get(slope));
        }

        maxi = Math.max(maxi, currMax);
    }

    return maxi;
};
```

# Maximize sum of an Array by flipping sign of all elements of a single subarray



hemanthswarna1506



Given an array **arr[]** of **N** integers, the task is to find the **maximum sum** of the array that can be obtained by **flipping signs** of any subarray of the given array at most once.

**Examples:**

return Total sum - 2*minSubarraySum

***Input:*** *arr[] = {-2, 3, -1, -4, -2}*
***Output:*** *8*
***Explanation:***

eg 1, -1, 1 sum is 1 but if we remove -1
our sum would be 2 ie currsum - 2*removed

*Flipping the signs of subarray {-1, -4, -2} modifies the array to {-2, 3, 1, 4, 2}. Therefore, the sum of the array = -2 + 3 + 1 + 4 + 2 = 8, which is the maximum possible.*

***Input:*** *arr[] = {1, 2, -10, 2, -20}*
***Output:*** *31*
***Explanation:***
*Flipping the signs of subarray {-10, 2, -20} modifies the array to {1, 2, 10, -2, 20}. Therefore, the sum of the array = 1 + 2 + 10 – 2 + 20 = 31, which is the maximum possible.*

```cpp
void nextPermutation(vector<int>& arr) {
    int n = arr.size();
    if(n==1) return;

    // 4 1 5 3 2  =>  4 2 1 3 5

    int idx1;       // step1 find downfall
    for(int i=n-2; i>=0; i--){
        if(arr[i] < arr[i+1]){
            idx1 = i;
            break;      // vImp we'll break out otherwise idx2 will get updated in each iteration
        }
    }
    /* A void function cannot return any values.
       But we can use it to return an "statement" .
       It indicates that the function is 'terminated' .
       It increases the readability of code. */

    if(idx1<0) return reverse(arr.begin(), arr.end());

    int idx2;       // find just greater
     for(int i=n-1; i>=0; i--){
        if(arr[i] > arr[idx1]){
            idx2 = i;
            break;      // vImp we'll break out otherwise idx2 will get updated in each iteration
        }
    }
    // step 3 swap both
    swap(arr[idx1], arr[idx2]);

    // sort fron idx1+1 till end to get smallest combination

    reverse(arr.begin() + idx1 + 1, arr.end());

}
```



# Maximum Index



Medium Accuracy: 13.46% Submissions: 195K+ Points: 4



Given an array **arr[]** of **n** non-negative integers. The task is to find the maximum of **j - i (i<=j)** subjected to the constraint of **arr[i] <= arr[j]**.

**Example 1:**

**Input:**
n = 9
arr[] = {34, 8, 10, 3, 2, 80, 30, 33, 1}
**Output:**
6
**Explanation:**
In the given array arr[1] < arr[7] satisfying the required condition (arr[i] <= arr[j]) thus giving the maximum difference of j - i which is 6(7-1).

**Example 2:**

**Input:**
N = 2
arr[] = {18, 17}
**Output:**
0
**Explanation:**
We can either take i and j as 0 and 0
or we cantake 1 and 1 both give the same result 0.

```cpp
class Solution{
public:
    int maxIndexDiff(int arr[], int n) {
      int ans = -1;
      vector<int> Lmin(n, INT_MAX);
      vector<int> Rmax(n, INT_MIN);

      Lmin[0] = arr[0];
      Rmax[n-1] = arr[n-1];

      for(int i=1; i<n; i++)
          Lmin[i] = min(arr[i], Lmin[i-1]);

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

      int i=0, j=0;
      while(i<n && j<n) {
          if(Lmin[i] <= Rmax[j]){
              ans = max(ans, j-i);
              j++;
          }
          else
              i++;  // move to get smaller maybe
      }
      return ans;
    }
};
```

## 121. Best Time to Buy and Sell Stock

Easy 26841 859 Add to List Share

You are given an array `prices` where `prices[i]` is the price of a given stock on the $i^{th}$ day.

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return `0`.

**Example 1:**

```
Input: prices = [7,1,5,3,6,4]
Output: 5
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.
```

**Example 2:**

```
Input: prices = [7,6,4,3,1]
Output: 0
Explanation: In this case, no transactions are done and the max profit = 0.
```

```javascript
var maxProfit = function(prices) {
    let n = prices.length;
    let profit = 0;
    let minBuy = Number.POSITIVE_INFINITY;

    for(let i=0; i<n; i++) {
        minBuy = Math.min(minBuy, prices[i]);
        profit = Math.max(profit, prices[i] - minBuy);
    }

    return profit;
};
```

## Count pairs with given sum

Easy Accuracy: 31.49% Submissions: 318K+ Points: 2



Given an array of **N** integers, and an integer **K**, find the number of pairs of elements in the array whose sum is equal to **K**.

**Example 1:**

**Input:**
N = 4, K = 6
arr[] = {1, 5, 7, 1}
**Output:** 2
**Explanation:**
arr[0] + arr[1] = 1 + 5 = 6
and arr[1] + arr[3] = 5 + 1 = 6.

**Example 2:**

**Input:**
N = 4, K = 2
arr[] = {1, 1, 1, 1}
**Output:** 6
**Explanation:**
Each 1 will produce sum 2 with any 1.

```
class Solution {
    getPairsCount(arr,n,k){
        //code here
        const mp = new Map();
        let count = 0;

        for(let i=0; i<n; i++) {
            if(mp.has(k-arr[i]))
                count += mp.get(k-arr[i]);

            mp.set(arr[i], mp.get(arr[i]) + 1 || 1);
        }

        return count;
    }
}
```

Given three arrays sorted in increasing order. Find the elements that are common in all three arrays.

**Note:** can you take care of the duplicates without using any additional Data Structure?

**Example 1:**

**Input:**
n1 = 6; A = {1, 5, 10, 20, 40, 80}
n2 = 5; B = {6, 7, 20, 80, 100}
n3 = 8; C = {3, 4, 15, 20, 30, 70, 80, 120}
**Output:** 20 80
**Explanation:** 20 and 80 are the only common elements in A, B and C.

**Your Task:**

You don't need to read input or print anything. Your task is to complete the function **commonElements()** which take the 3 arrays A[], B[], C[] and their respective sizes n1, n2 and n3 as inputs and returns an array containing the common element present in all the 3 arrays in sorted order.

If there are no such elements return an empty array. In this case the output will be printed as -1.

```javascript
class Solution {
    commonElements(A, B, C, n1, n2, n3) {
        let ans = [];
        let i = 0, j = 0, k = 0;

        while (i < n1 && j < n2 && k < n3) {
            if (A[i] === B[j] && B[j] === C[k]) {
                ans.push(A[i]);
                i++;
                j++;
                k++;
                while(A[i-1] == A[i])
                    i++;    // for 3 3 3 type

            } else if (A[i] < B[j]) {
                i++;
            } else if (B[j] < C[k]) {
                j++;
            } else {
                k++;
            }
        }

        return ans;
    }
}
```





## Rearrange array in alternating positive & negative items with O(1) extra space | Set 1



Given an **array of positive** and **negative numbers**, arrange them in an **alternate** fashion such that every positive number is followed by a negative and vice-versa maintaining the **order of appearance**. The number of positive and negative numbers need not be equal. If there are more positive numbers they appear at the end of the array. If there are more negative numbers, they too appear at the end of the array.

**Examples:**

*Input*: arr[] = {1, 2, 3, -4, -1, 4}
*Output*: arr[] = {-4, 1, -1, 2, 3, 4}

*Input*: arr[] = {-5, -2, 5, 2, 4, 7, 1, 8, 0, -8}
*Output*: arr[] = {-5, 5, -2, 2, -8, 4, 7, 1, 8, 0}

This question has been asked in many places (See this and this)

**Naive Approach:**

The above problem can be easily solved if O(n) extra space is allowed.

- We can store the positive values and negative values in two separate data structures.
- We will start filling the original array with alternating negative and positive values in the same order in which it appears in the original array.

It becomes interesting due to the limitations that O(1) extra space and order of appearances.

**Optimal Approach:**

The idea is to process the array from **left to right**. While processing, find the first out-of-place element in the remaining unprocessed array. An element is out of place if it is **negative** and at **odd index (0-based index)**, or if it is **positive** and at **even index (0-based index)**. Once we find an out-of-place element, we find the first element after it with an opposite sign. We right **rotate the subarray** between these two elements (including these two).

**Optimal Approach:**

The idea is to process the array from **left to right**. While processing, find the first out-of-place element in the remaining unprocessed array. An element is out of place if it is **negative** and at **odd index (0-based index)**, or if it is **positive** and at **even index (0-based index)**. Once we find an out-of-place element, we find the first element after it with an opposite sign. We right **rotate the subarray** between these two elements (including these two).

**Illustration:**

*Let the array be arr[] = { -5, -2, 5, 2, 4, 7, 1, 8, 0, -8 }*

***First iteration:***

- *{ -5, **-2**, 5, 2, 4, 7, 1, 8, 0, -8 } , -2 appears on odd index position and is out of place.*
  *We will look for the first element that appears with an opposite sign*
- *{ -5, **-2**, **5**, 2, 4, 7, 1, 8, 0, -8 } => perform rotation of subarray between this two elements*
- *{ -5, **5**, **-2**, 2, 4, 7, 1, 8, 0, -8 }*

***Second iteration:***

- *{ -5, 5, -2, 2, **4**, 7, 1, 8, 0, -8 } , 4 is out of place.*
- *{ -5, 5, -2, 2, **4**, 7, 1, 8, 0, **-8** } =>-8 is of different sign*
- *{ -5, 5, -2, 2, **-8, 4, 7, 1, 8, 0** } => after performing right rotation between 4 to -8*

**Algorithm:**

We will maintain a variable to mark if the element is in its correct position or not. Let the variable be **outofplace**. Initially, it is -1.

- We will iterate over the array
- If **outofplace** is -1, we will check if the current index is out of place.
  - If the current index is out of place we will update the **outofplace** with the index value.
  - Else we will keep the value as it is.
- If the **outofplace** is not -1, we will search for the next index which has a **different sign** than that of the value that is present in **outofplace** position.
  - Now we will pass this two positions to right rotate our array.
  - Update the value of **outofplace** by 2 unit.



Remove Watermark Now

```
function rearrange(arr, n) {
  let outOfPlaceIndex = -1;

  function rightRotate(curIndex) {
    const tmp = arr[curIndex];
    for (let i = curIndex; i > outOfPlaceIndex; i--)
      arr[i] = arr[i - 1];
    arr[outOfPlaceIndex] = tmp;
  }

  for (let currentIndex = 0; currentIndex < n; currentIndex++) {
    if (outOfPlaceIndex >= 0) {
      const isCurPositive = arr[currentIndex] >= 0;
      const isOutOfPlaceNegative = arr[outOfPlaceIndex] < 0;
      const isCurNegative = arr[currentIndex] < 0;
      const isOutOfPlacePositive = arr[outOfPlaceIndex] >= 0;

      if ((isCurPositive && isOutOfPlaceNegative) || (isCurNegative && isOutOfPlacePositive)) {
        rightRotate(currentIndex);
        if (currentIndex - outOfPlaceIndex >= 2)
          outOfPlaceIndex = outOfPlaceIndex + 2;
        else
          outOfPlaceIndex = -1;
      }
    }

    if (outOfPlaceIndex === -1) {
      const isCurPositive = arr[currentIndex] >= 0;
      const isEvenIndex = (currentIndex & 1) === 0;
      const isCurNegative = arr[currentIndex] < 0;
      const isOddIndex = (currentIndex & 1) === 1;

      if ((isCurPositive && isEvenIndex) || (isCurNegative && isOddIndex))
        outOfPlaceIndex = currentIndex;
    }
  }
}
```

```
function rearrange(arr, n) {
  let out = -1;
  function rightRotate(cur) {
    const tmp = arr[cur];
    for (let i = cur; i > out; i--)
      arr[i] = arr[i - 1];
    arr[out] = tmp;
  }

  for (let i = 0; i < n; i++) {
    if (out >= 0) {
      if ((arr[i] >= 0 && arr[out] < 0) || (arr[i] < 0 && arr[out] >= 0)) {
        rightRotate(i);
        out = (i - out >= 2) ? out + 2 : -1;
      }
    }
    if (out === -1 && ((arr[i] >= 0 && i % 2 === 0) || (arr[i] < 0 && i % 2 === 1)))
      out = i;
  }
}
```

Given an array of positive and negative numbers. Find if there is a **subarray** (of size at-least one) with **0 sum**.

**Example 1:**

**Input:**
5
4 2 -3 1 6

**Output:**
Yes

**Explanation:**
2, -3, 1 is the subarray with sum 0.

**Example 2:**

**Input:**
5
4 2 0 1 6

**Output:**
Yes

**Explanation:**
0 is one of the element in the array so there exist a subarray with sum 0.

```
class Solution {
    subArrayExists(arr, n){
        const mp = new Map();
        let sum = 0;
        mp.set(sum, -1);

        for(let i=0; i<n; i++) {
            sum += arr[i];
            if(mp.has(sum))
                return true;
            mp.set(sum, i);
        }

        return false;
    }
}
```

Given an array **Arr[]** that contains **N** integers (may be **positive**, **negative** or **zero**). Find the product of the maximum product subarray.

**Example 1:**

**Input:**
N = 5
Arr[] = {6, -3, -10, 0, 2}
**Output:** 180
**Explanation:** Subarray with maximum product is [6, -3, -10] which gives product as 180.

**Example 2:**

**Input:**
N = 6
Arr[] = {2, 3, 4, 5, -1, 0}
**Output:** 120
**Explanation:** Subarray with maximum product is [2, 3, 4, 5] which gives product as 120.

```
class Solution {
    maxProduct(arr,n){
      let leftProd = BigInt(1);
      let rightProd = BigInt(1);
      let maxProd = BigInt(Number.MIN_SAFE_INTEGER);

      for (let i = 0; i < n; i++) {
         leftProd *= BigInt(arr[i]);
         rightProd *= BigInt(arr[n - 1 - i]);

         maxProd = (maxProd > leftProd )? maxProd : leftProd;
         maxProd = (maxProd > rightProd) ? maxProd : rightProd;

         leftProd = leftProd === BigInt(0) ? BigInt(1) : leftProd;
         rightProd = rightProd === BigInt(0) ? BigInt(1) : rightProd;
      }

      return maxProd;
    }
}
```

Given an array of positive integers. Find the length of the longest sub-sequence such that elements in the subsequence are consecutive integers, the **consecutive numbers can be in any order.**

**Example 1:**

**Input:**
N = 7
a[] = {2,6,1,9,4,5,3}
**Output:**
6
**Explanation:**
The consecutive numbers here are 1, 2, 3, 4, 5, 6. These 6 numbers form the longest consecutive subsquence.

```
class Solution {
    findLongestConseqSubseq(arr, N) {
        let st = new Set();
        for (let i = 0; i < N; i++)
            st.add(arr[i]);

        let mxLen = 0;
        for (let i = 0; i < N; i++) {
            // aisa elm select karo jiska prev na ho
            if (!st.has(arr[i] - 1)) {
                let num = arr[i];
                let currStreak = 1;

                while (st.has(num + 1)) {
                    currStreak++;
                    num = num + 1;
                }

                mxLen = Math.max(mxLen, currStreak);
            }
        }

        return mxLen;
    }
}
```

## 873. Length of Longest Fibonacci Subsequence

Medium 1883 66 Add to List Share

A sequence $x_1, x_2, ..., x_n$ is *Fibonacci-like* if:

- $n >= 3$
- $x_i + x_{i+1} == x_{i+2}$ for all $i + 2 <= n$

Given a **strictly increasing** array `arr` of positive integers forming a sequence, return *the **length** of the longest Fibonacci-like subsequence of* `arr`. If one does not exist, return `0`.

A **subsequence** is derived from another sequence `arr` by deleting any number of elements (including none) from `arr`, without changing the order of the remaining elements. For example, `[3, 5, 8]` is a subsequence of `[3, 4, 5, 6, 7, 8]`.



**Example 1:**

```
Input: arr = [1,2,3,4,5,6,7,8]
Output: 5
Explanation: The longest subsequence that is
fibonacci-like: [1,2,3,5,8].
```

**Example 2:**

```
Input: arr = [1,3,7,11,12,14,18]
Output: 3
Explanation: The longest subsequence that is
fibonacci-like: [1,11,12], [3,11,14] or [7,11,18].
```



```javascript
/
var lenLongestFibSubseq = function(arr) {
    const n = arr.length;
    const st = new Set();
    for(let i=0; i<n; i++) {
        st.add(arr[i]);
    }

    let mx_fib_len = 2;
    for(let i=0; i<n; i++) {
        for(let j=i+1; j<n; j++) {
            let curr_streak = 2;
            let prev_2 = arr[i];
            let prev = arr[j];
            let curr = prev_2 + prev;

            while(st.has(curr)) {
                curr_streak++;
                prev_2 = prev;
                prev = curr;
                curr = prev_2 + prev;
            }
            mx_fib_len = Math.max(mx_fib_len, curr_streak);

        }
    }

    return mx_fib_len > 2 ? mx_fib_len: 0;

};
```

# Buy and Sell a Share at most twice

Medium Accuracy: 50.13% Submissions: 19K+ Points: 4

Share your Interview, Campus or Work Experience to win GFG Swag Kits and much more!

In daily share trading, a buyer buys shares in the morning and sells them on the same day. If the trader is allowed to make at most 2 transactions in a day, the second transaction can only start after the first one is complete (Buy->sell->Buy->sell). The stock prices throughout the day are represented in the form of an array of **prices**.

Given an array **price** of size **N**, find out the **maximum** profit that a share trader could have made.

**Example 1:**

**Input:**
6
10 22 5 75 65 80
**Output:**
87
**Explanation:**
Trader earns 87 as sum of 12, 75
Buy at 10, sell at 22,
Buy at 5 and sell at 80

**Example 2:**

**Input:**
7
2 30 15 10 8 25 80
**Output:**
100
**Explanation:**

Trader earns 100 as sum of 28 and 72
Buy at price 2, sell at 30,
Buy at 8 and sell at 80



```cpp
typedef vector<int> vi;
#define v vector
v<v<vi>> dp;

int helper(int index, int canBuy, int remainingTransactions, int n, vi &prices) {
    if (remainingTransactions == 0 || index == n)
        return 0;

    if (dp[index][canBuy][remainingTransactions] != -1)
        return dp[index][canBuy][remainingTransactions];

    int profit = 0;
    if (canBuy) {
        int buyStock = -prices[index] + helper(index + 1, 0, remainingTransactions, n, prices);
        int dontBuy = helper(index + 1, 1, remainingTransactions, n, prices);
        profit = max(buyStock, dontBuy);
    } else {
        int sellStock = prices[index] + helper(index + 1, 1, remainingTransactions - 1, n, prices);
        int dontSell = helper(index + 1, 0, remainingTransactions, n, prices);
        profit = max(sellStock, dontSell);
    }

    return dp[index][canBuy][remainingTransactions] = profit;
}

int maxProfit(vi &prices) {
    int n = prices.size();
    dp = v<v<vi>>(n, v<vi>(2, vi(3, -1)));
    return helper(0, 1, 2, n, prices);
}
```



```cpp
int maxProfit(vector<int>& price) {
    int n = price.size();
    vector<vector<vector<int>>> dp(n + 1, vector<vector<int>>(2, vector<int>(3, 0)));

    for (int i = n - 1; i >= 0; i--) {
        for (int buy = 0; buy <= 1; buy++) {
            for (int cap = 1; cap <= 2; cap++) {
                if (buy == 1) {
                    dp[i][buy][cap] = max(-price[i] + dp[i + 1][0][cap], 0 + dp[i + 1][1][cap]);
                } else {
                    dp[i][buy][cap] = max(price[i] + dp[i + 1][1][cap - 1], 0 + dp[i + 1][0][cap]);
                }
            }
        }
    }

    return dp[0][1][2];
}
```

```javascript
class Solution {
    /**
     * @param {number[]} price
     * @return {number}
     */
    maxProfit(price) {
        const n = price.length;
        const dp = new Array(n + 1).fill(null).map(() => new Array(2).fill(null).map(() => new Array(3).fill(0)));

        for (let i = n - 1; i >= 0; i--) {
            for (let buy = 0; buy <= 1; buy++) {
                for (let cap = 1; cap <= 2; cap++) {
                    if (buy === 1) {
                        dp[i][buy][cap] = Math.max(-price[i] + dp[i + 1][0][cap], 0 + dp[i + 1][1][cap]);
                    } else {
                        dp[i][buy][cap] = Math.max(price[i] + dp[i + 1][1][cap - 1], 0 + dp[i + 1][0][cap]);
                    }
                }
            }
        }

        return dp[0][1][2];
    }
}
```

```javascript
function helper(ind, buy, cap, n, price, dp) {
    if (cap === 0) return 0;
    if (ind === n) return 0;
    if (dp[ind][buy][cap] !== -1) return dp[ind][buy][cap];

    if (buy) {
        // means we are open to buy stock
        return (dp[ind][buy][cap] = Math.max(-price[ind] + helper(ind + 1, 0, cap, n, price, dp),
                                             0 + helper(ind + 1, 1, cap, n, price, dp)
        ));
    } else {
        // means we are open to sell
        return (dp[ind][buy][cap] = Math.max(price[ind] + helper(ind + 1, 1, cap - 1, n, price, dp),
                                             0 + helper(ind + 1, 0, cap, n, price, dp)
        ));
    }
}

function maxProfit(price) {
    const n = price.length;
    /** @type {vvv} */
    const dp = new Array(n).fill(null).map(() => new Array(2).fill(null).map(() => new Array(3).fill(-1)));
    return helper(0, 1, 2, n, price, dp);
}
```

# 1353. Maximum Number of Events That Can Be Attended

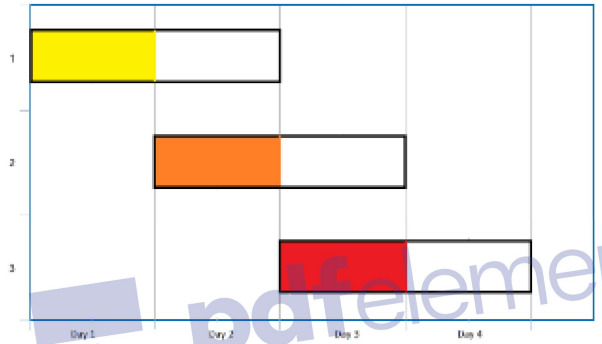
Medium 2685 361 Add to List Share

You are given an array of `events` where `events[i] = [startDay_i, endDay_i]`. Every event `i` starts at `startDay_i` and ends at `endDay_i`.

You can attend an event `i` at any day `d` where `startTime_i <= d <= endTime_i`. You can only attend one event at any time `d`.

Return *the maximum number of events you can attend.*

**Example 1:**



```
Input: events = [[1,2],[2,3],[3,4]]
Output: 3
Explanation: You can attend all the three events.
One way to attend them all is as shown.
Attend the first event on day 1.
Attend the second event on day 2.
Attend the third event on day 3.
```

**Example 2:**

```
Input: events= [[1,2],[2,3],[3,4],[1,2]]
Output: 4
```

```javascript
var maxEvents = function(events) {
    events.sort((a, b) => a[0] - b[0]);
    let totalDays = 0;
    for (const event of events) {
        totalDays = Math.max(totalDays, event[1]);
    }

    let ith = 0;
    let numEventsAttended = 0;
    const minHeap = new MinHeap();

    for (let day = 1; day <= totalDays; day++) {
        // pushing all the events that start today
        while (ith < events.length && events[ith][0] === day) {
            minHeap.push(events[ith][1]);
            ith++;
        }

        // whose end date was before today cant be attended
        while (!minHeap.isEmpty() && minHeap.top() < day) {
            minHeap.pop();
        }

        // If any event that can be attended today, let's attend it
        if (!minHeap.isEmpty()) {
            minHeap.pop();
            numEventsAttended++;
        }
    }

    return numEventsAttended;
};

// Implementation of MinHeap for the priority queue
class MinHeap {↔}
```

```javascript
// Implementation of MinHeap for the priority queue
class MinHeap {
    constructor() {
        this.heap = [];
    }
    push(value) {
        this.heap.push(value);
        this.bubbleUp(this.heap.length - 1);
    }
    pop() {
        if (this.heap.length === 0) return;
        this.swap(0, this.heap.length - 1);
        this.heap.pop();
        this.bubbleDown(0);
    }
    top() {
        if (this.heap.length === 0) return;
        return this.heap[0];
    }
    isEmpty() {
        return this.heap.length === 0;
    }
    bubbleUp(index) {
        while (index > 0) {
            const parentIndex = Math.floor((index - 1) / 2);
            if (this.heap[parentIndex] > this.heap[index]) {
                this.swap(parentIndex, index);
                index = parentIndex;
            } else {
                break;
            }
        }
    }
    bubbleDown(index) {
        const n = this.heap.length;
        while (index < n) {
            const leftChild = 2 * index + 1;
            const rightChild = 2 * index + 2;
            let smallest = index;

            if (leftChild < n && this.heap[leftChild] < this.heap[smallest]) {
                smallest = leftChild;
            }
            if (rightChild < n && this.heap[rightChild] < this.heap[smallest]) {
                smallest = rightChild;
            }

            if (smallest !== index) {
                this.swap(smallest, index);
                index = smallest;
            } else {
                break;
            }
        }
    }
```

```
string isSubset(int a[], int b[], int n, int m) {
    unordered_map<int, int> mp;
    for(int i=0; i<n; i++)
        mp[a[i]]++;

    for(int i=0; i<m; i++){
        if(mp.find(b[i]) == mp.end())
            return "No";
        else {
            mp[b[i]]--;
            if(mp[b[i]] == 0)
                mp.erase(b[i]);
        }
    }

    return "Yes";
}
```

```
class Solution {
    isSubset(a1, a2, n, m) {
        const mp = new Map();
        for (let i = 0; i < n; i++) {
            if (mp.has(a1[i])) {
                mp.set(a1[i], mp.get(a1[i]) + 1 || 1);
            }
        }

        for (let i = 0; i < m; i++) {
            if (!mp.has(a2[i])) {
                return "No";
            } else {
                mp.set(a2[i], mp.get(a2[i]) - 1);
                if (mp.get(a2[i]) === 0) {
                    mp.delete(a2[i]);
                }
            }
        }

        return "Yes";
    }
}
```



Given an array arr of size n and an integer X. Find if there's a triplet in the array which sums up to the given integer X.



**Example 1:**

**Input**:
n = 6, X = 13
arr[] = [1 4 45 6 10 8]
**Output**:
1
**Explanation**:
The triplet {1, 4, 8} in
the array sums up to 13.

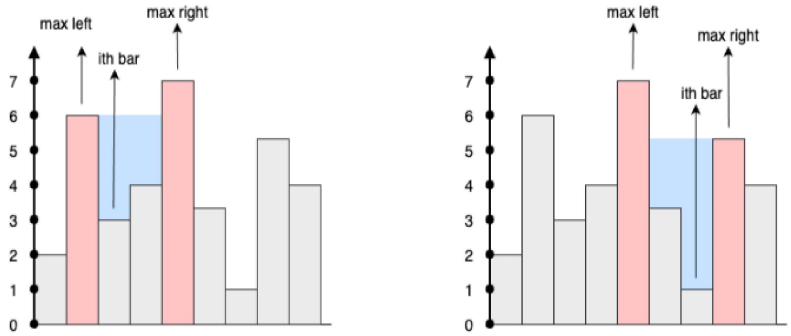
**Example 2:**

**Input**:
n = 5, X = 10
arr[] = [1 2 4 3 6]
**Output**:
1
**Explanation**:
The triplet {1, 3, 6} in
the array sums up to 10.

### Solution 1: Max Left, Max Right So Far!



- A `ith` bar can trap the water if and only if there exists a higher bar to the left and a higher bar to the right of `ith` bar.
- To calculate how much amount of water the `ith` bar can trap, we need to look at the maximum height of the left bar and the maximum height of the right bar, then
  - The water level can be formed at `ith` bar is: `waterLevel = min(maxLeft[i], maxRight[i])`
  - If `waterLevel >= height[i]` then `ith` bar can trap `waterLevel - height[i]` amount of water.
- To achieve in O(1) when looking at the maximum height of the bar on the left side and on the right side of `ith` bar, we pre-compute it:
  - Let `maxLeft[i]` is the maximum height of the bar on the left side of `ith` bar.
  - Let `maxRight[i]` is the maximum height of the bar on the right side of `ith` bar.









```
class Solution {
    trappingWater(arr, n) {
        let lbound = new Array(n);
        let rbound = new Array(n);
        lbound[0] = arr[0];
        rbound[n - 1] = arr[n - 1];

        for (let i = 1; i < n; i++) {
            lbound[i] = Math.max(lbound[i - 1], arr[i]);
        }
        for (let i = n - 2; i >= 0; i--) {
            rbound[i] = Math.max(rbound[i + 1], arr[i]);
        }

        // now calculating water
        let water = 0;
        for (let i = 0; i < n; i++) {
            water += Math.min(lbound[i], rbound[i]) - arr[i];
        }

        return water;
    }
}
```

## 26. Remove Duplicates from Sorted Array



Easy 11825 15734 Add to List

Given an integer array `nums` sorted in **non-decreasing order**, remove the duplicates **in-place** such that each unique element appears only **once**. The **relative order** of the elements should be kept the **same**. Then return *the number of unique elements in* `nums`.

Consider the number of unique elements of `nums` to be `k`, to get accepted, you need to do the following things:

- Change the array `nums` such that the first `k` elements of `nums` contain the unique elements in the order they were present in `nums` initially. The remaining elements of `nums` are not important as well as the size of `nums`.
- Return `k`.



**Custom Judge:**

The judge will test your solution with the following code:

```
int[] nums = [...]; // Input array
int[] expectedNums = [...]; // The
expected answer with correct length

int k = removeDuplicates(nums); //
Calls your implementation

assert k == expectedNums.length;
for (int i = 0; i < k; i++) {
    assert nums[i] == expectedNums[i];
}
```

sometimes copy works instead of swap and also never underestimate rotation

```cpp
class Solution {
public:
    // copy works sometimes not swap also rotation
    int removeDuplicates(vector<int>& nums) {
        int correct = 0;
        for(int j=1; j<nums.size(); j++){
            if(nums[j] != nums[correct]){
                correct++;
                nums[correct] = nums[j];
            }
        }
        return correct+1;
    }
};
```

Given an array **A[ ]** of positive integers of size **N**, where each value represents the number of chocolates in a packet. Each packet can have a variable number of chocolates. There are **M** students, the task is to distribute chocolate packets among **M** students such that :

1. Each student gets **exactly** one packet.
2. The difference between maximum number of chocolates given to a student and minimum number of chocolates given to a student is minimum.

**Example 1:**

**Input:**
N = 8, M = 5
A = {3, 4, 1, 9, 56, 7, 9, 12}
**Output:** 6
**Explanation:** The minimum difference between maximum chocolates and minimum chocolates is 9 - 3 = 6 by choosing following M packets :{3, 4, 9, 7, 9}.

**Example 2:**

**Input:**
N = 7, M = 3
A = {7, 3, 2, 4, 9, 12, 56}
**Output:** 2
**Explanation:** The minimum difference between maximum chocolates and minimum chocolates is 4 - 2 = 2 by choosing following M packets :{3, 2, 4}.

```cpp
class Solution {
    findMinDiff(arr, n, m) {
        arr.sort((a, b) => a - b);
        let minDiff = Infinity;
        let right = m - 1;
        for (let i = 0; right < n; i++) {
            let diff = arr[right] - arr[i];
            minDiff = Math.min(minDiff, diff);
            right++;
        }

        return minDiff;
    }
}
```

## Smallest subarray with sum greater than x

Easy Accuracy: 37.07% Submissions: 98K+ Points: 2



Given an array of integers (A[]) and a number x, find the smallest subarray with sum greater than the given value. If such a subarray do not exist return 0 in that case.

**Example 1:**

**Input:**
A[] = {1, 4, 45, 6, 0, 19}
x = 51
**Output:** 3
**Explanation:**
Minimum length subarray is
{4, 45, 6}

**Example 2:**

**Input:**
A[] = {1, 10, 5, 2, 7}
x = 9
**Output:** 1
**Explanation:**
Minimum length subarray is {10}



```cpp
// MANY EDGE CASES NOT EASY LEARN ALL EDGE CASES HERE
class Solution{
  public:
    int smallestSubWithSum(int arr[], int n, int x) {
        int minLen = n+1;
        int sum = 0;

        for(int i=0, j=0; i<n; i++) {
            if(arr[i] > x )
                return 1;

            sum += arr[i];
            while(sum > x) {
                minLen = min(minLen, i-j+1);
                sum -= arr[j++];
            }
        }
        // when x is greater than all (arr sum)
        if(sum <= x and minLen == n+1)
            return 0;

        return minLen;
    }
};
```

# 2090. K Radius Subarray Averages

Medium 1727 84 Add to List

You are given a **0-indexed** array `nums` of `n` integers, and an integer `k`.

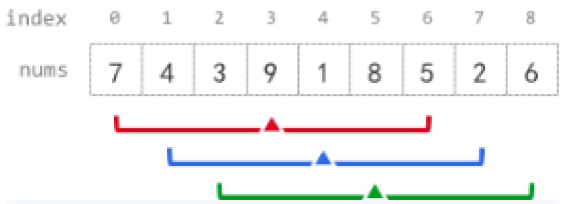
The **k-radius average** for a subarray of `nums` **centered** at some index `i` with the **radius** `k` is the average of **all** elements in `nums` between the indices `i - k` and `i + k` (**inclusive**). If there are less than `k` elements before **or** after the index `i`, then the **k-radius average** is `-1`.

Build and return *an array* `avgs` *of length* `n` *where* `avgs[i]` *is the* ***k-radius average*** *for the subarray centered at index* `i`.

The **average** of `x` elements is the sum of the `x` elements divided by `x`, using **integer division**. The integer division truncates toward zero, which means losing its fractional part.

- For example, the average of four elements `2`, `3`, `1`, and `5` is `(2 + 3 + 1 + 5) / 4 = 11 / 4 = 2.75`, which truncates to `2`.

**Example 1:**




```
Input: nums = [7,4,3,9,1,8,5,2,6], k = 3
Output: [-1,-1,-1,5,4,4,-1,-1,-1]
Explanation:
- avg[0], avg[1], and avg[2] are -1 because there are less than k elements before each index.
- The sum of the subarray centered at index 3 with radius 3 is: 7 + 4 + 3 + 9 + 1 + 8 + 5 = 37.
  Using integer division, avg[3] = 37 / 7 = 5.
- For the subarray centered at index 4, avg[4] = (4 + 3 + 9 + 1 + 8 + 5 + 2) / 7 = 4.
- For the subarray centered at index 5, avg[5] = (3 + 9 + 1 + 8 + 5 + 2 + 6) / 7 = 4.
- avg[6], avg[7], and avg[8] are -1 because there are less than k elements after each index.
```

**Example 2:**

```
Input: nums = [100000], k = 0
Output: [100000]
Explanation:
- The sum of the subarray centered at index 0 with radius 0 is: 100000.
  avg[0] = 100000 / 1 = 100000.
```

```javascript
/
var getAverages = function(arr, k) {
    let n = arr.length;
    let ans = new Array(n).fill(-1);
    let len = 2 * k + 1;
    let sum = 0, avg = 0;

    for (let right = 0, left = 0; right < n; right++) {
        sum += arr[right];
        if (right >= 2 * k) {
            avg = Math.floor(sum / len);
            ans[right - k] = avg;
            sum -= arr[left++];
        }
    }
    return ans;
};
```

# Three way partitioning

Easy Accuracy: 41.58% Submissions: 116K+ Points: 2

**Share your Interview, Campus or Work Experience to win GFG Swag Kits and much more!**

Given an array of size n and a range **[a, b]**. The task is to partition the array around the range such that array is divided into three parts.

1) All elements smaller than **a** come first.

2) All elements in range **a** to **b** come next.

3) All elements greater than **b** appear in the end.

The individual elements of three sets can appear in any order. You are required to return the modified array.

**Note**: The generated output is 1 if you modify the given array successfully.

**Example 1:**

**Input**:
n = 5
A[] = {1, 2, 3, 3, 4}
[a, b] = [1, 2]
**Output**: 1
**Explanation**: One possible arrangement is: {1, 2, 3, 3, 4}. If you return a valid arrangement, output will be 1.

```
class Solution{
public:
    void threeWayPartition(vector<int>& arr,int a, int b)
    {
        int low = 0, mid = 0, high = arr.size()-1;

        // code will give wrong o/p when used 3rd condition at 2nd position becz 3rd should
        // come after it 021 if greater than a but lesser than b than only in range(a, b)
        while(mid <= high) {
            if(arr[mid] < a){
                swap(arr[low++], arr[mid++]);
            }
            else if(arr[mid] > b) {
                swap(arr[mid], arr[high--]);
            }
            else {
                mid++;
            }
        }
    }
};
```

order 0 2 1 must else wrong

**Input**:
n = 3
A[] = {1, 2, 3}
[a, b] = [1, 3]
**Output**: 1
**Explanation:** One possible arrangement is: {1, 2, 3}. If you return a valid arrangement, output will be 1.

Given two integers n and r, find ${}^{n}C_{r}$. Since the answer may be very large, calculate the answer modulo $10^9+7$.

**Example 1:**

**Input:** n = 3, r = 2
**Output:** 3
**Explanation:** ${}^{3}C_{2} = 3$.

**Example 2:**

**Input:** n = 2, r = 4
**Output:** 0
**Explanation:** r is greater than n.

**Your Task:**
You do not need to take input or print anything. Your task is to complete the function **nCr()** which takes n and r as input parameters and returns ${}^{n}C_{r}$ modulo $10^9+7$..

**Expected Time Complexity:** O(n*r)
**Expected Auxiliary Space:** O(r)



## 3 Approaches || Pascal's triangle || nCr = n-1Cr + n-1Cr-1 with DP - Memoization/ Tabulation || Easy & Understandable C++ Solutions with explanation



---------- **PASCAL'S TRIANGLE APPROACH**----------

```
   1                          
  1 1                1C0  1C1
 1 2 1             2C0  2C1  2C2
1 3 3 1          3C0  3C1  3C2  3C3
1 4 6 4 1     4C0  4C1  4C2  4C3  4C4
```

- From the diagram itself, you might have understood that by creating Pascal's triangle up to the nth row, w e can easily answer nCr.
- No need to store the whole triangle; just use one current row and one previous row of the triangle.

**COMPLEXITY :** TC : O(n*r) || SC : O(n)

```
int mod=1e9+7;
    int nCr(int n, int r){
        // code here
        if(n<r)return 0;
        vector<int>pascal;
        for(int i=0;i<=n;i++){
            vector<int>v(i+1,1);
            for(int j=1;j<i;j++){
                v[j]=(pascal[j]+pascal[j-1])%mod;
            }
            pascal=v;
        }
        return pascal[r];
    }
```

You can reach out to me on my tele channel if you are still having doubts for the same https://t.me/leetcodegfg dailysolution

**COMPLEXITY (Memoization/Tabulation):**
TC : O(n*r) || SC : O(n*r)

**APPROACH :**

- **nCr = n-1Cr + n-1Cr-1**
- You must be aware of this combination formula. We will use this formula to calculate nCr.

**MEMOIZATION :**

- Create a 2D vector dp of size (n+1) by (r+1).
- Call the solve function
- Check the base cases, such that for

1. for r>n => nCr=0
2. for r=0 => nCr=1
3. for r=1 => nCr = n

- If the answer for n and r is already stored, retrieve the data from the dp table.
- Calculate the value of nCr using the combination formula and make the two required recursive calls.
- Return the answer while storing it in the dp table.

```
int mod=1e9+7;
    int solve(vector<vector<int>>&dp,int n,int r){
        if(r>n)return 0;
        if(r==0)return 1;
        if(r==1)return n;
        if(dp[n][r]!=-1)return dp[n][r];
        return dp[n][r]=(solve(dp,n-1,r-1)%mod + solve(dp,n-1,r)%mod)%mod;
    }
    int nCr(int n, int r){
        // code here
        vector<vector<int>>dp(n+1,vector<int>(r+1,-1));
        return solve(dp,n,r);
    }
```

**TABULATION :**

- Create a 2D vector dp of size (n+1) by (r+1).
- Initialize the base cases, such that for r=0, for all value of n, nCr=1
- Now simply iterate from 0 to n in a nested loop from 1 to min(i,r) to avoid cases where j > i, because in nCr, we know that n >= r.
- Calculate the value of nCr using the previously stored values in the dp table.
- Return the answer.

```
int mod=1e9+7;
    int nCr(int n, int r){
        // code here
        int i,j;
        vector<vector<int>>dp(n+1,vector<int>(r+1,0));
        for(i=0;i<=n;i++){
            dp[i][0]=1;
        }
        for(i=0;i<=n;i++){
            for(j=1;j<=min(i,r);j++){
                dp[i][j]=(dp[i-1][j]%mod + dp[i-1][j-1]%mod)%mod;
            }
        }
        return dp[n][r];
    }
```

```
class Solution{
    int mod = 1e9+7;
    public:
    int solve(vector<vector<int>>&dp,int n,int r){
        if(r > n)
            return 0;
        if(r == 0)
            return 1;
        if(r == 1)
            return n;
        if(dp[n][r] != -1)
            return dp[n][r];

        return dp[n][r] = (solve(dp, n-1, r-1)%mod + solve(dp, n-1, r)%mod)%mod;
    }

    int nCr(int n, int r){
        vector<vector<int>> dp(n+1, vector<int>(r+1, -1));
        return solve(dp, n, r);
    }
};
```

# Minimum pair merge operations required to make Array non-increasing



rohitpal210



Given an array **A[]**, the task is to find the minimum number of operations required in which two adjacent elements are removed from the array and replaced by their sum, such that the array is converted to a non-increasing array.

**Note:** An array with a single element is considered non-increasing.

**Examples:**

***Input:*** *A[] = {1, 5, 3, 9, 1}*
***Output:*** *2*
***Explanation:***
*Replacing {1, 5} by {6} modifies the array to {6, 3, 9, 1}*
*Replacing {6, 3} by {9} modifies the array to {9, 9, 1}*
***Input:*** *A[] = {0, 1, 2}*
***Output:*** *2*

Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.

**Approach:** The idea is to use Dynamic Programming. A memoization table is used to store the minimum count of operations required to make subarrays non-increasing from right to left of the given array. Follow the steps below to solve the problem:

- Initialize an array **dp[]** where **dp[i]** stores the minimum number of operations required to make the subarray {A[i], ..., A[N]} non-increasing. Therefore, the target is to compute dp[0].
- Find a minimal subarray **{A[i] .. A[j]}** such that **sum({A[i] ... A[j]}) > val[j+1]**, where, val[j + 1] is the merged sum obtained for the subarray **{A[j + 1], ... A[N]}**.
- Update **dp[i]** to **j – i + dp[j+1]** and **vals[i]** to **sum({A[i] ... A[j]})**.



Remove Watermark Now

```
function solve(a) {
    let n = a.length;
    let dp = new Array(n+1).fill(0);
    let val = new Array(n+1).fill(0);
    // dp[i]: Stores minimum number of operations required to make
    // subarray {A[i], ..., A[N]} non-increasing
    for (let i = n - 1; i >= 0; i--) {
        let sum = a[i];
        let j = i;
        while (j + 1 < n && sum < val[j + 1]) {
            j++;
            sum += a[j];
        }
        dp[i] = (j - i) + dp[j + 1];
        val[i] = sum;
    }
    return dp[0];
}
```

# 2735. Collecting Chocolates

Medium 245 475 Add to List

You are given a **0-indexed** integer array `nums` of size `n` representing the cost of collecting different chocolates. The cost of collecting the chocolate at the index `i` is `nums[i]`. Each chocolate is of a different type, and initially, the chocolate at the index `i` is of $i^{th}$ type.

In one operation, you can do the following with an incurred **cost** of `x`:

- Simultaneously change the chocolate of $i^{th}$ type to $((i + 1) \bmod n)^{th}$ type for all chocolates.

Return *the minimum cost to collect chocolates of all types, given that you can perform as many operations as you would like.*

**Example 1:**

```
Input: nums = [20,1,15], x = 5
Output: 13
Explanation: Initially, the chocolate types are [0,1,2]. We will buy the 1st type of chocolate at a cost of 1.
Now, we will perform the operation at a cost of 5, and the types of chocolates will become [1,2,0]. We will buy the 2nd type of chocolate at a cost of 1.
Now, we will again perform the operation at a cost of 5, and the chocolate types will become [2,0,1]. We will buy the 0th type of chocolate at a cost of 1.
Thus, the total cost will become (1 + 5 + 1 + 5 + 1) = 13. We can prove that this is optimal.
```

**Example 2:**

```
Input: nums = [1,2,3], x = 4
Output: 6
Explanation: We will collect all three types of chocolates at their own price without performing any operations. Therefore, the total cost is 1 + 2 + 3 = 6.
```

```cpp
class Solution {
public:
    void rotate(vector<int>& nums, int n){
        int first = nums[0];
        for(int i=0; i<n-1; i++)
            nums[i] = nums[i+1];
        nums[n-1] = first;
    }
    // at max n-1 rotation becz after n array will be same
    // first get normal cost without rotation then leftrotate and calc + x and get min of all
    long long minCost(vector<int>& nums, int x) {
        int n = nums.size();
        vector<int> rMin(n, INT_MAX);
        long long minCost = LONG_MAX;

        for(int i=0; i<n; i++) {
            // first time bina rotation wala so extras (0*x)
            if(i != 0)
                rotate(nums, n);

            long long cost = 0;
            for(int i=0; i<n; i++) {
                rMin[i] = min(nums[i], rMin[i]);
                cost += rMin[i];
            }
            // jitni baar rotate hoga utni rotation cost
            cost += ((long long)i*(long long)x);
            minCost = min(minCost, cost);

        }
        return minCost;
    }
};
```

# 1578. Minimum Time to Make Rope Colorful

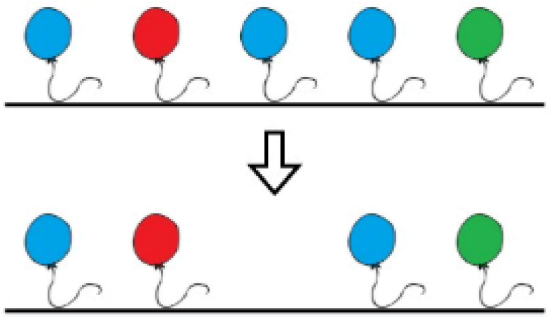
Medium 2845 82 Add to List

Alice has `n` balloons arranged on a rope. You are given a **0-indexed** string `colors` where `colors[i]` is the color of the $i^{th}$ balloon.

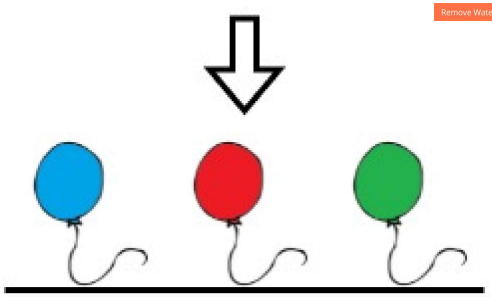
Alice wants the rope to be **colorful**. She does not want **two consecutive balloons** to be of the same color, so she asks Bob for help. Bob can remove some balloons from the rope to make it **colorful**. You are given a **0-indexed** integer array `neededTime` where `neededTime[i]` is the time (in seconds) that Bob needs to remove the $i^{th}$ balloon from the rope.

Return *the **minimum time** Bob needs to make the rope **colorful**.*

**Example 1:**

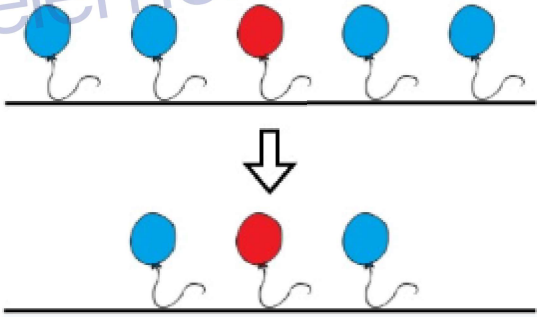


```
Input: colors = "abaac", neededTime = [1,2,3,4,5]
Output: 3
```



```
Input: colors = "abc", neededTime = [1,2,3]
Output: 0
Explanation: The rope is already colorful. Bob does not need to remove any balloons from the rope.
```

**Example 3:**



```
Input: colors = "aabaa", neededTime = [1,2,3,4,1]
Output: 2
Explanation: Bob will remove the ballons at indices 0 and 4. Each ballon takes 1 second to remove. There are no longer two consecutive balloons of the same color. Total time = 1 + 1 = 2.
```

```cpp
class Solution {
public:
    int minCost(string colors, vector<int>& neededTime) {
        int cost = 0;
        int n = colors.size(), i=0, j=0;
        while(j<n){
            int curCost = 0, maxTime = 0;
            while(j<n && colors[i]==colors[j]){
                curCost += neededTime[j];
                maxTime = max(maxTime, neededTime[j]);
                ++j;
            }
            cost += curCost - maxTime;    except max leave all cost t minimize
            i=j;
        }
        return cost;
    }
};
```

```javascript
var minCost = function(colors, neededTime) {
    let cost = 0;
    const n = neededTime.length;
    const costTrack = neededTime.slice(); // Create a copy of neededTime array

    for (let i = 1; i < n; i++) {
        if (colors[i - 1] === colors[i]) {
            cost += Math.min(costTrack[i - 1], costTrack[i]);
            // We took mini and left max for further in case of
// 1 2 1 3 => in 1 2 took 1 left 2 then in 2 1 took 1 left 2 and in 2 3 took 3 end
            costTrack[i] = Math.max(costTrack[i - 1], costTrack[i]);
        }
    }

    return cost;
};
```

```cpp
class Solution {
public:
    struct node{
        char c;
        int val;
    };

    int minCost(string s, vector<int>& cost) {
        stack<node> stack;
        int n=s.size(), result=0;

        for(int i=0; i<n; i++){
            if(!stack.empty() && stack.top().c==s[i]){
                if(stack.top().val<cost[i]){
                    result+=stack.top().val;
                    stack.pop();
                    stack.push({s[i], cost[i]});
                }
                else{
                    result+=cost[i];
                }
            }
            else{
                stack.push({s[i], cost[i]});
            }
        }

        return result;
    }
};
```



```cpp
class Solution {
public:
    int minCost(string colors, vector<int>& time) {
        int n = colors.length();
        int t = time.size();
        int ans = 0;
        priority_queue<int,vector<int>,greater<int>> pq;
        if(n <= 1)return 0;
        for(int i=0;i<n;i++){
            if(i+1 < n && colors[i] == colors[i+1]){
                pq.push(time[i]);
            }else{
                pq.push(time[i]);
                if(pq.size() == 1){
                    pq.pop();
                }
                else{
                    while(pq.size() > 1){
                        ans += pq.top();
                        pq.pop();
                    }
                    // to pop last element left to empty queue
                    pq.pop();
                }
            }
        }
        return ans;
    }
};
```

# Sliding Window Leetcode article



### Template 1: Sliding Window (Shrinkable)

The best template I've found so far:

```
int i = 0, j = 0, ans = 0;
for (; j < N; ++j) {
    // CODE: use A[j] to update state which might make the window invalid
    for (; invalid(); ++i) { // when invalid, keep shrinking the left edge until it's valid again
        // CODE: update state using A[i]
    }
    ans = max(ans, j - i + 1); // the window [i, j] is the maximum window we've found thus far
}
return ans;
```

Essentially, we want to **keep the window valid** at the end of each outer `for` loop.

**Solution for this question:**

1. What should we use as the `state`? It should be the sum of numbers in the window
2. How to determine `invalid`? The window is invalid if `(j - i + 1) * A[j] - sum > k`.

```
// OJ: https://leetcode.com/problems/frequency-of-the-most-frequent-element/
// Author: github.com/lzl124631x
// Time: O(NlogN)
// Space: O(1)
class Solution {
public:
    int maxFrequency(vector<int>& A, int k) {
        sort(begin(A), end(A));
        long i = 0, N = A.size(), ans = 1, sum = 0;
        for (int j = 0; j < N; ++j) {
            sum += A[j];
            while ((j - i + 1) * A[j] - sum > k) sum -= A[i++];
            ans = max(ans, j - i + 1);
        }
        return ans;
    }
};
```





**Solution for this question:**

```
// OJ: https://leetcode.com/problems/frequency-of-the-most-frequent-element/
// Author: github.com/lzl124631x
// Time: O(NlogN)
// Space: O(1)
class Solution {
public:
    int maxFrequency(vector<int>& A, int k) {
        sort(begin(A), end(A));
        long i = 0, j = 0, N = A.size(), sum = 0;
        for (; j < N; ++j) {
            sum += A[j];
            if ((j - i + 1) * A[j] - sum > k) sum -= A[i++];
        }
        return j - i;
    }
};
```

Apply these templates to other problems

1493. Longest Subarray of 1's After Deleting One Element (Medium)

Sliding Window (Shrinkable)

1. What's `state`? `cnt` as the number of `0`s in the window.
2. What's `invalid`? `cnt > 1` is invalid.

```
// OJ: https://leetcode.com/problems/longest-subarray-of-1s-after-deleting-one-element/
// Author: github.com/lzl124631x
// Time: O(N)
// Space: O(1)
class Solution {
public:
    int longestSubarray(vector<int>& A) {
        int i = 0, j = 0, N = A.size(), cnt = 0, ans = 0;
        for (; j < N; ++j) {
            cnt += A[j] == 0;
            while (cnt > 1) cnt -= A[i++] == 0;
            ans = max(ans, j - i); // note that the window is of size `j - i + 1`. We use `j - i`
        }                                                                  here becz we need to delete a char
        return ans;
    }
};
```

## Sliding Window (Non-shrinkable)

```
// OJ: https://leetcode.com/problems/longest-subarray-of-1s-after-deleting-one-element/
// Author: github.com/lzl124631x
// Time: O(N)
// Space: O(1)
class Solution {
public:
    int longestSubarray(vector<int>& A) {
        int i = 0, j = 0, N = A.size(), cnt = 0;
        for (; j < N; ++j) {
            cnt += A[j] == 0;
            if (cnt > 1) cnt -= A[i++] == 0;
        }
        return j - i - 1;
    }
};
```

## 3. Longest Substring Without Repeating Characters (Medium)

### Sliding Window (Shrinkable)

1. `state`: `cnt[ch]` is the number of occurrence of character `ch` in window.
2. `invalid`: `cnt[s[j]] > 1` is invalid.

```
// OJ: https://leetcode.com/problems/longest-substring-without-repeating-characters/
// Author: github.com/lzl124631x
// Time: O(N)
// Space: O(1)
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int i = 0, j = 0, N = s.size(), ans = 0, cnt[128] = {};
        for (; j < N; ++j) {
            cnt[s[j]]++;
            while (cnt[s[j]] > 1) cnt[s[i++]]--;
            ans = max(ans, j - i + 1);
        }
        return ans;
    }
};
```

### Sliding Window (Non-shrinkable)

Note that since the non-shrinkable window might include multiple duplicates, we need to add a variable to our state.

1. `state`: `dup` is the number of different kinds of characters that has duplicate in the window. For example, if window contains `aabbc`, then `dup = 2` because `a` and `b` has duplicates.
2. `invalid`: `dup > 0` is invalid

```
// OJ: https://leetcode.com/problems/longest-substring-without-repeating-characters/
// Author: github.com/lzl124631x
// Time: O(N)
// Space: O(1)
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int i = 0, j = 0, N = s.size(), cnt[128] = {}, dup = 0;
        for (; j < N; ++j) {
            dup += ++cnt[s[j]] == 2;
            if (dup) dup -= --cnt[s[i++]] == 1;
        }
        return j - i;
    }
};
```

## 713. Subarray Product Less Than K (Medium)

### Sliding Window (Shrinkable)

- `state`: `prod` is the product of the numbers in window
- `invalid`: `prod >= k` is invalid.

Note that since we want to make sure the window `[i, j]` is valid at the end of the `for` loop, we need `i <= j` check for the inner `for` loop. `i == j + 1` means this window is empty.

Each maximum window `[i, j]` can generate `j - i + 1` valid subarrays, so we need to add `j - i + 1` to the answer.


## 713. Subarray Product Less Than K (Medium)

### Sliding Window (Shrinkable)

- `state`: `prod` is the product of the numbers in window
- `invalid`: `prod >= k` is invalid.

Note that since we want to make sure the window `[i, j]` is valid at the end of the `for` loop, we need `i <= j` check for the inner `for` loop. `i == j + 1` means this window is empty.

Each maximum window `[i, j]` can generate `j - i + 1` valid subarrays, so we need to add `j - i + 1` to the answer.


```
// OJ: https://leetcode.com/problems/subarray-product-less-than-k/
// Author: github.com/lzl124631x
// Time: O(N)
// Space: O(1)
class Solution {
public:
    int numSubarrayProductLessThanK(vector<int>& A, int k) {
        if (k == 0) return 0;
        long i = 0, j = 0, N = A.size(), prod = 1, ans = 0;
        for (; j < N; ++j) {
            prod *= A[j];
            while (i <= j && prod >= k) prod /= A[i++];
            ans += j - i + 1;
        }
        return ans;
    }
};
```


Remove Watermark Now

**The non-shrinkable template is not applicable here since we need to the length of each maximum window ending at each position**

Below is my original answer during contest. As you can see, if I don't use this template, the solution could be a bit complex.



Solution 1. Sliding Window

Let two pointers `i, j` form a window `[i, j]`. The window is valid if `(j - i + 1) * A[j] - sum <= k`.

We keep increasing `j` to expand the window as much as possible. When the window becomes invalid, we increment `i`.

```
// OJ: https://leetcode.com/problems/frequency-of-the-most-frequent-element/
// Author: github.com/lzl124631x
// Time: O(NlogN)
// Space: O(1)
class Solution {
public:
    int maxFrequency(vector<int>& A, int k) {
        sort(begin(A), end(A));
        long i = 0, j = 0, N = A.size(), ans = 1, sum = A[0];
        for (; i < N; ++i) {
            while (j < N && (j - i + 1) * A[j] - sum <= k) {
                ans = max(ans, j - i + 1);
                ++j;
                if (j < N) sum += A[j];
            }
            sum -= A[i];
        }
        return ans;
    }
};
```



## Problems Solvable using this template

- 3. Longest Substring Without Repeating Characters
- 159. Longest Substring with At Most Two Distinct Characters (Medium)
- 340. Longest Substring with At Most K Distinct Characters
- 424. Longest Repeating Character Replacement
- 487. Max Consecutive Ones II
- 713. Subarray Product Less Than K
- 1004. Max Consecutive Ones III
- 1208. Get Equal Substrings Within Budget (Medium)
- 1493. Longest Subarray of 1's After Deleting One Element
- 1695. Maximum Erasure Value
- 1838. Frequency of the Most Frequent Element
- 2009. Minimum Number of Operations to Make Array Continuous
- 2024. Maximize the Confusion of an Exam

## The following problems are also solvable using the shrinkable template with the "At Most to Equal" trick

- 930. Binary Subarrays With Sum (Medium)
- 992. Subarrays with K Different Integers
- 1248. Count Number of Nice Subarrays (Medium)
- 2062. Count Vowel Substrings of a String (Easy)

Hi everyone!

I'm a university student who recently studied a lot of sliding window for summer intern interviews. I really appreciate people who write posts like this - they helped me so much. So I would also like to share some ideas, about sliding window, and what is not sliding window.



Sliding Window is very common in interviews and many questions that look like "(*Longest/Shortest/Number of*) (*Substrings/Subarrays*) with (*At most/Exactly*) K elements that fit (*some condition*)" have a common pattern. They are usually O(n).

**Sliding Window Questions**

- 3. Longest Substring Without Repeating Characters
- 340. Longest Substring with At Most K Distinct Characters
- 76. Minimum Window Substring
- 1004. Max Consecutive Ones III
- 904. Fruit Into Baskets
- 424. Longest Repeating Character Replacement
- 930. Binary Subarrays with Sum
- 992. Subarrays with K Different Integers
- 1248. Count Number of Nice Subarrays
- 1358. Number of Substrings Containing All Three Characters

**Not sliding window/not this pattern**

- Subarray Sum Equals K
- Longest Subarray with Absolute Difference <= Limit
- Longest Palindromic Substring

**Idea/Intuition**

We have a "window" of 2 pointers, `left` and `right`, and we keep increasing the right pointer.

- If the element at the right pointer makes the window not valid, we keep moving the left pointer to shrink the window until it becomes valid again.
- Then, we update the global min/max with the result from the valid window.
- To check if it is valid, we need to store the "state" of the window (ex. frequency of letters, number of distinct integers).



```
for(right = 0; right < n; right++):
    update window with element at right pointer
    while (condition not valid):
        remove element at left pointer from window, move left pointer to the right
    update global max
```

**Length of Substring/Subarray questions:**

The first type of common question we will look at is, Max/Min *length* of subarray that fits some condition.

3. Longest substring without repeating characters

```
        unordered_map<char, int> counts; // Frequencies of chars in the window
        int res = 0;
        int i = 0; // Left pointer
        for(int j = 0; j < s.length(); j++){
            counts[s[j]] ++; // Add right pointer to window
            while(counts[s[j]] > 1){ // While the element at right pointer created a repeat
                counts[s[i++]] --; // While condition not valid, remove element at left pointer from window by decreasing its count, and then increment le
ft pointer. In this case, it is while s[j] is a duplicate (we will stop after removing the duplicate copy of s[j]).
            } // Now the condition is valid
            res = max(res, j-i+1); // Update global max with the length of current valid substring
        }
        return res;
```

340. Longest Substring with At Most K Distinct Characters

This is a more generalized version of At Most 2 or 1 distinct characters. We keep a hashmap of the counts of characters in the current window, and also a numDistinct variable for how many distinct characters are in the current window.

```
int lengthOfLongestSubstringKDistinct(string &s, int k) {
    unordered_map<char, int> counts; // Frequencies of chars in the window
    int res = 0;
    int numDistinct = 0;
    int i = 0; // Left pointer
    for(int j = 0; j < s.length(); j++){
        if(counts[s[j]] == 0) numDistinct++;
        counts[s[j]] ++; // Add right pointer to window
        while(numDistinct > k){
            counts[s[i]] --;
            if(counts[s[i]] == 0) numDistinct --;
            i++;
        } // Now the condition is valid
        res = max(res, j-i+1);
    }
    return res;
}
```

76. Minimum Window Substring

Common in interviews, and although it says Hard, it is very similar to Medium sliding window questions.

Instead of counting `numDistinct` for number of distinct characters in the window, we count how many distinct characters of t are fully contained in the current window (how many are `remain`ing) so that if `remain = 0`, then all the characters of t are contained in the current window of s.

```
string minWindow(string s, string t) {
    unordered_map<char, int> count;
    int min_j = INT_MAX, min_i = 1, i = 0;
    // Initialize with count of chars of t
    for(int i = 0; i < t.length(); i++){
        count[t[i]] ++;
    }
    int remain = count.size(); // Number of distinct characters in t

    for(int j = 0; j < s.length(); j++){
        count[s[j]] --;
        if(count[s[j]] == 0) remain--; // The count of s[j] in the current window of s has equalled the count of s[j] in all of t, so we are done with thi
s character
        while(remain == 0){
            if(j-i < min_j - min_i){ // Update global variables since we need to return the string
                min_j = j;
                min_i = i;
            }
            // Remove i from the window and increment i
            count[s[i]] ++;
            if(count[s[i]] > 0) remain++;
            i++;
        }
    }
    return min_j == INT_MAX ? "" : s.substr(min_i, min_j - min_i+1);
}
```



## Differently Worded Max/Min Length Sliding Window Questions

These questions may look very different from the previous ones, but are just worded differently; the main idea is the same.

1004. Max Consecutive Ones III

This means, longest subarray with at most K `0`s.

```
int longestOnes(vector<int>& A, int K) {
    vector<int> freq(2);
    int i = 0, r=0;
    for(int j = 0; j < A.size(); j++){
        freq[A[j]] ++;
        while(freq[0] > K && i <= j){
            freq[A[i]] --;
            i++;
        }
        r = max(r, j-i+1);
    }
    return r;
}
```

904. Fruit Into Baskets

This is essentially the same as Longest substring with at most K distinct characters, but it is subarrays with at most 2 distinct numbers.

```
int totalFruit(vector<int>& a) {
    vector<int> freq(a.size(), 0);
    int res = 0, i=0, distinct=0;
    for(int j = 0; j < a.size(); j++){
        freq[a[j]] ++;
        if(freq[a[j]] == 1) distinct++;

        if(distinct <= 2)
            res= max(res,  j-i+1);

        while(distinct > 2 && i < j){
            freq[a[i]] --;
            if(freq[a[i]] == 0) distinct--;
            i++;
        }
    }
    return res;
}
```

424. Longest Repeating Character Replacement

This can be thought of as, Longest substring where (count of most frequent character) + k < length.



- On top of the hashmap for character counts, we also maintain `maxCount` the count of the most frequent character so far.
- When moving the left pointer we don't need to decrease maxCount, because if maxCount is too big, the "right - left + 1 - k > maxCount" is less likely to be true and we don't move the left pointer (as expected).
- But we do need to maintain the frequencies map so we can increase maxCount if needed.

```
int characterReplacement(string s, int k) {
    if(s.length() == 0) return 0;
    if(s.length() <= k) return s.length();
    unordered_map<char, int> m;
    int res = 0; int l = 0;
    int maxCount = 0;

    for(int i = 0; i < s.length(); i++){
        m[s[i]] ++;
        maxCount = max(maxCount, m[s[i]]);
        while(i - l + 1 - k > maxCount){
            m[s[l]] --;
            l++;
        }
        res = max(res, i-l+1);

    }
    return res;
}
```



**"Number of Subarrays" Questions**



Before, we were finding the **min/max length** of subarray. Now we want the **number of** subarrays.

930. Binary Subarrays with Sum

Another difference this question has to the previous ones, is that it asks for subarrays with **exactly** `K` 1s. Previous ones were **at most** `K` distinct characters. So, one way is we can do the `at most`, and then `atMost(k) - atMost(k-1)`.

```
int numSubarraysWithSum(vector<int>& A, int S) {
    return atMost(A, S) - atMost(A, S - 1);
}

int atMost(vector<int>& A, int S) {
    if (S < 0) return 0;
    int res = 0, i = 0;
    for (int j = 0; j <A.size(); j++) {
        S -= A[j];
        while (S < 0){
            S += A[i]; i++;}

        res += j - i + 1;
    }
    return res;
}
```

You might have noticed, why are we doing `res += j-i+1` instead of `res = max/min (res, j-i+1)`.

If [i ... j] is a valid subarray, ex. [1,2,3,4]

- There are 4 subarrays with length 1, [1], [2], [3], [4]
- 3 with length 2: [1,2], [2,3], [3,4]
- 2 with length 3
- 1 with length 4



So if at every step we added the length:

- [1] -> add 1
- [1,2] -> add 2
- [1,2,3] -> add 3
- [1,2,3,4] -> add 4

the sum 1+2+3+4 would be the same as the number of subarrays. So we can add the lengths of the valid subarrays.

**"Prefixed" Sliding Window**

If the subarray `[i,j]` contains `K` 1s, and the first `p` numbers of the subarray are 0, then that means all the subarrays from `[i,j]` to `[i+p, j]` inclusive are valid, which is `p+1` subarrays.

- In the first loop, we move the left pointer to the right while the sum is too high, just like in a standard sliding window question.
  - We set p to 0 because i is changing.
- In the 2nd loop, after finding some [i,j], we find how many leading 0s are in this subarray.
- Finally, we need to check if the current sum equals the target again - because the first loop didn't check if the sum is too low.

```
int numSubarraysWithSum(vector<int>& A, int S) {
    int p=0, i=0, r=0, c=0;
    for(int j = 0; j < A.size(); j++){
        c+= A[j];
        while(c > S && i < j){
            p=0;
            c -= A[i];
            i++;
        }
        while(A[i] == 0 && i < j){
            p++;
            c -= A[i];
            i++;
        }
        if(c == S) r += p+1;
    }
    return r;
```

We can also do this question with the same method as "Subarray Sum Equals K" (mentioned at the end) and count prefix sums.

992. Subarrays with K different integers

**Prefixed Sliding Window**

If the subarray `[i,j]` contains X distinct numbers, and the first `p` numbers are duplicates (are also in `[i + p, j]`, that means all the subarrays from `[i,j]`, `[i+1, j]`, … `[i+p, j]` have X distinct numbers as well. Then if `[i,j]` is valid, there will be `p+1` valid subarrays.

992. Subarrays with K different integers

**Prefixed Sliding Window**

If the subarray `[i,j]` contains X distinct numbers, and the first `p` numbers are duplicates (are also in `[i + p, j]`, that means all the subarrays from `[i,j]`, `[i+1, j]`, … `[i+p, j]` have X distinct numbers as well. Then if `[i,j]` is valid, there will be `p+1` valid subarrays.

```
int subarraysWithKDistinct(vector<int>& A, int K) {
    unordered_map<int, int> m;
    int i = 0, p=0, d=0, r = 0;

    for(int j = 0; j < A.size(); j++){
        m[A[j]] ++;

        if(m[A[j]] == 1) d++; // num distinct

        if(d > K) {
            m[A[i]] --;
            d--;
            i++;
            p = 0;
        }

        // while A[i] has a duplicate in the current [i..j] subarray
        while(m[A[i]] > 1){
            m[A[i]] --;
            i++; p++;
        }

        if(d == K) r += p+1;

    }
    return r;

}
```

We could also do the `atMost(k) - atMost(k-1)` strategy:

```
int subarraysWithKDistinct(vector<int>& A, int K) {
    return atMost(A, K) - atMost(A, K - 1);
}
int atMost(vector<int>& A, int K) {
    int i = 0, res = 0;
    unordered_map<int, int> count;

    for (int j = 0; j < A.size(); j++) {
        count[A[j]]++;
        if (count[A[j]] == 1) K--;
        while (K < 0) {
            count[A[i]]--;
            if (count[A[i]] == 0) K++;
            i++;
        }
        res += j - i + 1;
    }
    return res;
}
```

Count Number of Nice Subarrays

Number of subarrays with K odd numbers; this is very similar to the previous 2 questions, subarrays with K distinct integers, or with K 1s.

**Prefixed**:

If `[i...j]` has X odd numbers, and the first `p` numbers in this subarray ( `nums[i], nums[i+1], …, nums[i+p-1]` ) are even, then `[i..j]` contains `p+1` subarrays with X odd numbers. So, if `[i...j]` has K odd numbers, then it will contain `p+1` valid subarrays.

```
int numberOfSubarrays(vector<int>& nums, int k) {
    int r = 0, i=0, p=0;
    vector<int> freq(2, 0);
    for(int j = 0; j < nums.size(); j++){
        freq[nums[j] % 2] ++;
        while(freq[1] > k && i < j){
            p = 0; freq[nums[i] % 2] --; i++;
        }
        while(nums[i]% 2 == 0 && i < j){
            p++;
            freq[nums[i]%2] --; i++;
        }
        if(freq[1] == k) r += p+1;

    }
    return r;
}
```

**atMost:**

```
int numberOfSubarrays(vector<int>& nums, int k) {
    return atMost(nums, k) - atMost(nums, k - 1);
}

int atMost(vector<int>& A, int k) {
    int res = 0, i = 0, n = A.size();
     vector<int> freq(2, 0);
    for (int j = 0; j < n; j++) {
        freq[A[j] % 2] ++;
        while(freq[1] > k){
            freq[A[i] % 2] --;
            i++;
        }
        res += j - i + 1;
    }
    return res;
}
```

Subarrays containing all three characters

Similar to subarrays with K distinct integers, this one is Subarrays with 3 distinct characters.

```
int numberOfSubstrings(string s) {
    int  j=0, distinct = 0, res=0, prefix=0;
    vector<int> freq(3);

    for(int i = 0; i < s.length(); i++){
        if(freq[s[i] - 'a']== 0) distinct++;
        freq[s[i] - 'a'] ++;

        while(j < i && freq[s[j]-'a'] > 1){
            freq[s[j]-'a']--;
            j++;
            prefix++;
        }
        if(distinct == 3){
            res += prefix+1;
        }
    }
    return res;
}
```

**When does this template not work?**

**A common problem I faced when studying Leetcode is not knowing when to apply a pattern or not.**

Here are some questions that look like this pattern, but are not:

**1:** Sliding window doesn't work if knowing one element at the edges of the window, does not tell you how to update the state of the window, or whether it becomes valid.

For example, here is a very interesting Sliding Window I got asked on an interview:
Longest Subarray with Absolute Diff <= Limit

The previous types of sliding window don't work because we need the **max and min of the window**. How would we do that while loop, `while (condition is not valid) left++`, if we don't know whether `left` is the max or min of the window?

- We need to keep **monotonic queues** of the index of max and min elements in the current window. Store indices not values, so we can know the length of sliding window.
- The front of q1 is the max, and the front of q2 is the min.
- Elements in q1 are decreasing and elements in q2 are increasing. Every time we add new right pointer element, we have to pop old elements from the deques to maintain the order.
- Finally, if the limit is exceeded, we pop the old max/min from the left of the queues, and use the index of the popped element to check whether we need to pop the other queue or update res.

```
int longestSubarray(vector<int>& nums, int limit) {
    if(nums.size() == 0) return 0;
    int i = 0, res = 0;
    deque<int> q1, q2; // for max and min
    for(int j = 0; j < nums.size(); j++){
        while(!q1.empty() && nums[q1.back()] < nums[j] ) q1.pop_back();
        while(!q2.empty() && nums[q2.back()] > nums[j] ) q2.pop_back();
        q1.push_back(j);
        q2.push_back(j);
        while(!q1.empty() && !q2.empty() && (nums[q1.front()] - nums[j] > limit || nums[j] - nums[q2.front()] > limit )){
            if(nums[q1.front()] - nums[j] > limit) {  i = max(i, q1.front() +1); q1.pop_front();}
            if(nums[j] - nums[q2.front()] > limit) {i = max(i, q2.front() +1); q2.pop_front();}
        }
        res = max(res, j-i+1);

    }
    return res;
}
```

I recommend looking at **Monotonic Stack/Queue** problems as it is a very interesting category!



**2:** Another type of question that doesn't work, is if adding one new element could either increase or decrease the window's state.

- For `Longest substring with at most K distinct characters`, we know that adding a new character to the window can only increase the numDistinct. Removing from the window can only decrease it.
- But for `Subarray Sums Equals K`, with *negative numbers*, this is not the case.

So the usual sliding window template does not work, and we need to keep a `HashMap` of prefix sums, and at each new element check how many previous places had a prefix sum of [current prefix sum - k].
This is out of the scope of this post, but their solution is very detailed!

```
int subarraySum(vector<int>& nums, int k) {
    unordered_map<int, int> m;
    m[0] = 1;
    int total = 0;
    int count = 0;
    for(int i = 0; i < size(nums); i++){
        count += nums[i];

        if(m.find(count - k) != m.end()) total += m[count-k];
        m[count] += 1;

    }
    return total;

}
```

Subarray Sum/Product is also a whole other category of problem! Very interesting. For example:
https://leetcode.com/problems/subarray-product-less-than-k/
https://leetcode.com/problems/subarray-sums-divisible-by-k/

**3:** Similar to the 1st reason: If given an invalid subarray it is hard to check whether adding or removing from only one end at a time would ever make it valid.

5. Longest Palindromic Substring

Most "palindrome" questions are not sliding window, because the nature of palindrome means you either expand the 2 pointers from the center or move the pointers inwards, one starts at 0 and one starts at n-1.

If we tried to apply the pattern on longest palindrome question,

- it is hard to check if adding the element at right pointer makes the window valid
- and if the new element makes it invalid, hard to check how many times we need to move the left pointer to make it valid

The solution is to consider every possible "center" of the palindrome and expand the 2 pointers outward.

**More Useful Links**

- (Prefixed sliding window diagram) https://leetcode.com/problems/subarrays-with-k-different-integers/discuss/235235/C%2B%2BJava-with-picture-prefixed-sliding-window
- (List of similar questions) https://leetcode.com/problems/number-of-substrings-containing-all-three-characters/discuss/516977/JavaC++Python-Easy-and-Concise
- (Minimum Window Substring template) https://leetcode.com/problems/minimum-window-substring/discuss/26808/here-is-a-10-line-template-that-can-solve-most-substring-problems
- (What can or cannot be solved with Two Pointers) https://leetcode.com/problems/subarray-sum-equals-k/discuss/301242/General-summary-of-what-kind-of-problem-can-cannot-solved-by-Two-Pointers

Thanks for reading, and happy coding! Please let me know if I missed something or if you know of any interesting questions to learn!



# 1493. Longest Subarray of 1's After Deleting One Element



Medium 3198 54 Add to List Share

Given a binary array `nums`, you should delete one element from it.

Return *the size of the longest non-empty subarray containing only* `1` *'s in the resulting array*. Return `0` if there is no such subarray.

**Example 1:**

```
Input: nums = [1,1,0,1]
Output: 3
Explanation: After deleting the number in position 2,
[1,1,1] contains 3 numbers with value of 1's.
```

**Example 2:**

```
Input: nums = [0,1,1,1,0,1,1,0,1]
Output: 5
Explanation: After deleting the number in position 4,
[0,1,1,1,1,1,0,1] longest subarray with value of 1's is
[1,1,1,1,1].
```

**Example 3:**

```
Input: nums = [1,1,1]
Output: 2
Explanation: You must delete one element.
```

```cpp
class Solution {
public:
    int longestSubarray(vector<int>& arr) {
        int n = arr.size();
        int left = 0, right = 0;
        int zeroes = 0, ones = 0;
        int longest = 0;

        while (right < n) {
            if (arr[right] == 0)
                zeroes++;
            else
                ones++;
            // shrink window to remove zero
            while (zeroes > 1) {
                if (arr[left] == 0)
                    zeroes--;
                else
                    ones--;

                left++;
            }

            longest = max(longest, right - left);
            right++;
        }
        // case all 1s [1,1,1] we must delete so we delete 1 1
        return zeroes == 0 ? ones - 1 : longest;
    }
};
```





Amazon 49 | Microsoft 29 | Facebook 23

Bloomberg 23 | Apple 16 | Google 13

Spotify 10 | Adobe 6 | Uber 5 | VMware 5

Yahoo 4 | Goldman Sachs 4 | Oracle 4

Walmart Global Tech 3 | Salesforce 3 | Paypal 2

Samsung 2 | Intuit 2 | Yandex 2 | Zoho 2

JPMorgan 2



### 3. Longest Substring Without Repeating Characters

Medium 35599 1590 Add to List Share

Given a string `s`, find the length of the **longest substring** without repeating characters.



**Example 1:**

```
Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.
```

**Example 2:**

```
Input: s = "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.
```

**Example 3:**

```
Input: s = "pwwkew"
Output: 3
Explanation: The answer is "wke", with the length of 3.
Notice that the answer must be a substring, "pwke" is a
subsequence and not a substring.
```

```cpp
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int n = s.size();
        int left = 0, right = 0;
        int len = 0, mxLen = 0;
        unordered_map<char, int> mp;

        for(int right = 0; right < n; right++ ) {
            char rightChar = s[right];
            mp[rightChar]++;

        // left char ki freq more thn one so shrink from left
            while(mp[rightChar] > 1) {
                char leftChar = s[left];
                mp[leftChar]--;
                left++;
            }

            mxLen = max(mxLen, right- left + 1);
        }

        return mxLen;
    }
};
```



Google 17 | Amazon 3 | Adobe 2

## 1423. Maximum Points You Can Obtain from Cards

Medium | 5514 | 202 | Add to List | Share

There are several cards **arranged in a row**, and each card has an associated number of points. The points are given in the integer array `cardPoints`.

In one step, you can take one card from the beginning or from the end of the row. You have to take exactly `k` cards.

Your score is the sum of the points of the cards you have taken.

Given the integer array `cardPoints` and the integer `k`, return the *maximum score* you can obtain.

**Example 1:**

```
Input: cardPoints = [1,2,3,4,5,6,1], k = 3
Output: 12
Explanation: After the first step, your score will always be 1. However, choosing the rightmost card first will maximize your total score. The optimal strategy is to take the three cards on the right, giving a final score of 1 + 6 + 5 = 12.
```

**Example 2:**

```
Input: cardPoints = [2,2,2], k = 2
Output: 4
Explanation: Regardless of which two cards you take, your score will always be 4.
```

**Example 3:**

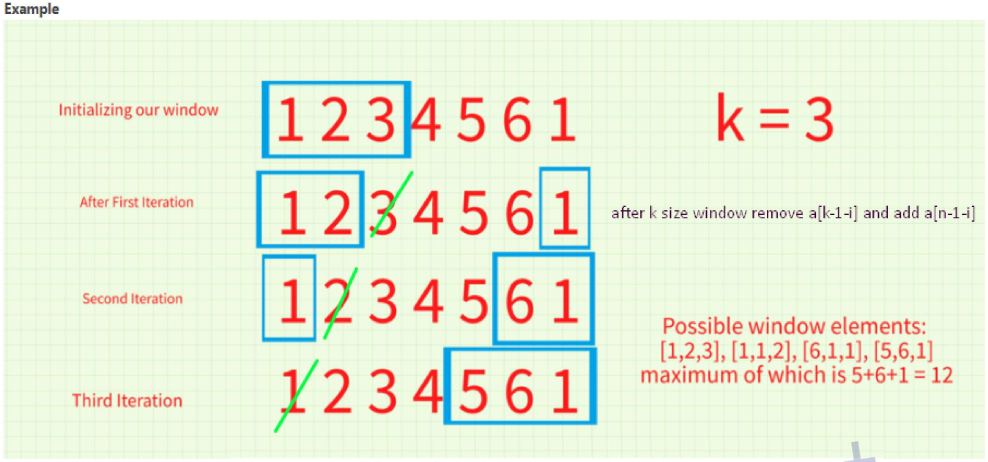
```
Input: cardPoints = [9,7,7,9,7,7,9], k = 7
Output: 55
Explanation: You have to take all the cards. Your score is the sum of points of all cards.
```

**Intuition**

The idea here is pretty tricky to guess...

- What we do is we initialize a window of size k...
- Now, since we can select either from the start or from the end, we only have access to either the first k items or last k items, and hence we are trying to limit our access using this window...
- So, we include all the elements from start in our window, till its full...
- The sum of elements at each instance in our window will be kept track of using another variable that will store our result.
- Now, we remove the last element from our window, and add the last unvisited element of our cardPoints array... Similarly we keep on removing 1 element from our window and start adding the last unvisited element of our cardPoints array...
- We keep doing this until we reach the start of our array, in this way we have covered all our possible picks...

**Example**




In our possible windows, we can see that we are covering all the possible picks in our given array using 3 cards...

```
class Solution {
public:
    int maxScore(vector<int>& points, int k) {
        int n = points.size();
        long long currSum = 0;
        for(int i=0; i<k; i++)
            currSum += points[i];

        if(k == n)
            return currSum;

        long long maxPoints = currSum;

        for(int i=0; i<k; i++) {
            // remove window elm right to left
            currSum -= points[k-1-i];
            // add last k elms
            currSum += points[n-1-i];

            maxPoints = max(maxPoints, currSum);

        }

        return maxPoints;
    }
};
```

cleaner

Microsoft 2 | Adobe 2 | DE Shaw 1

## 1358. Number of Substrings Containing All Three Characters

Medium | 2613 | 41 | Add to List | Share

Given a string `s` consisting only of characters *a*, *b* and *c*.

Return the number of substrings containing **at least** one occurrence of all these characters *a*, *b* and *c*.

**Example 1:**

abc, abca, abcab, abcabc then one removed from left ptr cnt (n-1-validIdx) bca. bcab, bcabc, cab, cabc, abc

```
Input: s = "abcabc"
Output: 10
Explanation: The substrings containing at
least one occurrence of the characters a, b and c
are "abc", "abca", "abcab", "abcabc", "bca",
"bcab", "bcabc", "cab", "cabc" and "abc" (again).
```

**Example 2:**

```
Input: s = "aaacb"
Output: 3
Explanation: The substrings containing at
least one occurrence of the characters a, b and c
are "aaacb", "aacb" and "acb".
```

**Example 3:**

```
Input: s = "abc"
Output: 1
```

The primary aim of the question is to find all substrings which satisfy the given conditions n(A),n(B),n(C) > 0. We can solve the problem by use of sliding window in th

1. Keep Expanding the sliding windows rightwards until the above condition is satisfied.
2. Now, the sliding window contains a substring which satisifies the required conditions. However, when counting the total no.of substrings we need to consider all those other strings which could have the sliding widow substring as their prefix.
   For eg: If we consider the string "aabcccc" and the sliding window "abc" we also need to count strings "abcc", "abccc" which have their prefix as "abc"
3. Once all substrings are counted, compress the sliding window leftwards and continue this process.

```cpp
class Solution {
public:
    int numberOfSubstrings(string s) {

        unordered_map<char, int> mp;
        int end = s.size() - 1;
        int left = 0, right = 0, count = 0;

        while(right <= end) {

            mp[s[right]]++;
            while(mp['a'] and mp['b'] and mp['c']) {
        // endPtr means aage ki sari substr jiska pref abc hai
                count += (end - right + 1);
        // since aage ki substr count ho chuki hai to left shrink
                mp[s[left]]--;
                left++;
            }

            right++;
        }

        return count;
    }
};
```

# 1838. Frequency of the Most Frequent Element

Medium 2632 73 Add to List Share

The **frequency** of an element is the number of times it occurs in an array.

You are given an integer array `nums` and an integer `k`. In one operation, you can choose an index of `nums` and increment the element at that index by `1`.

Return *the **maximum possible frequency** of an element after performing **at most** `k` operations.*

**Example 1:**

```
Input: nums = [1,2,4], k = 5
Output: 3
Explanation: Increment the first element three times and the second element two times to make nums = [4,4,4].
4 has a frequency of 3.
```

**Example 2:**

```
Input: nums = [1,4,8,13], k = 5
Output: 2
Explanation: There are multiple optimal solutions:
- Increment the first element three times to make nums = [4,4,8,13]. 4 has a frequency of 2.
- Increment the second element four times to make nums = [1,8,8,13]. 8 has a frequency of 2.
- Increment the third element five times to make nums = [1,4,13,13]. 13 has a frequency of 2.
```

**Example 3:**

```
Input: nums = [3,9,6], k = 2
Output: 1
```

```cpp
class Solution {
public:
    int maxFrequency(vector<int>& nums, int k) {
        // sorting makes sure all elm on left are smaller
        sort(begin(nums), end(nums));
        int left = 0, right = 0;
        long long windowSum = 0;
        int maxlen = 0;
        while (right < nums.size()) {
            windowSum += nums[right];
            // 1,2,4 all we wanna make 4 4 4 so if 4*windowSize - prefSum < k
            // till that its valid else move
            while (windowSum + k < 1LL * nums[right] * (right - left + 1)) {
                windowSum -= nums[left];
                left++;
            }
            maxlen = max(maxlen, right - left + 1);
            right++;
        }
        return maxlen;
    }
};
```

all 4 4 4 ie windowSize*rightElm
becz that is greater so 1+2+3+(5) <= 12 OK





**Intuition**

Considering a number `x` from the given array.

1. Only the numbers smaller than `x` can be incremented to increase the frequency of `x`
2. To maximize the frequency, numbers closest to `x` should be incremented as they require less additions to reach the value `x`.

.

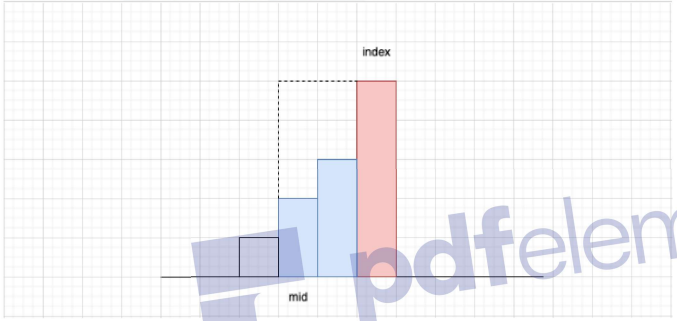
So our rough algo would be sorting the array and finding the max window for each element `x` in which its each elements can be incremented to the value equal to `x` with given steps `K`.

Further, deducing the condition for a valid window with `K` steps:

Let the size of window be `s` for an element `x` and sum of the window be `sum`, so if we increment each value of this window to x, the following should hold:
`s * x <= sum + K`.

**Final Algorithm**

1. Sort the array.
2. Maintain a prefix sum array to find the window `sum`.
3. Find the maximum window for each element.
4. Return the overall max of all windows.
5. **Follow Up:** For the `3rd` step we can apply `Binary Search` to find the maximum possible window size. (Furhter expained with comments in Code below)






Time: `O(NLogN)`
Space: `O(N)`

```cpp
class Solution {
public:
    bool check(int mid, vector<int>& nums, int k) {
        long long int windowsum=0,totalsum=0;
        for(int i=0;i<mid;++i)
            windowsum += nums[i];
        totalsum = 1LL*nums[mid-1]*mid;
        if(totalsum - windowsum <= k) return 1;

        int j = 0;      // SLIDING WINDOW
        for(int i=mid; i<nums.size(); i++) {
            windowsum -= nums[j];
            windowsum += nums[i];
            totalsum = 1LL*nums[i]*mid;

            if(totalsum - windowsum <= k)
                return 1;
            j++;
        }
        return 0;
    }
    // BINARY SEARCH
    int maxFrequency(vector<int>& nums, int k) {
        int lo = 1, hi = nums.size();
        sort(nums.begin(), nums.end());
        int ans;
        while(lo <= hi){
            int mid = lo + (hi-lo)/2;
            if(check(mid, nums, k)){
                ans = mid;
                lo = mid + 1;
            }
            else{
                hi = mid - 1;
            }
        }

        return ans;
    }
};
```

Google 7 | Amazon 6 | Uber 4 | Adobe 3

## 424. Longest Repeating Character Replacement

Medium | 8840 | 377 | Add to List | Share

You are given a string `s` and an integer `k`. You can choose any character of the string and change it to any other uppercase English character. You can perform this operation at most `k` times.

Return *the length of the longest substring containing the same letter you can get after performing the above operations.*

**Example 1:**

```
Input: s = "ABAB", k = 2
Output: 4
Explanation: Replace the two 'A's with two 'B's
or vice versa.
```

**Example 2:**

```
Input: s = "AABABBA", k = 1
Output: 4
Explanation: Replace the one 'A' in the middle
with 'B' and form "AABBBBA".
The substring "BBBB" has the longest repeating
letters, which is 4.
There may exists other ways to achive this answer
too.
```

```cpp
class Solution {
public:
    int characterReplacement(string s, int k) {
        vector<int> freq(26, 0);  //hash table
        int start = 0;
        int maxCount = 0;
        int maxLength = 0;

        for (int end = 0; end < s.length(); end++) {
            freq[s[end] - 'A']++;
            maxCount = max(maxCount, freq[s[end] - 'A']);

            int charReplacable = end - start + 1 - maxCount;
            if (charReplacable > k) {
                //sliding window condition
               freq[s[start] - 'A']--;
                start++;
            }

            maxLength = max(maxLength, end - start + 1);
        }

        return maxLength;
    }
};
```





## 2272. Substring With Largest Variance

Hard 1736 193 Add to List Share

The **variance** of a string is defined as the largest difference between the number of occurrences of **any** `2` characters present in the string. Note the two characters may or may not be the same.

Given a string `s` consisting of lowercase English letters only, return *the* ***largest variance*** *possible among all* ***substrings*** *of* `s`.

A **substring** is a contiguous sequence of characters within a string.

**Example 1:**

```
Input: s = "aababbb"
Output: 3
Explanation:
All possible variances along with their respective substrings are listed below:
- Variance 0 for substrings "a", "aa", "ab", "abab", "aababb", "ba", "b", "bb", and "bbb".
- Variance 1 for substrings "aab", "aba", "abb", "aabab", "ababb", "aababbb", and "bab".
- Variance 2 for substrings "aaba", "ababbb", "abbb", and "babb".
- Variance 3 for substring "babbb".
Since the largest possible variance is 3, we return it.
```



**Example 2:**

```
Input: s = "abcde"
Output: 0
Explanation:
No letter occurs more than once in s, so the variance of every substring is 0.
```

The difference of this implementation from other is: constant space complexity.
We can avoid additional arrays creation - just carefully implement Kadane's algo and take into account the algo details - commented in the code below.

1. Try every possible pair of chars { `highFreqChar`, `lowFreqChar` } - e.g., {'a','b'}
2. For the input string `s` calculate `highFreq` and `lowFreq`, find max `highFreq - lowFreq` using Kadane's algorithm. But according to the problem description, we must have at least 1 occurence of the `lowFreqChar` - so update the result only in these 2 cases:
   2.1 when `lowFreq > 0` - i.e., there is at least 1 `lowFreqChar` in current interval, our current interval result is `highFreq - lowFreq`.
   2.2 else (when `lowFreq == 0`) - but if we previously restarted the Kadane's interval (if `lowFreqAbandoned` == true) - we can extend back our interval for 1 `lowFreqChar`, and our current interval result is `highFreq - 1`.

```
int largestVariance(string s) {
    int result = 0;
    for (char highFreqChar = 'a'; highFreqChar <= 'z'; ++highFreqChar) {
        for (char lowFreqChar = 'a'; lowFreqChar <= 'z'; ++lowFreqChar) {
            if (highFreqChar == lowFreqChar) continue;

            int highFreq = 0;
            int lowFreq = 0;
            bool lowFreqAbandoned = false;

            for (const char& ch : s) {
                if (ch == highFreqChar) ++highFreq;
                if (ch == lowFreqChar) ++lowFreq;

                if (lowFreq > 0) {
                    result = max(result, highFreq - lowFreq);
                } else {
                    // Edge case: there are no `lowFreqChar` in current interval.
                    // In case if we re-started Kadane algo calculation -
                    // we can "extend" current interval with 1 previously abandoned 'lowFreqChar'
                    if (lowFreqAbandoned) {
                        result = max(result, highFreq - 1);
                    }
                }

                if (lowFreq > highFreq) {
                    // Kadane's algo calculation re-start: abandon previous chars and their freqs.
                    // Important: the last abandoned char is the `lowFreqChar` - this can be used on further iterations
                    lowFreq = 0;
                    highFreq = 0;
                    lowFreqAbandoned = true;
                }
            }
        }
    }
    return result;
}
```

Time complexity: O(26 * 26 * N) = O(N)
Space complexity: O(1)



Microsoft 4 | Cruise Automation 3 | Google 2 | Uber 2

## 218. The Skyline Problem

Hard | 5548 | 251 | Add to List | Share

A city's **skyline** is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Given the locations and heights of all the buildings, return *the **skyline** formed by these buildings collectively*.

The geometric information of each building is given in the array `buildings` where `buildings[i] = [left_i, right_i, height_i]`:

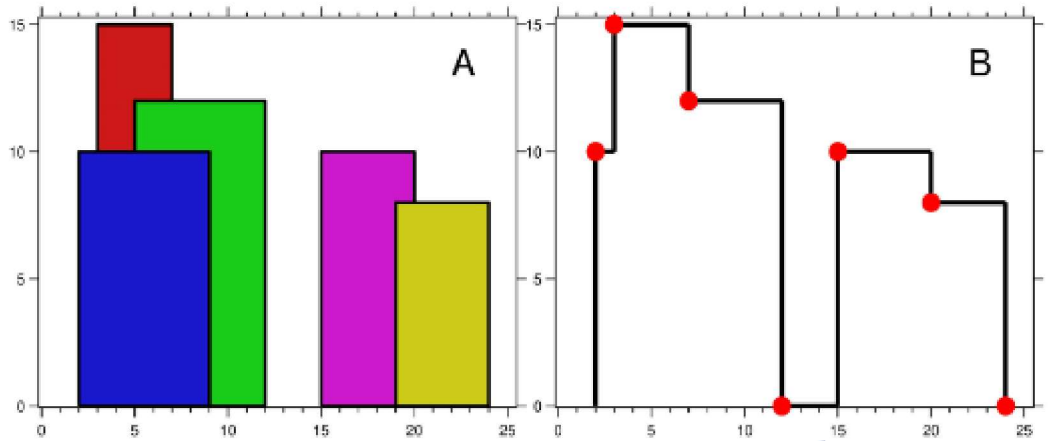
- `left_i` is the x coordinate of the left edge of the `i^th` building.
- `right_i` is the x coordinate of the right edge of the `i^th` building.
- `height_i` is the height of the `i^th` building.

You may assume all buildings are perfect rectangles grounded on an absolutely flat surface at height `0`.

The **skyline** should be represented as a list of "key points" **sorted by their x-coordinate** in the form `[[x1,y1],[x2,y2],...]`. Each key point is the left endpoint of some horizontal segment in the skyline except the last point in the list, which always has a y-coordinate `0` and is used to mark the skyline's termination where the rightmost building ends. Any ground between the leftmost and rightmost buildings should be part of the skyline's contour.

**Note:** There must be no consecutive horizontal lines of equal height in the output skyline. For instance, `[...,[2 3],[4 5],[7 5],[11 5],[12 7],...]` is not acceptable; the three lines of height 5 should be merged into one in the final output as such: `[...,[2 3],[4 5],[12 7],...]`

**Example 1:**




```
Input: buildings = [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]
Output: [[2,10],[3,15],[7,12],[12,0],[15,10],[20,8],[24,0]]
Explanation:
Figure A shows the buildings of the input.
Figure B shows the skyline formed by those buildings. The red points in
figure B represent the key points in the output list.
```

**Example 2:**

```
Input: buildings = [[0,2,3],[2,5,3]]
Output: [[0,3],[5,0]]
```

```cpp
class Solution {
public:
    vector<vector<int>> getSkyline(vector<vector<int>>& buildings) {
        vector<vector<int>> ans;

        vector<pair<int, int>> points;

        for(auto b: buildings){
            points.push_back({b[0], -b[2]});
            points.push_back({b[1], b[2]});
            // becz keyPoints are either start or end
        }

        sort(points.begin(), points.end());
        multiset<int> pq{0};   // dups reason
        int ongoingHeight = 0;

        // points.first x cord, points.sec height
        for(int i = 0; i < points.size(); i++){
            int currentPoint = points[i].first;
            int heightAtCurrentPoint = points[i].second;

            if(heightAtCurrentPoint < 0){
                pq.insert(-heightAtCurrentPoint);
            }else {
                pq.erase(pq.find(heightAtCurrentPoint));
            }
            // now get the top height and match with prev
            // if same it means horizontal end and diff means up or down
            auto maxHeight = *pq.rbegin();
            if(ongoingHeight != maxHeight){
                ongoingHeight = maxHeight;
                ans.push_back({currentPoint, ongoingHeight});
            }
        }
        return ans;
    }
};
```