Prefix Sum

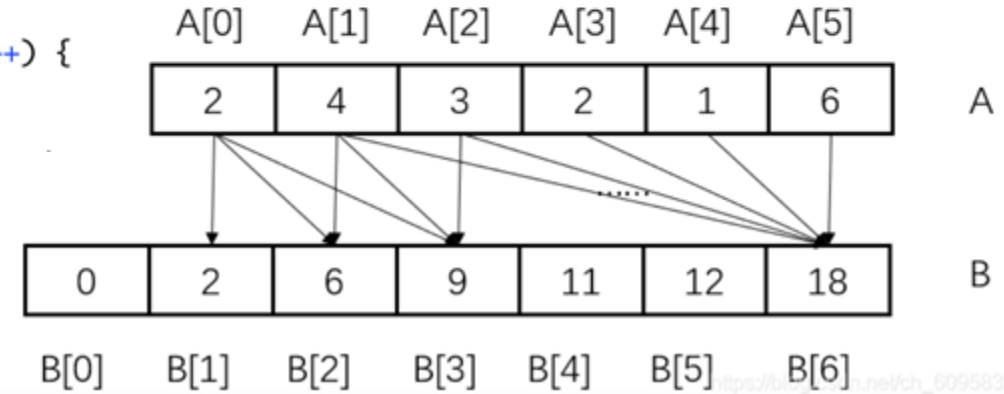
560. Subarray Sum Equals K
974. Subarray Sums Divisible by K
523. Continuous Subarray Sum
525. Contiguous Array
370. Range Addition
304. Range Sum Query 2D - Immutable
209. Minimum Size Subarray Sum
862. Shortest Subarray with Sum at Least K

PreFix Sum)

PrefixSum[I]= A[0]+A[1]+…+A[I-1],
Sum(i..j) = PrefixSum[j+1] - PrefixSum[I]

```
6   int[] sum = new int[nums.length];
7   sum[0] = nums[0];
8   for (int i = 1; i < nums.length; i++) {
9       sum[i] = sum[i - 1] + nums[i];
10  }
```

```
8   int[] sum = new int[N + 1];
9   for (int i = 0; i < N; i++)
10      sum[i + 1] = sum[i] + A[i];
```



2sum

There are over 50 2sum Serial Leetcode Problems.
diff = target - nums[i]

target
diff = nums[i] - target,

```
64 ▾    public int[] twoSum(int[] nums, int target) {
65          Map<Integer, Integer> map = new HashMap<>();
66 ▾        for (int i = 0; i < nums.length; i++) {
67              int diff = target - nums[i];
68              if (map.containsKey(diff)) return new int[]{map.get(diff), i};
69              map.put(nums[i], i);
70          }
71          return null;
72      }
73  }
```

1. Two Sum

Easy 22340 759 Add to List Share

Given an array of integers `nums` and an integer `target`, return *indices of the two numbers such that they add up to* `target`.

You may assume that each input would have ***exactly* one solution**, and you may not use the *same* element twice.

You can return the answer in any order.

Example 1:

```
Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Output: Because nums[0] + nums[1] == 9, we
return [0, 1].
```

Example 2:

```
Input: nums = [3,2,4], target = 6
Output: [1,2]
```

560. Subarray Sum Equals K

Medium 8078 271 Add to List Share

Given an array of integers `nums` and an integer `k`, return *the total number of continuous subarrays whose sum equals to* `k`.

Example 1:

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

Example 2:

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

Constraints:

- `1 <= nums.length <= 2 * 10^4`
- `-1000 <= nums[i] <= 1000`
- `-10^7 <= k <= 10^7`

```java
public class Solution {
    public int subarraySum(int[] nums, int k) {
        int count = 0;
        int[] sum = new int[nums.length + 1];
        sum[0] = 0;
        for (int i = 1; i <= nums.length; i++)
            sum[i] = sum[i - 1] + nums[i - 1];
        for (int start = 0; start < nums.length; start++) {
            for (int end = start + 1; end <= nums.length; end++) {
                if (sum[end] - sum[start] == k)
                    count++;
            }
        }
        return count;
    }
}
```

```java
class Solution {
// preSum - preSum = target, and check if map contains (preSum - target)
    public int subarraySum(int[] nums, int k) {
        Map<Integer,Integer> map= new HashMap<>();
        int sum = 0, res = 0;
        map.put(0,1); // prefix sum => frequency
        for(int num:nums){
            sum += num;
            if(map.containsKey(sum -k)) res += map.get(sum-k);
            map.put(sum,map.getOrDefault(sum,0) + 1);
        }
        return res;
    }
}
```

974. Subarray Sums Divisible by K

Medium 1819 122 Add to List Share

Given an array `nums` of integers, return the number of (contiguous, non-empty) subarrays that have a sum divisible by `k`.

Example 1:

```
Input: nums = [4,5,0,-2,-3,1], k = 5
Output: 7
Explanation: There are 7 subarrays with a sum divisible by k = 5:
[4, 5, 0, -2, -3, 1], [5], [5, 0], [5, 0, -2, -3], [0], [0, -2, -3], [-2, -3]
```

Note:

1. `1 <= nums.length <= 30000`
2. `-10000 <= nums[i] <= 10000`
3. `2 <= k <= 10000`

Same Remainers in i, j subarray

[4, 5, 0, -2, -3, 1] k= 5

[5] remainder 0
[5, 0] remainder 0
[-2,-3] remainder 0

[1, 2, 3, 4] k = 5
1, remainder 1
3, remainder 3
6, remainder 1 again [2,3] divisible by 5

```
class Solution {
    public int subarraysDivByK(int[] nums, int k) {
        Map<Integer,Integer> count = new HashMap<>();
        count.put(0,1);
        int prefix = 0, res=0;
        for(int a: nums){
            prefix = (prefix + a%k + k) %k;  // make sure k is positive
            int RemainderCount = count.getOrDefault(prefix, 0);
            res += RemainderCount;
            count.put(prefix,RemainderCount + 1);
        }
        return res;
    }
}
```

523. Continuous Subarray Sum

Medium 454 84 Add to List Share

Given an integer array `nums` and an integer `k`, return `true` *if* `nums` *has a continuous subarray of size* ***at least two*** *whose elements sum up to a multiple of* `k`, *or* `false` *otherwise.*

An integer `x` is a multiple of `k` if there exists an integer `n` such that `x = n * k`. `0` is **always** a multiple of `k`.

Example 1:

```
Input: nums = [23,2,4,6,7], k = 6
Output: true
Explanation: [2, 4] is a continuous subarray of size 2
whose elements sum up to 6.
```

Example 2:

```
Input: nums = [23,2,6,4,7], k = 6
Output: true
Explanation: [23, 2, 6, 4, 7] is an continuous
subarray of size 5 whose elements sum up to 42.
42 is a multiple of 6 because 42 = 7 * 6 and 7 is an
integer.
```

```java
class Solution {
    public boolean checkSubarraySum(int[] nums, int k) {
        int sum = 0;
        Map<Integer,Integer> map = new HashMap<>(); // remainder -> index
        map.put(0,-1);
        for(int i=0;i<nums.length; i++){
            sum += nums[i];
            if(k!=0) sum %=k;
            if(map.containsKey(sum)){ if(i - map.get(sum) > 1) return true; }
            else map.put(sum,i);
        }
        return false;
    }
}
```

525. Contiguous Array

Medium 3012 148 Add to List Share

Given a binary array `nums`, return *the maximum length of a contiguous subarray with an equal number of* `0` *and* `1`.

Example 1:

Input: nums = [0,1]
Output: 2
Explanation: [0, 1] is the longest contiguous subarray with an equal number of 0 and 1.

Input: nums = [0,1,0]
Output: 2
Explanation: [0, 1] (or [1, 0]) is a longest contiguous subarray with equal number of 0 and 1.

```java
public int findMaxLength(int[] nums) {
    int maxlen = 0;
    for (int start = 0; start < nums.length; start++) {
        int zeroes = 0, ones = 0;
        for (int end = start; end < nums.length; end++) {
            if (nums[end] == 0) zeroes++;
            else ones++;
            if (zeroes == ones)
                maxlen = Math.max(maxlen, end - start + 1);
        }
    }
    return maxlen;
}
```

O(N^2)

O(N)

```java
class Solution {
    public int findMaxLength(int[] nums) {
        for(int i = 0;i<nums.length; i++) if(nums[i]==0) nums[i] = -1;
        int res=0, sum = 0;
        Map<Integer,Integer> map = new HashMap<>();  map.put(0, -1);
        for(int i = 0;i<nums.length; i++){
            sum += nums[i];
            if(map.containsKey(sum)) res = Math.max(res, i - map.get(sum));
            else map.put(sum,i);
        }
        return res;
```

370. Range Addition

Medium 685 27 Add to List Share

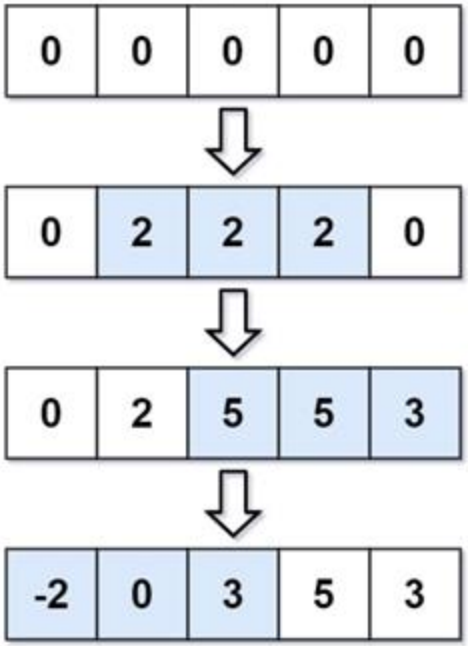
You are given an integer `length` and an array `updates` where `updates[i] = [startIdx_i, endIdx_i, inc_i]`.

You have an array `arr` of length `length` with all zeros, and you have some operation to apply on `arr`. In the `i^th` operation, you should increment all the elements `arr[startIdx_i], arr[startIdx_i + 1], ..., arr[endIdx_i]` by `inc_i`.

Return `arr` *after applying all the* `updates`.

```
3   public int[] getModifiedArray(int length, int[][] updates) {
4       int[] res = new int[length];
5       for(int[] update : updates) {
6           int value = update[2];
7           int start = update[0];
8           int end = update[1];
9           res[start] += value;
10          if (end < length - 1) res[end + 1] -= value;
11
12      }
13      int sum = 0;
14      for (int i = 0; i < length; i++) {
15          sum += res[i];
16          res[i] = sum;
17      }
18      return res;
19  }
```

https://leetcode.com/articles/a-recursive-approach-to-segment-trees-range-sum-queries-lazy-propagation/



Input: length = 5, updates = [[1,3,2],[2,4,3],[0,2,-2]]
Output: [-2,0,3,5,3]

[1 , 3 , 2] , [2, 3, 3] (length = 5)

res[0, 2, ,0, 0 -2]

res[0 ,2, 3, 0, -5]

sum 0, 2, 5, 5, 0

res[0, 2, 5, 5, 0]

304. Range Sum Query 2D - Immutable

Medium 1855 220 Add to List Share

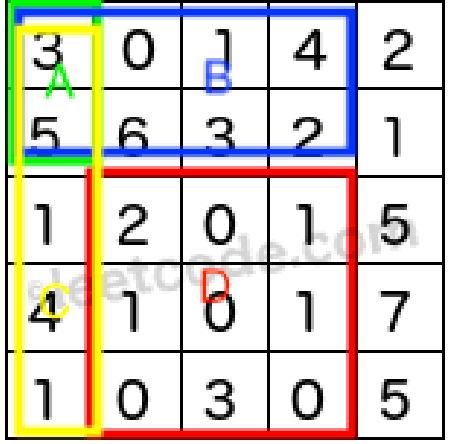
Given a 2D matrix `matrix`, handle multiple queries of the following type:

1. Calculate the **sum** of the elements of `matrix` inside the rectangle defined by its **upper left corner** `(row1, col1)` and **lower right corner** `(row2, col2)`.

Implement the NumMatrix class:

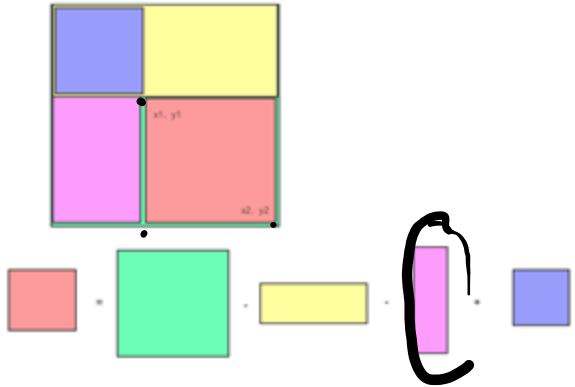
- `NumMatrix(int[][] matrix)` Initializes the object with the integer matrix `matrix`.
- `int sumRegion(int row1, int col1, int row2, int col2)` Returns the **sum** of the elements of `matrix` inside the rectangle defined by its **upper left corner** `(row1, col1)` and **lower right corner** `(row2, col2)`.

3	0	1	4	2
5	6	3	2	1
1	2	0	1	5
4	1	0	1	7
1	0	3	0	5




```
Input
["NumMatrix", "sumRegion", "sumRegion", "sumRegion"]
[[[[3, 0, 1, 4, 2], [5, 6, 3, 2, 1], [1, 2, 0, 1, 5], [4, 1, 0, 1, 7], [1, 0, 3, 0, 5]]], [2, 1, 4, 3], [1, 1, 2, 2], [1, 2, 2, 4]]
Output
[null, 8, 11, 12]
```

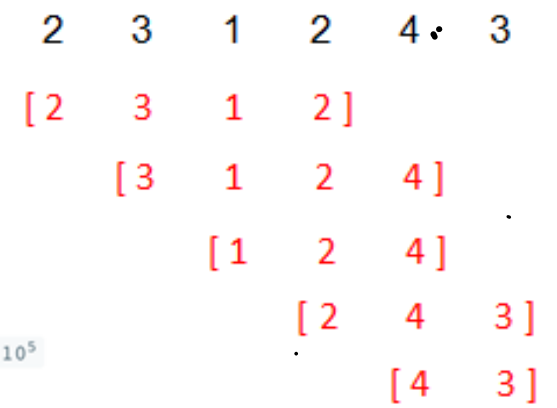
```java
    int[][] sums;
    public NumMatrix(int[][] matrix) {
        int row = matrix.length, col = matrix[0].length;
        sums = new int[row + 1][col + 1];
        for (int i = 0; i < row; i++)
            for (int j = 0; j < col; j++)
                sums[i + 1][j + 1] = sums[i][j + 1] + sums[i + 1][j]
                                   + matrix[i][j] - sums[i][j];
    }
    public int sumRegion(int row1, int col1, int row2, int col2) {
        return sums[row2 + 1][col2 + 1] - sums[row1][col2 + 1] - sums[row2 + 1][col1]
             + sums[row1][col1];
    }
}
```



209. Minimum Size Subarray Sum

Medium 4127 146 Add to List Share

Given an array of positive integers `nums` and a positive integer `target`, return the minimal length of a **contiguous subarray** `[nums_l, nums_{l+1}, ..., nums_{r-1}, nums_r]` of which the sum is greater than or equal to `target`. If there is no such subarray, return `0` instead.



Constraints:

- `1 <= target <= 10^9`
- `1 <= nums.length <= 10^5`
- `1 <= nums[i] <= 10^5`

Example 1:

```
Input: target = 7, nums = [2,3,1,2,4,3]
Output: 2
Explanation: The subarray [4,3] has the minimal length under the problem constraint.
```

Example 2:

```
Input: target = 4, nums = [1,4,4]
Output: 1
```

Example 3:

```
Input: target = 11, nums = [1,1,1,1,1,1,1,1]
Output: 0
```

```
4  public int minSubArrayLen(int target, int[] A) {
5      int left = 0, N = A.length, res = Integer.MAX_VALUE, sum = 0;
6      for (int i = 0; i < N; i++) {
7          sum += A[i];
8          while (sum >= target) {
9              res = Math.min(res, i - left + 1);
10             sum -= A[left++];
11         }
12     }
13     return res == Integer.MAX_VALUE ? 0 : res;
14 }
```

862. Shortest Subarray with Sum at Least K

Hard 1955 45 Add to List Share

Return the **length** of the shortest, non-empty, contiguous subarray of `nums` with sum at least `k`.

If there is no non-empty subarray with sum at least `k`, return `-1`.

Example 1:

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

Example 2:

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

Example 3:

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

Note:

1. `1 <= nums.length <= 50000`
2. `-10^5 <= nums[i] <= 10^5`
3. `1 <= k <= 10^9`

Approach 1: Sliding Window

Intuition

We can rephrase this as a problem about the prefix sums of `A`. Let `P[i] = A[0] + A[1] + ... + A[i-1]`. We want the smallest `y-x` such that `y > x` and `P[y] - P[x] >= K`.

Motivated by that equation, let `opt(y)` be the largest `x` such that `P[x] <= P[y] - K`. We need two key observations:

- If `x1 < x2` and `P[x2] <= P[x1]`, then `opt(y)` can never be `x1`, as if `P[x1] <= P[y] - K`, then `P[x2] <= P[x1] <= P[y] - K` but `y - x2` is smaller. This implies that our candidates `x` for `opt(y)` will have increasing values of `P[x]`.
- If `opt(y1) = x`, then we do not need to consider this `x` again. For if we find some `y2 > y1` with `opt(y2) = x`, then it represents an answer of `y2 - x` which is worse (larger) than `y1 - x`.

Algorithm

Maintain a "monoqueue" of indices of `P`: a deque of indices `x_0, x_1, ...` such that `P[x_0], P[x_1], ...` is increasing.

When adding a new index `y`, we'll pop `x_i` from the end of the deque so that `P[x_0], P[x_1], ..., P[y]` will be increasing.

If `P[y] >= P[x_0] + K`, then (as previously described), we don't need to consider this `x_0` again, and we can pop it from the front of the deque.

```
5  public int shortestSubarray(int[] A, int k) {
6      int N = A.length, res = N + 1;
7
8      int[] sum = new int[N + 1];
9      for (int i = 0; i < N; i++)
10         sum[i + 1] = sum[i] + A[i];
11
12     Deque<Integer> q = new ArrayDeque<>();
13     for (int i = 0; i < N + 1; i++) {
14         while (!q.isEmpty() && sum[q.peekLast()] >= sum[i]) q.pollLast();
15         while (!q.isEmpty() && sum[i] - sum[q.peekFirst()] >= k)
16             res = Math.min(res, i - q.pollFirst());
17         q.offerLast(i);
18     }
19     return res <= N ? res : -1;
20 }
```