3. Longest Substring Without Repeating Characters

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.
```

Example 4:

```
Input: s = ""
Output: 0
```

Constraints:

- `0 <= s.length <= 5 * 10^4`
- `s` consists of English letters, digits, symbols and spaces.

My soln

```
class Solution {
    public int lengthOfLongestSubstring(String s) {
        int dp[]=new int[s.length()];
        HashMap<Character,Integer> h=new HashMap<Character,Integer>();
        if(s.length()==0)
            return 0;
        dp[0]=1;
        h.put(s.charAt(0),0);
        int index=0;
        for(int i=1;i<s.length();i++)
        {
            if(h.containsKey(s.charAt(i)))
            {
                 dp[i]=Math.min(i-h.get(s.charAt(i)),dp[i-1]+1);
/*
This line ensures that if character already occurs then i-h.get() will be consider
but minimum is used as it is possible that other character may repeat in between this.
i.e if character already found before length of dp[i-1]+1 then we should take dp[i-1]+1(for
current place)
consider example 1 2 3 4 9 8 8 4
consider 4 now for already found at index 3 when we iterate for i=7 then i-h.get() will beco
me 3 but 8 is repaeting hence we have to take min()
*/
            }
            else
            {
                dp[i]=dp[i-1]+1;
            }
            h.put(s.charAt(i),i);
        }
        int m=dp[0];
        for(int i=1;i<s.length();i++)
        {
            m=Math.max(m,dp[i]);
        }
        return m;
    }
}
```

Sliding window

```
public class Solution {
    public int lengthOfLongestSubstring(String s) {
        int n = s.length(), ans = 0;
        Map<Character, Integer> map = new HashMap<>(); // current index of character
        // try to extend the range [i, j]
        for (int j = 0, i = 0; j < n; j++) {
            if (map.containsKey(s.charAt(j))) {
                i = Math.max(map.get(s.charAt(j)), i);
            }
            ans = Math.max(ans, j - i + 1);
            map.put(s.charAt(j), j + 1);
        }
        return ans;
    }
}
```

---

53. Maximum Subarray

Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest sum and return *its sum*.

Follow up: If you have figured out the `O(n)` solution, try coding another solution using the **divide and conquer** approach, which is more subtle.

Example 1:

```
Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: [4,-1,2,1] has the largest sum = 6.
```

Example 2:

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

Example 3:

```
Input: nums = [0]
Output: 0
```

Example 4:

```
Input: nums = [-1]
Output: -1
```

Example 5:

```
Input: nums = [-2147483647]
Output: -2147483647
```

Constraints:

- 1 <= nums.length <= $2 * 10^4$
- -2^{31} <= nums[i] <= $2^{31} - 1$

MY Soln Kadane's Algorithm

```
class Solution {
    public int maxSubArray(int[] nums) {
        int max=Integer.MIN_VALUE;
        int sum=0;
        for(int i=0;i<nums.length;i++)
        {
            sum=sum+nums[i];
            max=Math.max(max,sum);
            if(sum<0)
            {
                sum=0;
            }
        }
        return max;
    }
}
```

DP SOLN The key is to find the induction function from the bottom. Let's take a look at the problem...

```
[-2,1,-3,4,-1,2,1,-5,4]
My DP array setup:

int[] dp = new int[nums.length + 1];
dp[0] = 0;     // initialize
dp = [0,0,0,0,0,0,0,0,0,0]
```

So what is the induction rule?

// 1st iteration: i = 1; dp[1] = dp[0] + nums[0] = 0 + (-2) = -2 dp = [0,-2,0,0,0,0,0,0,0,0]

// 2nd iteration: i = 2; dp[2] = dp[1] + nums[1] = (-2) + 1 = -1 // is this the right solution for dp[2] ? No. // Another option for dp[2] dp[2] = nums[1] = 1; // Apparently 1 is greater than -1, let's update the dp array dp = [0,-2,1,0,0,0,0,0,0,0]

// 3rd iteration: i = 3; dp[3] = dp[2] + nums[2] = 1 + (-3) = -2
// Another option for dp[3] dp[3] = nums[2] = -3; // -2 is greater than -3, let's update the dp array dp = [0,-2,1,-2,0,0,0,0,0,0]

// 4th iteration: i = 4; dp[4] = dp[3] + nums[3] = (-2) + 4 = 2
// Another option for dp[4] dp[4] = nums[3] = 4; // 4 is greater than 2, let's update the dp array dp = [0,-2,1,-2,4,0,0,0,0,0] So at this point, what is the induction rule? if we look at the dp array, we can see that whenever dp[i-1] < 0, we take dp[i] = nums[i] instead of nums[i] + dp[i-1] Why? Because if dp[i-1] < 0, whatever a number + a negative number is always smaller than itself. e.g. A + B < A if B < 0 We want the sum to be maximum, this is the goal. If dp[i-1] > 0, then it is possible that num[i] > 0, we want to keep that way. But we are sure that if dp[i-1] < 0, we definitely don't want dp[i-1] + A < A. That's how the induction rule comes.

So the induction is as following:

if (dp[i-1] < 0) dp[i] = nums[i] else dp[i] = nums[i] + dp[i-1] // make sure at each iteration we record and update maxSum = dp[i] Solution:

```
public static int maxSubArray(int[] nums) {
        if(nums.length == 0 || nums == null) return 0;
        // initialize
        int n = nums.length;
        int[] dp = new int[n + 1];
        dp[0] = 0;
        int res = Integer.MIN_VALUE;
        // dp induction rules
        for(int i = 1; i < dp.length; i++){
            if(dp[i-1] > 0) dp[i] = dp[i-1] + nums[i-1];
            else dp[i] = nums[i-1];
            res = Math.max(res, dp[i]);
        }
        return res;
    }
}
```

---

70. Climbing Stairs

You are climbing a stair case. It takes *n* steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Example 1:

```
Input: 2
Output: 2
Explanation: There are two ways to climb to the top.
1. 1 step + 1 step
2. 2 steps
```

Example 2:

```
Input: 3
Output: 3
Explanation: There are three ways to climb to the top.
1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step
```

Constraints:

- 1 <= n <= 45

My solution

```
class Solution {
    int dp[]=new int[46];
    Solution()
    {
        Arrays.fill(dp,-1);
    }
    public int climbStairs(int n) {
       if(n==0 || n==1)
           return 1;
        if(dp[n]!=-1)
            return dp[n];
        return dp[n]=(climbStairs(n-1)+climbStairs(n-2));
    }
}
```

Second soln

```
public class Solution {
    public int climbStairs(int n) {
        if (n == 1) {
            return 1;
        }
        int[] dp = new int[n + 1];
        dp[1] = 1;
        dp[2] = 2;
        for (int i = 3; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }
        return dp[n];
    }
}
```

Fibonnaci

```
public class Solution {
    public int climbStairs(int n) {
        if (n == 1) {
            return 1;
        }
        int first = 1;
        int second = 2;
        for (int i = 3; i <= n; i++) {
            int third = first + second;
            first = second;
            second = third;
        }
        return second;
    }
}
```

Approach 5: Binets Method Algorithm log(n)

```
 public class Solution {
    public int climbStairs(int n) {
        int[][] q = {{1, 1}, {1, 0}};
        int[][] res = pow(q, n);
        return res[0][0];
    }
    public int[][] pow(int[][] a, int n) {
        int[][] ret = {{1, 0}, {0, 1}};
        while (n > 0) {
            if ((n & 1) == 1) {
                ret = multiply(ret, a);
            }
            n >>= 1;
            a = multiply(a, a);
        }
        return ret;
    }
    public int[][] multiply(int[][] a, int[][] b) {
        int[][] c = new int[2][2];
        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 2; j++) {
                c[i][j] = a[i][0] * b[0][j] + a[i][1] * b[1][j];
            }
        }
        return c;
    }
}
```

Approach 6: Fibonacci Formula

```
public class Solution {
    public int climbStairs(int n) {
        double sqrt5=Math.sqrt(5);
        double fibn=Math.pow((1+sqrt5)/2,n+1)-Math.pow((1-sqrt5)/2,n+1);
        return (int)(fibn/sqrt5);
    }
}
```

---

198. House Robber

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight **without alerting the police**.

Example 1:

```
Input: nums = [1,2,3,1]
Output: 4
Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).
             Total amount you can rob = 1 + 3 = 4.
```

Example 2:

```
Input: nums = [2,7,9,3,1]
Output: 12
Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1).
             Total amount you can rob = 2 + 9 + 1 = 12.
```

Constraints:

- `0 <= nums.length <= 100`
- `0 <= nums[i] <= 400`

My Solution

```
class Solution {
    public int rob(int[] nums) {
        int n=nums.length;
        int dp[]=new int[n+1];
        if(n==0)
            return dp[0];
        dp[1]=nums[0];
        for(int i=1;i<n;i++)
        {
            dp[i+1]=Math.max(dp[i],(dp[i-1]+nums[i]));
        }
        return dp[n];
    }
}
```

Solution with explanation

```
class Solution {
    public int rob(int[] nums) {
        int n = nums.length;
        // [] case
        if(n == 0) return 0;
        //[2] single element
        if(n ==1) return nums[0];
        if(n ==2) return Math.max(nums[0],nums[1]);

        // [1,2,3]

        int[]dp = new int[nums.length];
        //if  nums length is 1 then ans = nums[0]
        dp[0]= nums[0];
        dp[1] = Math.max(nums[0],nums[1]);
   // we store  solution for the problem size i int dp[i]
    //   It is a decision problem. We ask should we take nums[i]
    // a decision is usually solve by dp
   // consider problem of size 1
    //[1] = 1  if there only is a hous then  answer is that one element
    // so solution store solution into dp. dp[0]= nums[0];

    // we ask should we take 2
    //[1,2] =2  We take max of the both. If we take 2, then we have give up 1
     // dp[1] = Math.max(nums[0],nums[1])
    //  [1,2,3] = 4  should we take 3?
    // if we take 3 that means we not gonna take 2. Our max will be [1] case + current nums
[i];
        // dp[i-2]+ nums[i]
        // we we don't take 3. I will just be same as solution of[1,2] , and we already have
that  which is dp[i-1]

        // we solution for maxium profit should be  the max of the tow choices.
        // this formula will work for problem of atleast size 3
        //dp[i]= Math.max(dp[i-1],dp[i-2]+nums[i])

        for(int i = 2; i < n; i ++){
            dp[i]= Math.max(dp[i-1],dp[i-2]+nums[i]);

    }

        return dp[n-1];
}
}
```

Solution with two variables We can notice that in the previous step we use only memo[i] and memo[i-1], so going just 2 steps back. We can hold them in 2 variables instead. This optimization is met in Fibonacci sequence creation and some other problems [to paste links].

```
/* the order is: prev2, prev1, num  */
public int rob(int[] nums) {
    if (nums.length == 0) return 0;
    int prev1 = 0;
    int prev2 = 0;
    for (int num : nums) {
        int tmp = prev1;
        prev1 = Math.max(prev2 + num, prev1);
        prev2 = tmp;
    }
    return prev1;
}
```

---

204. Count Primes

Count the number of prime numbers less than a non-negative number, ***n***.

Example:

```
Input: 10
Output: 4
Explanation: There are 4 prime numbers less than 10, they are 2, 3, 5, 7.
```

Look at test case fo 499979 in for loop i* i is used to overflow conditions

---

303. Range Sum Query - Immutable

Given an integer array *nums*, find the sum of the elements between indices *i* and *j* (*i* ≤ *j*), inclusive.

Example:

```
Given nums = [-2, 0, 3, -5, 2, -1]

sumRange(0, 2) -> 1
sumRange(2, 5) -> -1
sumRange(0, 5) -> -3
```

Constraints:

- You may assume that the array does not change.
- There are many calls to *sumRange* function.
- `0 <= nums.length <= 10^4`
- `-10^5 <= nums[i] <= 10^5`

- `0 <= i <= j < nums.length`

Prefix Sum Array Soln

```
class NumArray {
    int ans[];
    public NumArray(int[] nums) {
        int sum=0;
        ans=new int[nums.length+1];
        for(int i=0;i<nums.length;i++)
        {
            ans[i+1]=nums[i]+sum;
            sum=ans[i+1];
        }
    }

    public int sumRange(int i, int j) {
        return ans[j+1]-ans[i];
    }
}

/**
 * Your NumArray object will be instantiated and called as such:
 * NumArray obj = new NumArray(nums);
 * int param_1 = obj.sumRange(i,j);
 */
```

HashMap soln

```
private Map<Pair<Integer, Integer>, Integer> map = new HashMap<>();

public NumArray(int[] nums) {
    for (int i = 0; i < nums.length; i++) {
        int sum = 0;
        for (int j = i; j < nums.length; j++) {
            sum += nums[j];
            map.put(Pair.create(i, j), sum);
        }
    }
}

public int sumRange(int i, int j) {
    return map.get(Pair.create(i, j));
}
```

338. Counting Bits

Given a non negative integer number **num**. For every numbers **i** in the range **0 ≤ i ≤ num** calculate the number of 1's in their binary representation and return them as an array.

Example 1:

```
Input: 2
Output: [0,1,1]
```

Example 2:

```
Input: 5
Output: [0,1,1,2,1,2]
```

Follow up:

- It is very easy to come up with a solution with run time **O(n*sizeof(integer))**. But can you do it in linear time **O(n)** /possibly in a single pass?
- Space complexity should be **O(n)**.
- Can you do it like a boss? Do it without using any builtin function like **__builtin_popcount** in c++ or in any other language.

My solution Look up Table Intuition behind filling table: ans[i]= ans[i>>1] + (i&1) ans[i] = part 1 part2 consider 13= 000110 1 13= part 1 part 2 part 1 is basically 13/2 i.e. 13>>1 part 2 to check last bit is set or not.

```
class Solution {
    int table[];
    Solution()
    {
        table=new int[256];
        table[0]=0;
        for(int i=1;i<256;i++)
            table[i]=(i&1)+table[i/2];
    }
    public int[] countBits(int num) {
        int ans[]=new int[num+1];
        for(int i=0;i<=num;i++)
        {
            ans[i]=table[i & 0xFF];
            ans[i]+=table[i>>8 & 0xFF];
            ans[i]+=table[i>>16 & 0xFF];
            ans[i]+=table[i>>24 & 0xFF];
        }
        return ans;
    }
}
```

Better solution

```
class Solution {
    public int[] countBits(int num) {
        int ans[]=new int[num+1];
        for(int i=1;i<=num;i++)
        {
            ans[i]=ans[i>>1]+(i&1); //use bracket to follow bodmass rule
        }
        return ans;
    }
}
```

**Easy to understand **

```
class Solution {
    public int[] countBits(int num) {
        int[] dp=new int[num+1];
        dp[0]=0;
        for(int i=1;i<=num;i++){
            if(i%2==0){
                dp[i]=dp[i/2];
            }
            else dp[i]=1+dp[i/2];
        }
        return dp;
    }
}
```

---

392. Is Subsequence

Given a string **s** and a string **t**, check if **s** is subsequence of **t**.

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, `"ace"` is a subsequence of `"abcde"` while `"aec"` is not).

Follow up:
If there are lots of incoming S, say S1, S2, ... , Sk where k >= 1B, and you want to check one by one to see if T has its subsequence. In this scenario, how would you change your code?

Credits:
Special thanks to @pbrother (https://leetcode.com/pbrother/) for adding this problem and creating all test cases.

Example 1:

```
Input: s = "abc", t = "ahbgdc"
Output: true
```

Example 2:

```
Input: s = "axc", t = "ahbgdc"
Output: false
```

Constraints:

- `0 <= s.length <= 100`
- `0 <= t.length <= 10^4`
- Both strings consists only of lowercase characters.

Solution

```
class Solution {
    public boolean isSubsequence(String s, String t) {
        int dp[]=new int[s.length()];
        int k=0;
        for(int i=0;i<s.length();i++)
        {
            for(int j=k;j<t.length();j++)
            {
                if(s.charAt(i)==t.charAt(j))
                {
                    k=j+1;
                    dp[i]=1;
                    break;
                }
            }
        }
        for(int i=0;i<s.length();i++)
        {
            if(dp[i]!=1)
                return false;
        }
        return true;
    }
}
```

Two pointer Approach

```
class Solution {
    public boolean isSubsequence(String s, String t) {
        if(s.length()==0)
            return true;
        int i,j;
        for(i=0,j=0;i<t.length()&& j<s.length();i++)
        {
            if(s.charAt(j)==t.charAt(i))
                j++;
        }
        if(j==s.length())
            return true;
        return false;
    }
}
```

---

746. Min Cost Climbing Stairs

On a staircase, the `i`-th step has some non-negative cost `cost[i]` assigned (0 indexed).

Once you pay the cost, you can either climb one or two steps. You need to find minimum cost to reach the top of the floor, and you can either start from the step with index 0, or the step with index 1.

Example 1:

```
Input: cost = [10, 15, 20]
Output: 15
Explanation: Cheapest is start on cost[1], pay that cost and go to the top.
```

Example 2:

```
Input: cost = [1, 100, 1, 1, 1, 100, 1, 1, 100, 1]
Output: 6
Explanation: Cheapest is start on cost[0], and only step on 1s, skipping cost[3].
```

Note:

1. `cost` will have a length in the range `[2, 1000]`.
2. Every `cost[i]` will be an integer in the range `[0, 999]`.

Solution :::::::::

```
int solve1(int cost[],int n,int dp[])
    {
        if(n==0 || n==1) return dp[n]=0;
        if(dp[n]!=-1) return dp[n];
        return dp[n]=Math.min(solve1(cost,n-1,dp)+cost[n-1],solve1(cost,n-2,dp)+cost[n-2]);
    }
```

```
class Solution {
    public int minCostClimbingStairs(int[] cost) {
        int n=cost.length;
        int f1=cost[0],f2=cost[1];
        for(int i=2;i<n;i++)
        {
            int var=cost[i]+Math.min(f1,f2);
            f1=f2;
            f2=var;
        }
        return Math.min(f1,f2);
    }
}
```