

DATA STRUCTURES AND ALGORITHMS EASY PROBLEMS

By Jani Ahamed Habeeb Mohamed

Table of Contents

| | |
|---|-----------|
| 1. ARRAYS..... | 4 |
| 1.1 Remove Duplicates From Sorted Array | 4 |
| 1.2 Best Time To Buy And Sell Stocks II | 5 |
| 1.3 Rotate Array..... | 7 |
| 1.4 Contains Duplicate | 9 |
| 1.5 Single Number | 10 |
| 1.6 Intersection Of Two Arrays II | 11 |
| 1.7 Plus One | 13 |
| 1.8 Move Zeroes | 15 |
| 1.9 Two Sum..... | 17 |
| 1.10 Valid Sudoku | 18 |
| 1.11 Rotate Image..... | 22 |
| 2. STRINGS..... | 25 |
| 2.1 Reverse String..... | 25 |
| 2.2 Reverse Integer..... | 26 |
| 2.3 First Unique Character In A String | 27 |
| 2.4 Valid Anagram | 28 |
| 2.5 Valid Palindrome | 29 |
| 2.6 String To Integer (atoi) | 31 |
| 2.7 StrStr Implementation | 34 |
| 2.8 Longest Common Prefix | 36 |
| 3. LINKED LIST..... | 39 |
| 3.1 Delete Node in Linked List..... | 39 |
| 3.2 Remove Nth Node From End Of List..... | 41 |
| 3.3 Reverse Linked List | 42 |
| 3.4 Merge Two Sorted Lists | 44 |
| 3.5 Palindrome Linked List..... | 46 |
| 3.6 Linked List Cycle | 48 |
| 4. TREES..... | 51 |
| 4.1 Maximum Depth Of Binary Tree | 51 |
| 4.2 Validate Binary Search Tree..... | 52 |
| 4.3 Symmetric Tree..... | 54 |
| 4.4 Binary Tree Level Order Traversal..... | 56 |

| | | |
|-----|---|-----------|
| 4.5 | Sorted Array To Binary Search Tree..... | 60 |
| 5. | SEARCHING AND SORTING | 63 |
| 5.1 | Merge Two Sorted Arrays..... | 63 |
| 5.2 | First Bad Version..... | 64 |
| 6. | DESIGN | 67 |
| 6.1 | Shuffle An Array | 67 |
| 6.2 | MinStack | 69 |
| 7. | MATH | 73 |
| 7.1 | Fizz Buzz..... | 73 |
| 7.2 | Count Primes | 74 |
| 7.3 | Power Of Three..... | 76 |
| 7.4 | Roman To Integer..... | 77 |
| 8. | OTHERS | 81 |
| 8.1 | Number Of 1 Bits..... | 81 |
| 8.2 | Hamming Distance | 82 |
| 8.3 | Reverse Bits..... | 83 |
| 8.4 | Pascals Triangle | 84 |
| 8.5 | Valid Prantheses | 86 |
| 8.6 | Missing Number | 89 |
| 9. | DYNAMIC PROGRAMMING | 91 |
| 9.1 | Climbing Stairs | 91 |
| 9.2 | Best Time To Buy And Sell Stock | 92 |
| 9.3 | Maximum SubArray | 93 |
| 9.4 | House Robber | 94 |

1. ARRAYS

1.1 Remove Duplicates From Sorted Array

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];
}
```

If all assertions pass, then your solution will be accepted.

Example 1:

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

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

Explanation: Your function should return $k = 2$, with the first two elements of `nums` being 1 and 2 respectively.

It does not matter what you leave beyond the returned k (hence they are underscores).

Example 2:

Input: `nums = [0,0,1,1,1,2,2,3,3,4]`

Output: 5, `nums = [0,1,2,3,4,_,_,_,_,_]`

Explanation: Your function should return $k = 5$, with the first five elements of `nums` being 0, 1, 2, 3, and 4 respectively.

It does not matter what you leave beyond the returned k (hence they are underscores).

C:

```
int removeDuplicates(int* nums, int numsSize){
    int i = 0; int k = 1;
    for (i = 0; i < numsSize-1; i++) {
        if (nums[i] != nums[i+1]) {
            nums[k] = nums[i+1];
            k++;
        }
    }
    return k;
}
```

Java:

```
class Solution {
    public int removeDuplicates(int[] nums) {
        int i = 0; int k = 1;
        for (i = 0; i < nums.length-1; i++) {
            if (nums[i] != nums[i+1]) {
                nums[k] = nums[i+1];
                k++;
            }
        }
        return k;
    }
}
```

1.2 Best Time To Buy And Sell Stocks II

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

On each day, you may decide to buy and/or sell the stock. You can only hold at most one share of the stock at any time. However, you can buy it then immediately sell it on the same day.

Find and return the maximum profit you can achieve.

Example 1:

Input: prices = [7,1,5,3,6,4]

Output: 7

Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = $5 - 1 = 4$.

Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = $6 - 3 = 3$.

Total profit is $4 + 3 = 7$.

Example 2:

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

Output: 4

Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = $5 - 1 = 4$.

Total profit is 4.

Example 3:

Input: prices = [7,6,4,3,1]

Output: 0

Explanation: There is no way to make a positive profit, so we never buy the stock to achieve the maximum profit of 0.

C:

```
int maxProfit(int* prices, int pricesSize){
    int i = 0;
    int result = 0;
    for (i = 1; i < pricesSize; i++) {
        if (prices[i] > prices[i-1]) result+=(prices[i]-prices[i-1]) ;
    }
    return result;
}
```

JAVA:

```
class Solution {  
    public int maxProfit(int[] prices) {  
        int i = 0;  
        int profit = 0;  
        for (i = 1; i < prices.length; i++) {  
            if (prices[i] > prices[i-1]) {  
                profit += (prices[i]-prices[i-1]);  
            }  
        }  
        return profit;  
    }  
}
```

1.3 Rotate Array

Given an integer array nums, rotate the array to the right by k steps, where k is non-negative.

Example 1:

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

Output: [5,6,7,1,2,3,4]

Explanation:

rotate 1 steps to the right: [7,1,2,3,4,5,6]

rotate 2 steps to the right: [6,7,1,2,3,4,5]

rotate 3 steps to the right: [5,6,7,1,2,3,4]

Example 2:

Input: nums = [-1,-100,3,99], k = 2

Output: [3,99,-1,-100]

Explanation:

rotate 1 steps to the right: [99,-1,-100,3]

rotate 2 steps to the right: [3,99,-1,-100]

C:

```
void reverse ( int* nums, int start, int end) {
    while (start < end) {
        int temp = nums[start];
        nums[start] = nums[end];
        nums[end] = temp;
        start++;
        end--;
    }
}

void rotate(int* nums, int numsSize, int k){
    k = k % numsSize;
    reverse(nums, 0, numsSize-1);
    reverse(nums, 0, k-1);
    reverse(nums, k, numsSize-1);
}
```

JAVA:

```
class Solution {
    public void reverse(int[] nums, int start, int end) {
        int temp = 0;
        while (start <= end) {
            temp = nums[start];
            nums[start] = nums[end];
            nums[end] = temp;
            start++;
            end--;
        }
    }

    public void rotate(int[] nums, int k) {
        k %= nums.length;
        reverse(nums, 0, nums.length-1);
    }
}
```



```

        reverse(nums, 0, k-1);
        reverse(nums, k, nums.length-1);

    }
}

```

1.4 Contains Duplicate

Given an integer array `nums`, return `true` if any value appears at least twice in the array, and return `false` if every element is distinct.

Example 1:

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

Output: `true`

Example 2:

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

Output: `false`

Example 3:

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

Output: `true`

C:

```

bool containsDuplicate(int* nums, int numsSize){
    int temp = 0;
    int i = 0;
    int j = 0;
    for (i = 0; i < numsSize; i++) {
        for (j = i+1; j < numsSize; j++) {
            if (nums[i] == nums[j]) {
                return true;
            }
        }
    }
    return false;
}

```

```

    }
    }
}

return false;
}

```

JAVA:

```

class Solution {
    public boolean containsDuplicate(int[] nums) {
        HashSet<Integer> hs = new HashSet(nums.length);
        for (int i: nums) {
            if (!hs.contains(i)) hs.add(i);
            else return true;
        }
        return false;
    }
}

```

1.5 Single Number

Given a non-empty array of integers nums, every element appears twice except for one. Find that single one.

You must implement a solution with a linear runtime complexity and use only constant extra space.

Example 1:

Input: nums = [2,2,1]

Output: 1

Example 2:

Input: nums = [4,1,2,1,2]

Output: 4

Example 3:

Input: nums = [1]

Output: 1

C:

```
int singleNumber(int* nums, int numsSize){
    int i = 0;
    int result = 0;
    for (i = 0; i < numsSize; i++) {
        result ^= nums[i];
    }
    return result;
}
```

JAVA:

```
class Solution {
    public int singleNumber(int[] nums) {
        int result = 0;
        for (int i: nums) result ^= i;
        return result;
    }
}
```

1.6 Intersection Of Two Arrays II

Given two integer arrays nums1 and nums2, return an array of their intersection. Each element in the result must appear as many times as it shows in both arrays and you may return the result in any order.

Example 1:

Input: nums1 = [1,2,2,1], nums2 = [2,2]

Output: [2,2]

Example 2:

Input: nums1 = [4,9,5], nums2 = [9,4,9,8,4]

Output: [4,9]

Explanation: [9,4] is also accepted.

C:

```
int* intersect(int* nums1, int nums1Size, int* nums2, int nums2Size, int* returnSize) {  
    // Count the frequency of each element in nums1  
    int freq[1001] = {0}; // Assuming the maximum element value is 1000  
    for (int i = 0; i < nums1Size; i++) {  
        freq[nums1[i]]++;  
    }  
    int index = 0;  
    for (int i = 0; i < nums2Size; i++) {  
        if (freq[nums2[i]] > 0) {  
            nums1[index++] = nums2[i];  
            freq[nums2[i]]--;  
        }  
    }  
    // Set the returnSize to the size of the result array  
    *returnSize = index;  
    return nums1;  
}
```

JAVA:

```
import java.util.*;  
class Solution {  
    public int[] intersect(int[] nums1, int[] nums2) {  
        Arrays.sort(nums1);  
        Arrays.sort(nums2);  
        int i = 0, j = 0, k = 0;  
        while (i < nums1.length && j < nums2.length) {  
            if (nums1[i] < nums2[j]) {
```

```

        ++i;
    } else if (nums1[i] > nums2[j]) {
        ++j;
    } else {
        nums1[k++] = nums1[i++];
        ++j;
    }
}
return Arrays.copyOfRange(nums1, 0, k);
}
}

```

1.7 Plus One

You are given a large integer represented as an integer array `digits`, where each `digits[i]` is the *i*th digit of the integer. The digits are ordered from most significant to least significant in left-to-right order. The large integer does not contain any leading 0's.

Increment the large integer by one and return the resulting array of digits.

Example 1:

Input: `digits = [1,2,3]`

Output: `[1,2,4]`

Explanation: The array represents the integer 123.

Incrementing by one gives $123 + 1 = 124$.

Thus, the result should be `[1,2,4]`.

Example 2:

Input: `digits = [4,3,2,1]`

Output: `[4,3,2,2]`

Explanation: The array represents the integer 4321.

Incrementing by one gives $4321 + 1 = 4322$.

Thus, the result should be [4,3,2,2].

Example 3:

Input: digits = [9]

Output: [1,0]

Explanation: The array represents the integer 9.

Incrementing by one gives $9 + 1 = 10$.

Thus, the result should be [1,0].

C:

```
int* plusOne(int* digits, int digitsSize, int* returnSize){
    int i = 0;
    int *return_arr = NULL;
    for (i = digitsSize-1; i >= 0; i-- ) {
        if (digits[i] == 9) digits[i] = 0;
        else {
            digits[i] += 1;
            *returnSize = digitsSize;
            return digits;
        }
    }
    return_arr = (int*)malloc((digitsSize+1)*sizeof(int));
    return_arr[0] = 1;
    for (i = 0; i <= digitsSize-1; i++ ) {
        return_arr[i+1] = digits[i];
    }
    *returnSize = digitsSize+1;
    return return_arr;
}
```

JAVA:

```
class Solution {  
    public int[] plusOne(int[] digits) {  
        int n = digits.length;  
  
        // move along the input array starting from the end  
        for (int idx = n - 1; idx >= 0; --idx) {  
            // set all the nines at the end of array to zeros  
            if (digits[idx] == 9) {  
                digits[idx] = 0;  
            }  
            // here we have the rightmost not-nine  
            else {  
                // increase this rightmost not-nine by 1  
                digits[idx]++;  
                // and the job is done  
                return digits;  
            }  
        }  
        // we're here because all the digits are nines  
        digits = new int[n + 1];  
        digits[0] = 1;  
        return digits;  
    }  
}
```

1.8 Move Zeroes

Given an integer array `nums`, move all 0's to the end of it while maintaining the relative order of the non-zero elements.

Note that you must do this in-place without making a copy of the array.

Example 1:

Input: nums = [0,1,0,3,12]

Output: [1,3,12,0,0]

Example 2:

Input: nums = [0]

Output: [0]

C:

```
void moveZeroes(int* nums, int numsSize){
    int i = 0;
    int k = 0;
    int count = 0;
    for (i = 0 ; i < numsSize; i++) {
        if (nums[i] == 0) count++;
        else nums[k++] = nums[i];
    }
    for (i = k; i < numsSize; i++) nums[i] = 0;
}
```

JAVA:

```
class Solution {
    public void moveZeroes(int[] nums) {
        int count = 0;
        int k = 0;
        for (int i : nums) {
            if (i == 0) count++;
            else {
                nums[k] = i;
                k++;
            }
        }
        System.out.println(k);
        System.out.println(count);
        for (int j = k; j < k+count; j++) {
            nums[j] = 0;
        }
    }
}
```



```
}  
}
```

1.9 Two Sum

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]`

Explanation: Because `nums[0] + nums[1] == 9`, we return `[0, 1]`.

Example 2:

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

Output: `[1,2]`

Example 3:

Input: `nums = [3,3]`, `target = 6`

Output: `[0,1]`

C:

```
int* twoSum(int* nums, int numsSize, int target, int* returnSize){  
    for (int i = 0; i < numsSize; i++) {  
        for (int j = i + 1; j < numsSize; j++) {  
            if (nums[j] == target - nums[i]) {  
                nums[0] = i;  
                nums[1] = j;  
                *returnSize = 2;  
                return nums;  
            }  
        }  
    }  
}
```

```

    return NULL;
}

```

JAVA:

```

class Solution {
    public int[] twoSum(int[] nums, int target) {
        HashMap<Integer, Integer> hm = new HashMap<>();
        for (int i = 0 ; i < nums.length; i++) {
            if (hm.containsKey(target-nums[i])) return new int[]{hm.get(target-nums[i]), i};
            else hm.put(nums[i], i);
        }
        return null;
    }
}

```

1.10 Valid Sudoku

Determine if a 9 x 9 Sudoku board is valid. Only the filled cells need to be validated according to the following rules:

1. Each row must contain the digits 1-9 without repetition.
2. Each column must contain the digits 1-9 without repetition.
3. Each of the nine 3 x 3 sub-boxes of the grid must contain the digits 1-9 without repetition.

Note:

- A Sudoku board (partially filled) could be valid but is not necessarily solvable.
- Only the filled cells need to be validated according to the mentioned rules.

Example 1:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | | | 7 | | | | |
| 6 | | | 1 | 9 | 5 | | | |
| | 9 | 8 | | | | | 6 | |
| 8 | | | | 6 | | | | 3 |
| 4 | | | 8 | | 3 | | | 1 |
| 7 | | | | 2 | | | | 6 |
| | 6 | | | | | 2 | 8 | |
| | | | 4 | 1 | 9 | | | 5 |
| | | | | 8 | | | 7 | 9 |

Input: board =

```

[["5","3",".",".","7",".",".","."],
["6",".",".","1","9","5",".","."],
[["9","8",".",".",".","6","."],
["8",".",".","6",".",".","3"],
["4",".","8",".","3",".","1"],
["7",".","2",".","6"],
[["6",".","2","8","."],
[["4","1","9",".","5"],
[["8",".","7","9"]]

```

Output: true

Example 2:

Input: board =

```

[["8","3",".",".","7",".",".","."],
["6",".","1","9","5",".","."],
[["9","8",".",".","6","."],
["8",".","6",".","3"],
["4","8","3","1"],
["7","2","6"],
[["6","2","8","."],
[["4","1","9","5"],
[["8","7","9"]]

```

Output: false

Explanation: Same as Example 1, except with the 5 in the top left corner being modified to 8. Since there are two 8's in t

C:

```
bool isValidSudoku(char** board, int boardSize, int* boardColSize){
```

```

int rows[9][9];
int cols[9][9];
int box[9][9];
int r = 0;
int c = 0;

for (r = 0; r < 9; r++) {
    for (c = 0; c < 9; c++) {
        if (board[r][c] == '.') continue;

        int pos = board[r][c] - '1';

        if (rows[r][pos] == 1) return false;

        rows[r][pos] = 1;

        if (cols[c][pos] == 1) return false;

        cols[c][pos] = 1;

        if(box[r/3*3+c/3][pos] == 1) return false;

        box[r/3*3+c/3][pos] = 1;
    }
}
return true;
}

```

JAVA:

```

class Solution {
    public boolean isValidSudoku(char[][] board) {
        int N = 9;

        // Use a binary number to record previous occurrence
    }
}

```

```

int[] rows = new int[N];
int[] cols = new int[N];
int[] boxes = new int[N];

for (int r = 0; r < N; r++) {
    for (int c = 0; c < N; c++) {
        // Check if the position is filled with number
        if (board[r][c] == '.') {
            continue;
        }
        int val = board[r][c] - '1';
        int pos = 1 << (val);

        // Check the row
        if ((rows[r] & pos) > 0) {
            return false;
        }
        rows[r] |= pos;

        // Check the column
        if ((cols[c] & pos) > 0) {
            return false;
        }
        cols[c] |= pos;

        // Check the box
        int idx = (r / 3) * 3 + c / 3;
        if ((boxes[idx] & pos) > 0) {
            return false;
        }
        boxes[idx] |= pos;
    }
}

```

```
    return true;
}
}
```

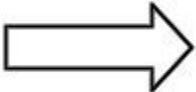
1.11 Rotate Image

You are given an $n \times n$ 2D matrix representing an image, rotate the image by 90 degrees (clockwise).

You have to rotate the image in-place, which means you have to modify the input 2D matrix directly. DO NOT allocate another 2D matrix and do the rotation.

Example 1:

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |



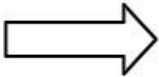
| | | |
|---|---|---|
| 7 | 4 | 1 |
| 8 | 5 | 2 |
| 9 | 6 | 3 |

Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]

Output: [[7,4,1],[8,5,2],[9,6,3]]

Example 2:

| | | | |
|----|----|----|----|
| 5 | 1 | 9 | 11 |
| 2 | 4 | 8 | 10 |
| 13 | 3 | 6 | 7 |
| 15 | 14 | 12 | 16 |



| | | | |
|----|----|----|----|
| 15 | 13 | 2 | 5 |
| 14 | 3 | 4 | 1 |
| 12 | 6 | 8 | 9 |
| 16 | 7 | 10 | 11 |

Input: matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]

Output: [[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]

C:

```
void transpose(int** matrix, int row, int col) {
    int i = 0;
    int j = 0;
    for (i = 0; i < row; i++) {
        for (j = i+1; j < col; j++) {
            int temp = matrix[i][j];
            matrix[i][j] = matrix[j][i];
            matrix[j][i] = temp;
        }
    }
}

void reverse(int** matrix, int row, int col) {
    int i = 0;
    int j = 0;
    for (i = 0; i < row; i++) {
        for (j = 0; j < col/2; j++) {
            int temp = matrix[i][j];
            matrix[i][j] = matrix[i][col-j-1];
            matrix[i][col-j-1] = temp;
        }
    }
}

void rotate(int** matrix, int matrixSize, int* matrixColSize){
    transpose(matrix, matrixSize, matrixSize);
    reverse(matrix, matrixSize, matrixSize);
}
```

JAVA:

```
class Solution {
    public void transpose(int[][] matrix) {
        int i = 0;
        int j = 0;
        int temp = 0;
        for (i = 0; i < matrix.length; i++) {
            for (j = i+1; j < matrix[0].length; j++) {
                temp = matrix[i][j];
                matrix[i][j] = matrix[j][i];
                matrix[j][i] = temp;
            }
        }
    }
}
```

```

public void reverse(int [][] matrix, int rows, int cols) {
    int i = 0;
    int temp = 0;
    int j = 0;
    for (i = 0; i < rows; i++) {
        for (j = 0; j < cols/2; j++) {
            temp = matrix[i][j];
            matrix[i][j] = matrix[i][cols-j-1];
            matrix[i][cols-j-1] = temp;
        }
    }
}

public void rotate(int[][] matrix) {
    transpose(matrix);
    reverse(matrix, matrix.length, matrix[0].length);
}
}

```


2. STRINGS

2.1 Reverse String

Write a function that reverses a string. The input string is given as an array of characters `s`.

You must do this by modifying the input array in-place with $O(1)$ extra memory.

Example 1:

Input: `s = ["h","e","l","l","o"]`

Output: `["o","l","l","e","h"]`

Example 2:

Input: `s = ["H","a","n","n","a","h"]`

Output: `["h","a","n","n","a","H"]`

C:

```
void reverseString(char* s, int sSize){
    for (int i = 0; i < sSize/2; i++) {
        char tmp = s[i];
        s[i] = s[sSize-i-1];
        s[sSize-i-1] = tmp;
    }
}
```

JAVA:

```
class Solution {
    public void reverseString(char[] s) {
        int i = 0;
        int j = s.length-1;
        while (i <= j) {
            char tmp = s[i];
            s[i] = s[j];
            s[j] = tmp;
            i++;
            j--;
        }
    }
}
```

2.2 Reverse Integer

Given a signed 32-bit integer x , return x with its digits reversed. If reversing x causes the value to go outside the signed 32-bit integer range $[-2^{31}, 2^{31} - 1]$, then return 0.

Assume the environment does not allow you to store 64-bit integers (signed or unsigned).

Example 1:

Input: $x = 123$

Output: 321

Example 2:

Input: $x = -123$

Output: -321

Example 3:

Input: $x = 120$

Output: 21

C:

```
int reverse(int x){
    int result = 0;

    while (x!=0) {
        int y = x%10;
        if (result > INT_MAX/10 || (result == INT_MAX / 10 && y > 7)) return 0;
        if (result < INT_MIN/10 || (result == INT_MIN / 10 && y < -8)) return 0;
        result = (result*10)+y;
        x= x/10;
    }

    return result;
}
```

JAVA:

```
class Solution {
    public int reverse(int x) {
        int rev = 0;
        while (x != 0) {
```

```

        int pop = x % 10;
        x /= 10;
        if (rev > Integer.MAX_VALUE/10 || (rev == Integer.MAX_VALUE / 10 && pop > 7)) return 0;
        if (rev < Integer.MIN_VALUE/10 || (rev == Integer.MIN_VALUE / 10 && pop < -8)) return 0;
        rev = rev * 10 + pop;
    }
    return rev;
}
}

```

2.3 First Unique Character In A String

Given a string **s**, *find the first non-repeating character in it and return its index*. If it does not exist, return **-1**.

Example 1:

Input: s = "leetcode"

Output: 0

Example 2:

Input: s = "loveleetcode"

Output: 2

Example 3:

Input: s = "aabb"

Output: -1

C:

```

int firstUniqChar(char * s){
    int count_map[26] = {0};
    int i = 0;
    for (i = 0; i < strlen(s); i++) {
        count_map[s[i]-'a']++;
    }
    for (i = 0; i < strlen(s); i++) {
        if (count_map[s[i]-'a'] == 1) return i;
    }
    return -1;
}

```

```
}
```

JAVA:

```
class Solution {
    public int firstUniqChar(String s) {
        int len = s.length();
        int count_map[] = new int[26];

        int i = 0;
        for (i = 0; i < len; i++) {
            count_map[s.charAt(i)-'a']++;
        }
        for (i = 0; i < len; i++) {
            if (count_map[s.charAt(i)-'a'] == 1) return i;
        }
        return -1;
    }
}
```

2.4 Valid Anagram

Given two strings **s** and **t**, return **true** if **t** is an anagram of **s**, and **false** otherwise.

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

Example 1:

Input: s = "anagram", t = "nagaram"

Output: true

Example 2:

Input: s = "rat", t = "car"

Output: false

C:

```
bool isAnagram(char * s, char * t){
    int count_map[26] = {0};
    int i = 0;
    if (strlen(s)!=strlen(t)) return false;
    for(i = 0; i < strlen(s); i++) {
        count_map[s[i]-'a']++;
    }
}
```

```

        count_map[t[i]-'a']--;
    }
    for(i = 0; i < 26; i++) {
        if (count_map[i] != 0) return false;
    }
    return true;
}

```

JAVA:

```

class Solution {
    public boolean isAnagram(String s, String t) {
        int i = 0;
        int count_map[] = new int[26];
        if (s.length() != t.length()) return false;
        for (i = 0; i < s.length(); i++) {
            count_map[s.charAt(i)-'a']++;
            count_map[t.charAt(i)-'a']--;
        }
        for (i = 0; i < 26; i++) {
            if (count_map[i] > 0 || count_map[i] < 0) return false;
        }
        return true;
    }
}

```

2.5 Valid Palindrome

A phrase is a **palindrome** if, after converting all uppercase letters into lowercase letters and removing all non-alphanumeric characters, it reads the same forward and backward. Alphanumeric characters include letters and numbers.

Given a string `s`, return `true` if it is a **palindrome**, or `false` otherwise.

Example 1:

Input: `s = "A man, a plan, a canal: Panama"`

Output: `true`

Explanation: "amanaplanacanalpanama" is a palindrome.

Example 2:

Input: `s = "race a car"`

Output: `false`

Explanation: "raceacar" is not a palindrome.

Example 3:

Input: s = " "

Output: true

Explanation: s is an empty string "" after removing non-alphanumeric characters. Since an empty string reads the same forward and backward, it is a palindrome.

C:

```
bool isPalindrome(char * s){
    int start = 0;
    int end = strlen(s);
    while (start < end) {
        if (!isalnum(s[start])) {
            start++;
            continue;
        }
        if (!isalnum(s[end])) {
            end--;
            continue;
        }
        if (tolower(s[start]) != tolower(s[end])) return false;
        else {
            start++;
            end--;
        }
    }
    return true;
}
```

JAVA:

```
class Solution {
    public boolean isPalindrome(String s) {
        int i = 0;
        int j = s.length()-1;
        while (i <= j) {
            if ((Character.toLowerCase(s.charAt(i))) == (Character.toLowerCase(s.charAt(j)))) {
                // System.out.println("Equal i is"+s.charAt(i)+"j is"+s.charAt(j));
                i++;
                j--;
            }
            else {

```

```

        if(!Character.isLetterOrDigit(s.charAt(i))) {
            // System.out.println("non alpha numeric in i"+s.charAt(i));
            i++;
        }
        else if (!Character.isLetterOrDigit(s.charAt(j))) {
            // System.out.println("non alpha numeric in j"+s.charAt(j));
            j--;
        }
        else {
            return false;
        }
    }
}
return true;
}
}

```

2.6 String To Integer (atoi)

Implement the `myAtoi(string s)` function, which converts a string to a 32-bit signed integer (similar to C/C++'s `atoi` function).

The algorithm for `myAtoi(string s)` is as follows:

1. Read in and ignore any leading whitespace.
2. Check if the next character (if not already at the end of the string) is '-' or '+'. Read this character in if it is either. This determines if the final result is negative or positive respectively. Assume the result is positive if neither is present.
3. Read in next the characters until the next non-digit character or the end of the input is reached. The rest of the string is ignored.
4. Convert these digits into an integer (i.e. "123" -> 123, "0032" -> 32). If no digits were read, then the integer is 0. Change the sign as necessary (from step 2).
5. If the integer is out of the 32-bit signed integer range $[-2^{31}, 2^{31} - 1]$, then clamp the integer so that it remains in the range. Specifically, integers less than -2^{31} should be clamped to -2^{31} , and integers greater than $2^{31} - 1$ should be clamped to $2^{31} - 1$.
6. Return the integer as the final result.

Note:

- Only the space character ' ' is considered a whitespace character.
- **Do not ignore** any characters other than the leading whitespace or the rest of the string after the digits.

Example 1:

Input: `s = "42"`

Output: 42

Explanation: The underlined characters are what is read in, the caret is the current reader position.

Step 1: "42" (no characters read because there is no leading whitespace)

^

Step 2: "42" (no characters read because there is neither a '-' nor '+')

^

Step 3: "42" ("42" is read in)

^

The parsed integer is 42.

Since 42 is in the range $[-2^{31}, 2^{31} - 1]$, the final result is 42.

Example 2:

Input: s = " -42"

Output: -42

Explanation:

Step 1: "___-42" (leading whitespace is read and ignored)

^

Step 2: " _-42" ('-' is read, so the result should be negative)

^

Step 3: " _-42" ("42" is read in)

^

The parsed integer is -42.

Since -42 is in the range $[-2^{31}, 2^{31} - 1]$, the final result is -42.

Example 3:

Input: s = "4193 with words"

Output: 4193

Explanation:

Step 1: "4193 with words" (no characters read because there is no leading whitespace)

^

Step 2: "4193 with words" (no characters read because there is neither a '-' nor '+')

^

Step 3: "4193 with words" ("4193" is read in; reading stops because the next character is a non-digit)

^

The parsed integer is 4193.

Since 4193 is in the range $[-2^{31}, 2^{31} - 1]$, the final result is 4193.

C:

```
int myAtoi(char * s){
    int i = 0;
    int j = strlen(s)-1;
    int result = 0;
    int sign = 1;
    if (isalpha(s[i])) {
        return 0;
    }
    while (s[i] == ' ') {
        i++;
    }
    if (s[i] == '+') {
        sign = 1;
        i++;
    } else if (s[i] == '-') {
        sign = -1;
        i++;
    } else {
        sign = 1;
    }
    while (i <= j) {
        if (isdigit(s[i])) {
            if ((result > INT_MAX/10) ||
                (result == INT_MAX/10) && (s[i]-'0') > INT_MAX%10) {
                return (sign == -1)? INT_MIN:INT_MAX;
            }
            result = 10*result + (s[i]-'0');
            i++;
        }
    }
}
```

```

    } else {
        i++;
        break;
    }
}
result = result*sign;
return result;
}

```

JAVA:

```

class Solution {
    public int myAtoi(String s) {
        int i = 0;
        int j = s.length()-1;
        int sign = 0;
        int result = 0;
        if (s.length() == 0) return 0;
        while (i < s.length()-1 && s.charAt(i) == ' ') {
            i++;
        }

        if (s.charAt(i) == '+') {
            sign = 1;
            i++;
        } else if (s.charAt(i) == '-') {
            sign = -1;
            i++;
        } else {
            sign = 1;
        }
        while (i <= j) {
            if (!Character.isDigit(s.charAt(i))) break;
            if ((result > Integer.MAX_VALUE/10) ||
                (result == Integer.MAX_VALUE/10 && s.charAt(i) > '7')) {
                if (sign == -1) return Integer.MIN_VALUE;
                else return Integer.MAX_VALUE;
            }
            result = result*10 + s.charAt(i)-'0';
            i++;
        }
        return sign*result;
    }
}

```

2.7 StrStr Implementation

Given two strings **needle** and **haystack**, return the index of the first occurrence of **needle** in **haystack**, or **-1** if **needle** is not part of **haystack**.

Example 1:

Input: haystack = "sadbutsad", needle = "sad"

Output: 0

Explanation: "sad" occurs at index 0 and 6.

The first occurrence is at index 0, so we return 0.

Example 2:

Input: haystack = "leetcode", needle = "leeto"

Output: -1

Explanation: "leeto" did not occur in "leetcode", so we return -1.

C:

```
int strStr(char * haystack, char * needle){
    int i = 0;
    int j = 0;
    int start = -1;
    while (i < strlen(haystack)) {
        if (haystack[i] == needle[j]) {
            if (start == -1) {
                start = i;
            }
            if (j == strlen(needle)-1) {
                return start;
            }
            i++;
            j++;
        }
        else {
            if (i == strlen(haystack)-1) {
                break;
            }
            i=start+1;
            start = i;
            j = 0;
        }
    }
    return -1;
}
```

```
}
```

JAVA:

```
class Solution {  
    public int strStr(String haystack, String needle) {  
        int m = needle.length();  
        int n = haystack.length();  
  
        for (int windowStart = 0; windowStart <= n; windowStart++) {  
            for (int i = 0; i < m; i++) {  
                if (needle.charAt(i) != haystack.charAt(windowStart + i)) {  
                    break;  
                }  
                if (i == m - 1) {  
                    return windowStart;  
                }  
            }  
        }  
  
        return -1;  
    }  
}
```

2.8 Longest Common Prefix

Write a function to find the longest common prefix string amongst an array of strings.

If there is no common prefix, return an empty string `""`.

Example 1:

Input: strs = ["flower","flow","flight"]

Output: "fl"

Example 2:

Input: strs = ["dog","racecar","car"]

Output: ""

Explanation: There is no common prefix among the input strings.

C:

```
char * longestCommonPrefix(char ** strs, int strsSize){  
    int i = 0;
```

```

int j = 0;
int min = strlen(strs[0]);
char *result = NULL;
for (i = 1; i < strsSize; i++) {
    if (strlen(strs[i]) < min) {
        min = strlen(strs[i]);
    }
}
result = (char*)malloc((min + 1) * sizeof(char));
memset(result, '\0', (min + 1) * sizeof(char));
for (i = 0; i < min; i++) {
    for (j = 0; j < strsSize; j++) {
        if (result[i] == 0) {
            result[i] = strs[j][i];
        } else {
            if (result[i] != strs[j][i]) {
                result[i] = '\0';
                return result;
            }
        }
    }
}

return result;
}

```

JAVA:

```

class Solution {

```

```

    public String longestCommonPrefix(String[] strs) {
        // Longest common prefix string
        StringBuilder longestCommonPrefix = new StringBuilder();
        // Base condition
        if (strs == null || strs.length == 0) {
            return longestCommonPrefix.toString();
        }
        // Find the minimum length string from the array
        int minLength = strs[0].length();
        for (int i = 1; i < strs.length; i++) {
            minLength = Math.min(minLength, strs[i].length());
        }
        // Loop for the minimum length
        for (int i = 0; i < minLength; i++) {
            // Get the current character from first string
            char current = strs[0].charAt(i);
            // Check if this character is found in all other strings or not

```

```
    for (String str : strs) {  
        if (str.charAt(i) != current) {  
            return longestCommonPrefix.toString();  
        }  
    }  
    longestCommonPrefix.append(current);  
}  
return longestCommonPrefix.toString();  
}  
}
```

3. LINKED LIST

3.1 Delete Node in Linked List

There is a singly-linked list **head** and we want to delete a node **node** in it.

You are given the node to be deleted **node**. You will **not be given access** to the first node of **head**.

All the values of the linked list are **unique**, and it is guaranteed that the given node **node** is not the last node in the linked list.

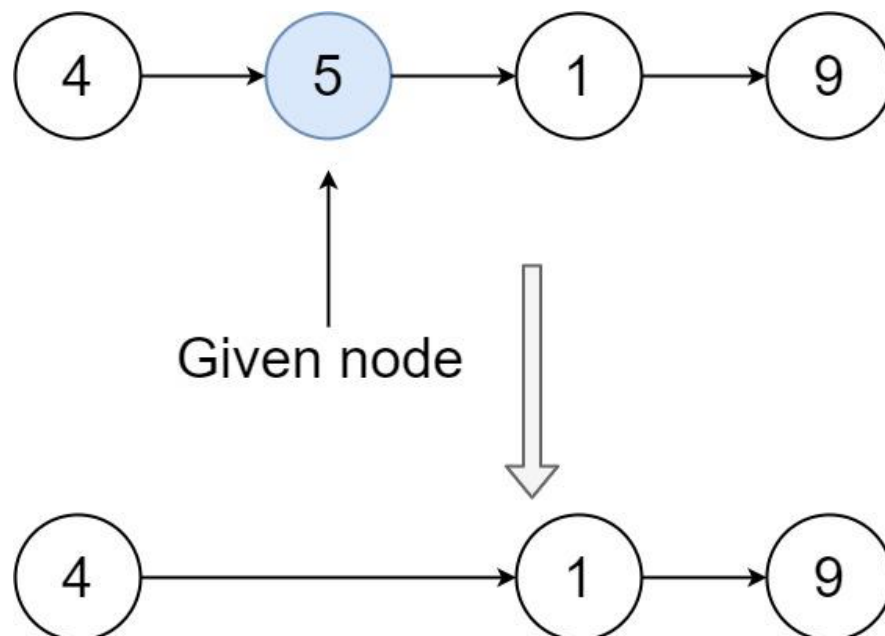
Delete the given node. Note that by deleting the node, we do not mean removing it from memory. We mean:

- The value of the given node should not exist in the linked list.
- The number of nodes in the linked list should decrease by one.
- All the values before **node** should be in the same order.
- All the values after **node** should be in the same order.

Custom testing:

- For the input, you should provide the entire linked list **head** and the node to be given **node**. **node** should not be the last node of the list and should be an actual node in the list.
- We will build the linked list and pass the node to your function.
- The output will be the entire list after calling your function.

Example 1:

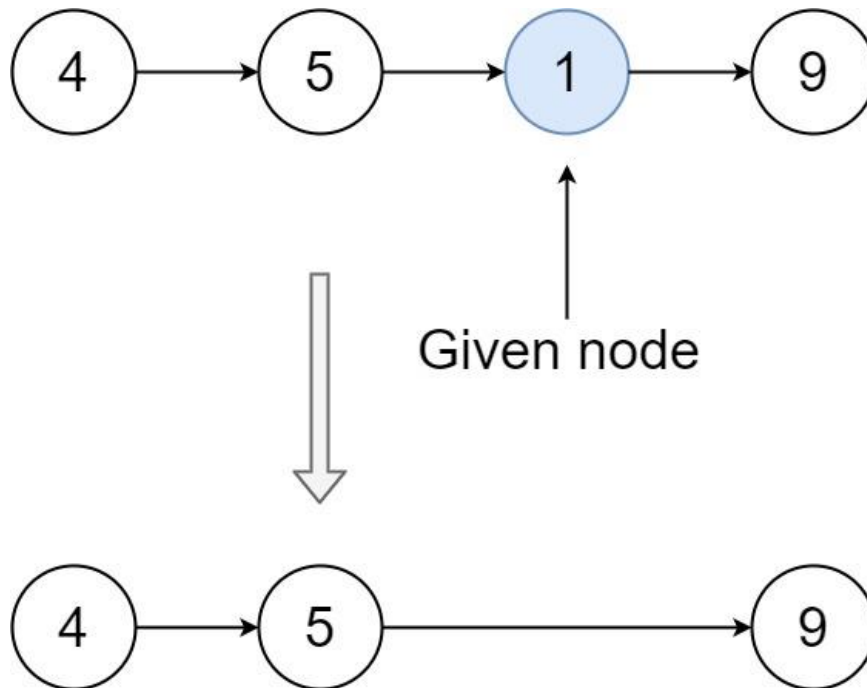


Input: head = [4,5,1,9], node = 5

Output: [4,1,9]

Explanation: You are given the second node with value 5, the linked list should become 4 -> 1 -> 9 after calling your function.

Example 2:



Input: head = [4,5,1,9], node = 1

Output: [4,5,9]

Explanation: You are given the third node with value 1, the linked list should become 4 -> 5 -> 9 after calling your function

C:

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
void deleteNode(struct ListNode* node) {
    struct ListNode* next_node = node->next;
    node->val = next_node->val;
    node->next = next_node->next;
    free(next_node);
}
```

JAVA:


```

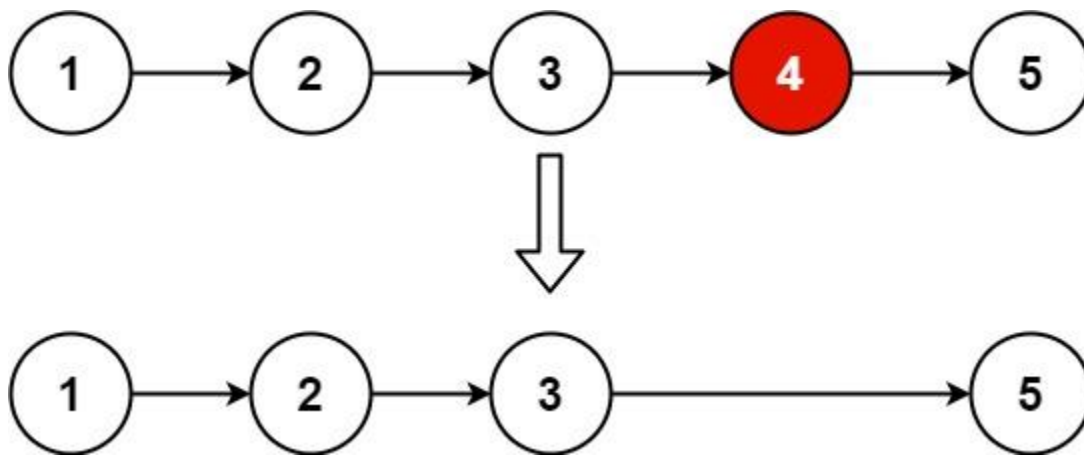
class Solution {
    public void deleteNode(ListNode node) {
        ListNode next = node.next;
        node.val = next.val;
        node.next = next.next;
    }
}

```

3.2 Remove Nth Node From End Of List

Given the **head** of a linked list, remove the n^{th} node from the end of the list and return its head.

Example 1:



Input: head = [1,2,3,4,5], n = 2

Output: [1,2,3,5]

Example 2:

Input: head = [1], n = 1

Output: []

Example 3:

Input: head = [1,2], n = 1

Output: [1]

C:

```
struct ListNode* removeNthFromEnd(struct ListNode* head, int n){
    struct ListNode *fast = head, *slow = head;
    for (int i = 0; i < n; i++) fast = fast->next;
    if (fast == NULL) return head->next;
    while (fast->next) fast = fast->next, slow = slow->next;
    slow->next = slow->next->next;
    return head;
}
```

JAVA:

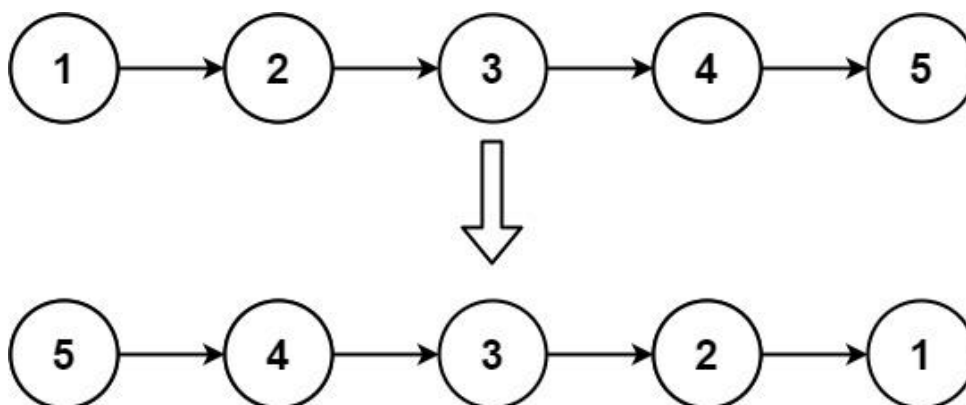
```
class Solution {
    ListNode removeNthFromEnd(ListNode head, int n){
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode slow = dummy;
        ListNode fast = dummy;

        for (int i = 0; i <= n; i++) {
            fast = fast.next;
        }
        while (fast != null) {
            fast = fast.next;
            slow = slow.next;
        }
        slow.next = slow.next.next;
        return dummy.next;
    }
}
```

3.3 Reverse Linked List

Given the **head** of a singly linked list, reverse the list, and return *the reversed list*.

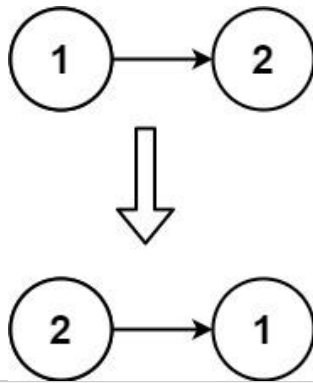
Example 1:



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

Output: [5,4,3,2,1]

Example 2:



Input: head = [1,2]

Output: [2,1]

Example 3:

Input: head = []

Output: []

C:

```
struct ListNode* reverseList(struct ListNode* head){
    struct ListNode* prev = NULL;
    struct ListNode* curr = head;
    while (curr) {
        struct ListNode *temp = curr->next;
        curr->next = prev;
        prev = curr;
        curr = temp;
    }
    return prev;
}
```

JAVA:

```
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode curr = head;
        ListNode prev = null;
        ListNode next = null;
        while (curr != null) {
            next = curr.next;
```

```

    curr.next = prev;
    prev = curr;
    curr = next;
  }
  return prev;
}

```

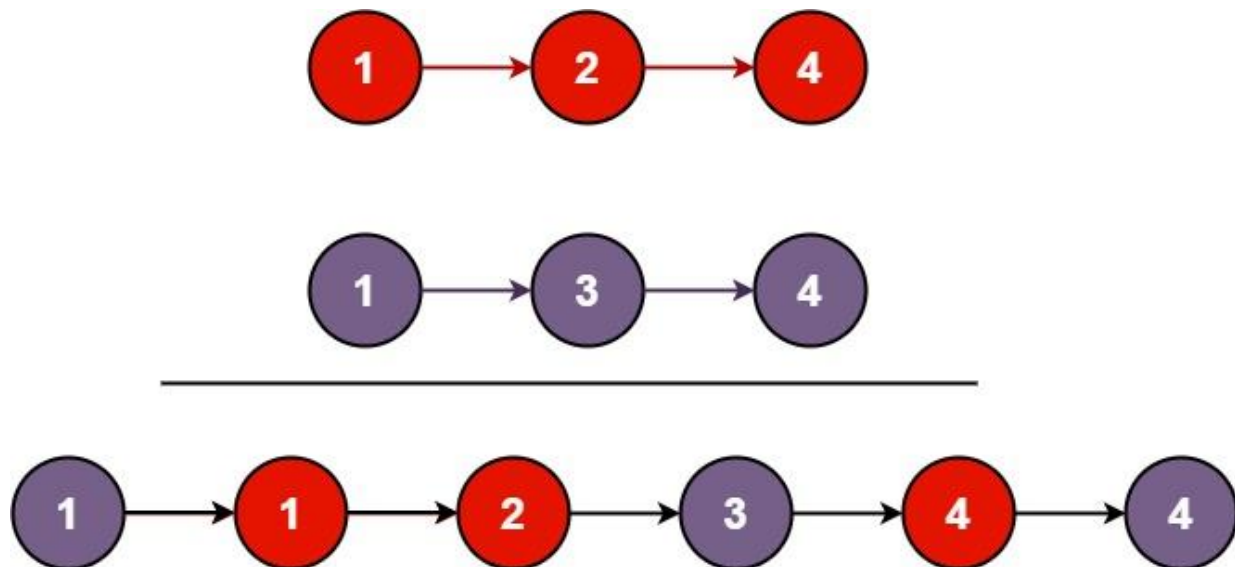
3.4 Merge Two Sorted Lists

You are given the heads of two sorted linked lists `list1` and `list2`.

Merge the two lists into one **sorted** list. The list should be made by splicing together the nodes of the first two lists.

Return *the head of the merged linked list*.

Example 1:



Input: list1 = [1,2,4], list2 = [1,3,4]

Output: [1,1,2,3,4,4]

Example 2:

Input: list1 = [], list2 = []

Output: []

Example 3:

Input: list1 = [], list2 = [0]

Output: [0]

C:

```
struct ListNode* mergeTwoLists(struct ListNode* list1, struct ListNode* list2){
    struct ListNode *dummy = (struct ListNode *)malloc(sizeof(struct ListNode));
    struct ListNode *curr = dummy;

    if (list1 == NULL) return list2;
    if (list2 == NULL) return list1;
    if (list1 == NULL && list2 == NULL) return NULL;
    while (list1!=NULL && list2!=NULL) {
        if (list1->val < list2->val) {
            dummy->next = list1;
            dummy = dummy->next;
            list1 = list1->next;
        } else {
            dummy->next = list2;
            dummy = dummy->next;
            list2 = list2->next;
        }
    }
    while (list1 != NULL) {
        dummy->next = list1;
        dummy = dummy->next;
        list1 = list1->next;
    }
    while (list2 != NULL) {
        dummy->next = list2;
        dummy = dummy->next;
        list2 = list2->next;
    }
    return curr->next;
}
```

JAVA:

```
class Solution {
    public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
        ListNode curr = new ListNode(0);
        ListNode dummy = curr;
        if (list1 == null) return list2;
        if (list2 == null) return list1;
        if (list1 == null && list2 == null) return null;
        while (list1!=null && list2!=null) {
```

```

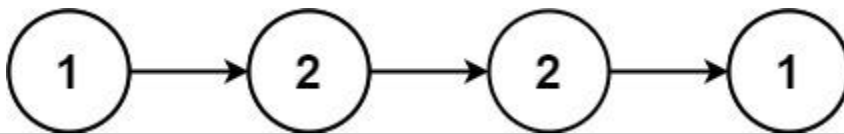
    if (list1.val < list2.val) {
        curr.next=list1;
        curr=curr.next;
        list1=list1.next;
    }
    else {
        curr.next=list2;
        curr=curr.next;
        list2=list2.next;
    }
}
while (list1!=null) {
    curr.next=list1;
    curr=curr.next;
    list1=list1.next;
}
while (list2!=null) {
    curr.next=list2;
    curr=curr.next;
    list2=list2.next;
}
return dummy.next;
}
}

```

3.5 Palindrome Linked List

Given the **head** of a singly linked list, return **true** if it is a *palindrome* or **false** otherwise.

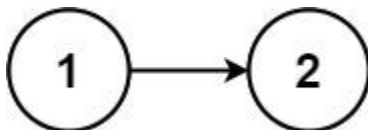
Example 1:



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

Output: true

Example 2:



Input: head = [1,2]

Output: false

C:

```
struct ListNode *endoffirsthalf(struct ListNode *head) {
    struct ListNode *slow = head;
    struct ListNode *fast = head;
    while (fast->next != NULL && fast->next->next != NULL) {
        fast = fast->next->next;
        slow = slow->next;
    }
    return slow;
}
```

```
struct ListNode *reverselist(struct ListNode *head) {
    struct ListNode *curr = head;
    struct ListNode *prev = NULL;
    struct ListNode *next = NULL;
    while (curr != NULL) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}
```

```
bool isPalindrome(struct ListNode* head){
```

```
    if (head == NULL) return true;
```

```
    struct ListNode *firsthalfend = endoffirsthalf(head);
    struct ListNode *secondhalfstart = reverselist(firsthalfend->next);
    struct ListNode* one = head;
    struct ListNode* two = secondhalfstart;
    while (two != NULL) {
        if (two->val == one->val) {
            two = two->next;
            one = one->next;
        } else {
            return false;
        }
    }
}
```

```
    firsthalfend->next = reverselist(secondhalfstart);
    return true;
}
```

JAVA:

```
class Solution {
```

```

public ListNode reverselist (ListNode head) {
    ListNode curr = head;
    ListNode prev = null;
    ListNode next = null;
    while (curr != null) {
        next = curr.next;
        curr.next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}

public ListNode endfirst (ListNode head) {
    ListNode slow = head;
    ListNode fast = head;
    while (fast.next != null && fast.next.next != null) {
        fast = fast.next.next;
        slow = slow.next;
    }
    return slow;
}

public boolean isPalindrome(ListNode head) {
    ListNode endoffirsthalf = endfirst(head);
    ListNode secondhalfstart = reverselist(endoffirsthalf.next);
    ListNode one = head;
    ListNode two = secondhalfstart;

    while (two != null) {
        if (one.val != two.val) return false;
        one = one.next;
        two = two.next;
    }

    endoffirsthalf.next = reverselist(secondhalfstart);
    return true;
}
}

```

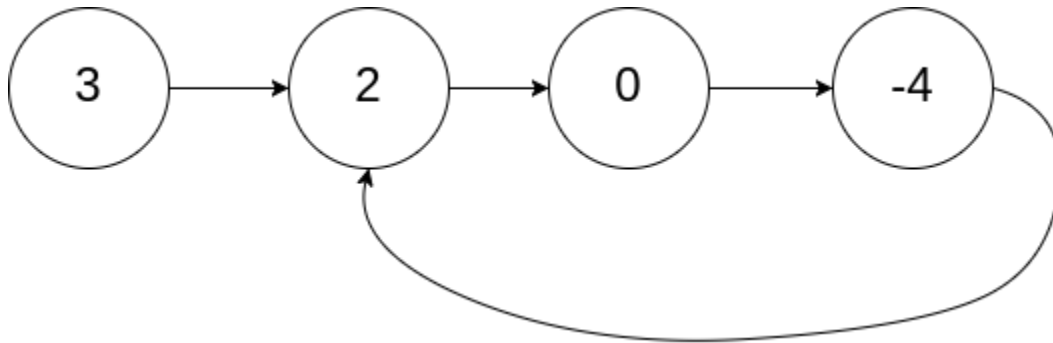
3.6 Linked List Cycle

Given **head**, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the **next** pointer. Internally, **pos** is used to denote the index of the node that tail's **next** pointer is connected to. **Note that pos is not passed as a parameter.**

Return **true** if there is a cycle in the linked list. Otherwise, return **false**.

Example 1:

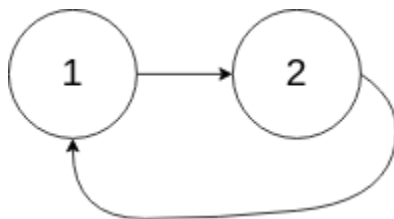


Input: head = [3,2,0,-4], pos = 1

Output: true

Explanation: There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).

Example 2:



Input: head = [1,2], pos = 0

Output: true

Explanation: There is a cycle in the linked list, where the tail connects to the 0th node.

Example 3:



Input: head = [1], pos = -1

Output: false

Explanation: There is no cycle in the linked list.

C:

```
bool hasCycle(struct ListNode *head) {  
    if( head == NULL) return false;  
    struct ListNode *slow = head;  
    struct ListNode *fast = head;  
  
    while (fast->next != NULL && fast->next->next != NULL) {  
        slow = slow->next;  
        fast = fast->next->next;  
        if (slow == fast) return true;  
    }  
    return false;  
}
```

JAVA:

```
public class Solution {  
    public boolean hasCycle(ListNode head) {  
        if( head == null) return false;  
        ListNode slow = head;  
        ListNode fast = head;  
  
        while (fast.next != null && fast.next.next != null) {  
            slow = slow.next;  
            fast = fast.next.next;  
            if (slow == fast) return true;  
        }  
        return false;  
    }  
}
```

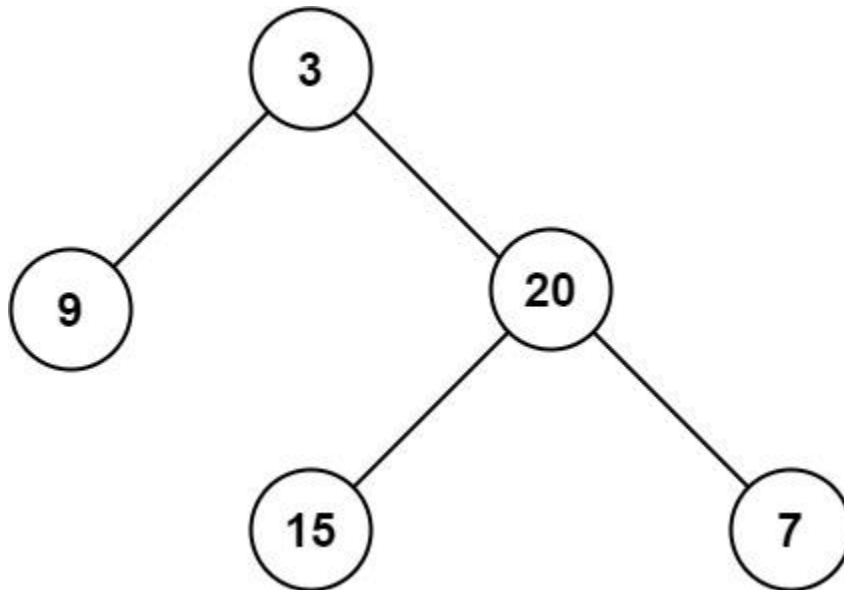
4. TREES

4.1 Maximum Depth Of Binary Tree

Given the **root** of a binary tree, return *its maximum depth*.

A binary tree's **maximum depth** is the number of nodes along the longest path from the root node down to the farthest leaf node.

Example 1:



Input: root = [3,9,20,null,null,15,7]

Output: 3

Example 2:

Input: root = [1,null,2]

Output: 2

C:

```
int max(int a, int b) {  
    return (a > b) ? a : b;  
}  
int maxDepth(struct TreeNode* root){  
    if (root == NULL) {  
        return 0;  
    }  
}
```

```

    return max(1 + maxDepth(root -> left), 1 + maxDepth(root -> right));
}

```

JAVA:

```

class Solution {
    public int maxDepth(TreeNode root) {
        LinkedList<TreeNode> node = new LinkedList<TreeNode>();
        LinkedList<Integer> node_depth = new LinkedList<Integer>();
        int depth = 0;

        if (root == null) return 0;

        node.add(root);
        node_depth.add(1);

        while (!node.isEmpty()) {
            TreeNode current_node = node.pollLast();
            int current_node_depth = node_depth.pollLast();
            if (current_node != null) {
                depth = Math.max(depth, current_node_depth);
                if (current_node.left != null) {
                    node.add(current_node.left);
                    node_depth.add(current_node_depth+1);
                }
                if (current_node.right != null) {
                    node.add(current_node.right);
                    node_depth.add(current_node_depth+1);
                }
            }
        }

        return depth;
    }
}

```

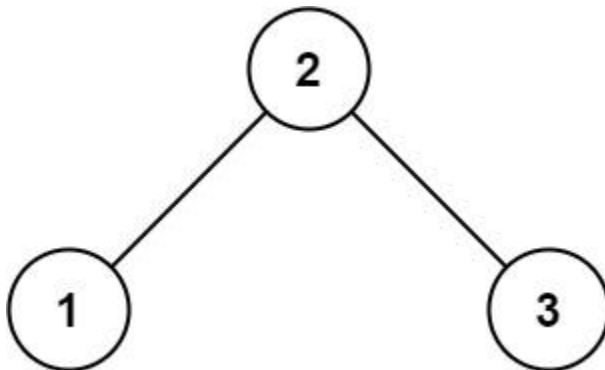
4.2 Validate Binary Search Tree

Given the **root** of a binary tree, *determine if it is a valid binary search tree (BST)*.

A **valid BST** is defined as follows:

- The left subtree of a node contains only nodes with keys **less than** the node's key.
- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

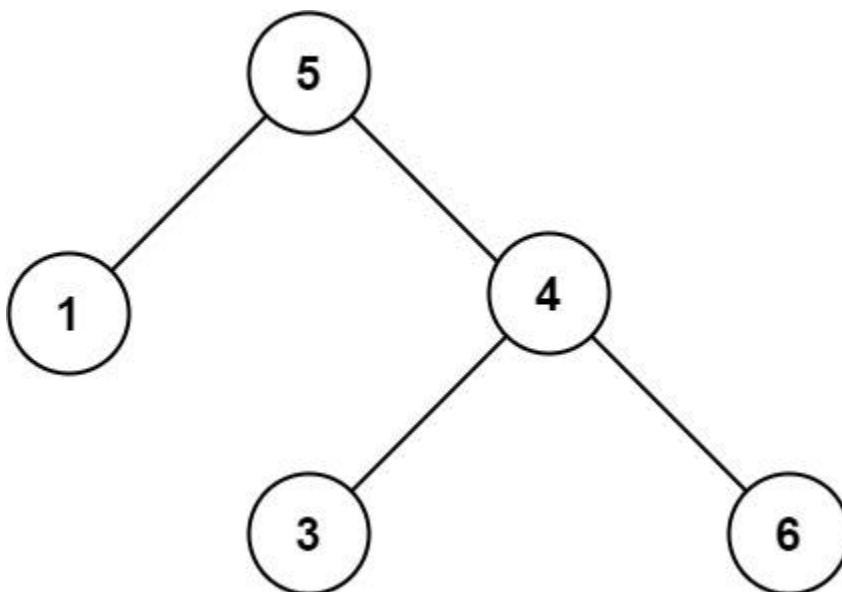
Example 1:



Input: root = [2,1,3]

Output: true

Example 2:



Input: root = [5,1,4,null,null,3,6]

Output: false

Explanation: The root node's value is 5 but its right child's value is 4.

C:

```
bool isValidBSTHelper(struct TreeNode* node, int *min, int *max) {  
    if (node == NULL) return true;  
    if ((min != NULL && node->val <= *min) || (max != NULL && node->val >= *max)) return false;
```

```

    return isValidBSTHelper(node->left,min,&node->val) && isValidBSTHelper(node->right, &node->val,
max);
}
bool isValidBST(struct TreeNode* root){
    return isValidBSTHelper(root,NULL,NULL);
}

```

JAVA:

```

class Solution {
    public boolean isValidBST(TreeNode root) {
        return isValidBSTHelper(root, null, null);
    }

    private boolean isValidBSTHelper(TreeNode node, Integer min, Integer max) {
        if (node == null) {
            return true;
        }

        if ((min != null && node.val <= min) || (max != null && node.val >= max)) {
            return false;
        }

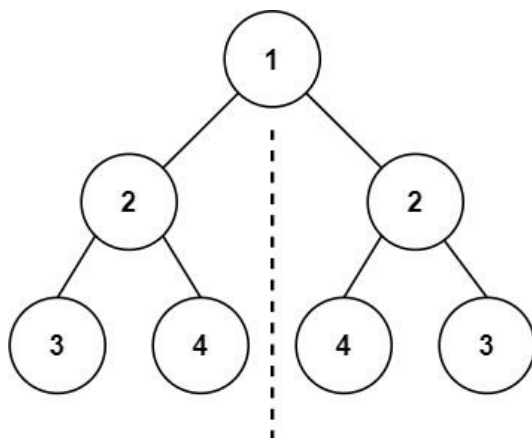
        return isValidBSTHelper(node.left, min, node.val) && isValidBSTHelper(node.right, node.val, max);
    }
}

```

4.3 Symmetric Tree

Given the **root** of a binary tree, *check whether it is a mirror of itself* (i.e., symmetric around its center).

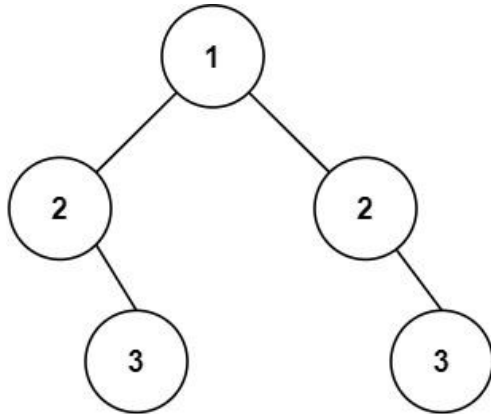
Example 1:



Input: root = [1,2,2,3,4,4,3]

Output: true

Example 2:



Input: root = [1,2,2,null,3,null,3]

Output: false

Constraints:

- The number of nodes in the tree is in the range [1, 1000].
- $-100 \leq \text{Node.val} \leq 100$

C:

```
bool isSymmetric(struct TreeNode* root){
    if (root == NULL) {
        return true;
    }

    struct TreeNode* stack[200];
    int top = -1;
    stack[++top] = root;
    stack[++top] = root;

    while (top >= 0) {
        struct TreeNode* left = stack[top--];
        struct TreeNode* right = stack[top--];

        if (left == NULL && right == NULL) {
            continue;
        }
    }
}
```

```

    if (left == NULL || right == NULL) {
        return false;
    }

    if (left->val != right->val) {
        return false;
    }

    stack[++top] = left->left;
    stack[++top] = right->right;
    stack[++top] = left->right;
    stack[++top] = right->left;
}

return true;
}

```

JAVA:

```

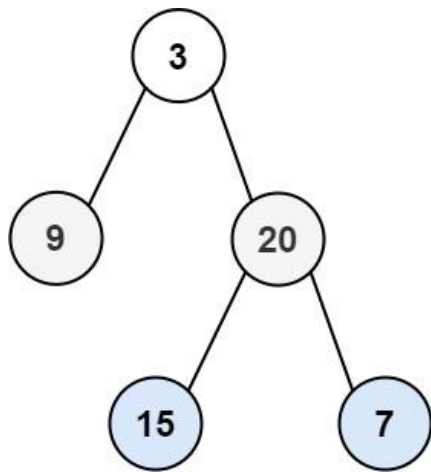
class Solution {
    public boolean isSymmetric(TreeNode root) {
        Stack<TreeNode> st = new Stack<TreeNode>();
        st.push(root);
        st.push(root);
        while (!st.isEmpty()) {
            TreeNode left = st.pop();
            TreeNode right = st.pop();
            if (left == null && right == null) continue;
            if (left == null || right == null) return false;
            if (left.val != right.val) return false;
            st.push(left.left);
            st.push(right.right);
            st.push(left.right);
            st.push(right.left);
        }
        return true;
    }
}

```

4.4 **Binary Tree Level Order Traversal**

Given the **root** of a binary tree, return *the level order traversal of its nodes' values*. (i.e., from left to right, level by level).

Example 1:



Input: root = [3,9,20,null,null,15,7]

Output: [[3],[9,20],[15,7]]

Example 2:

Input: root = [1]

Output: [[1]]

Example 3:

Input: root = []

Output: []

C:

```
#include <stdlib.h>
```

```
struct TreeNode {
    int val;
    struct TreeNode* left;
    struct TreeNode* right;
};
```

```
struct QueueNode {
    struct TreeNode* node;
    struct QueueNode* next;
};
```

```
struct Queue {
    struct QueueNode* front;
    struct QueueNode* rear;
```

```

};

struct Queue* createQueue() {
    struct Queue* queue = (struct Queue*)malloc(sizeof(struct Queue));
    queue->front = queue->rear = NULL;
    return queue;
}

void enqueue(struct Queue* queue, struct TreeNode* node) {
    struct QueueNode* newQueueNode = (struct QueueNode*)malloc(sizeof(struct QueueNode));
    newQueueNode->node = node;
    newQueueNode->next = NULL;

    if (queue->rear == NULL) {
        queue->front = queue->rear = newQueueNode;
        return;
    }

    queue->rear->next = newQueueNode;
    queue->rear = newQueueNode;
}

struct TreeNode* dequeue(struct Queue* queue) {
    if (queue->front == NULL)
        return NULL;

    struct TreeNode* node = queue->front->node;
    struct QueueNode* temp = queue->front;
    queue->front = queue->front->next;

    if (queue->front == NULL)
        queue->rear = NULL;

    free(temp);
    return node;
}

int** levelOrder(struct TreeNode* root, int* returnSize, int** returnColumnSizes) {
    if (root == NULL) {
        *returnSize = 0;
        *returnColumnSizes = NULL;
        return NULL;
    }

    struct Queue* queue = createQueue();
    int** result = (int**)malloc(sizeof(int*) * 1000);
    *returnColumnSizes = (int*)malloc(sizeof(int) * 1000);

```

```

int level = 0, levelSize = 1, nextLevelSize = 0;
struct TreeNode* currentNode;

enqueue(queue, root);

while (queue->front != NULL) {
    result[level] = (int*)malloc(sizeof(int) * levelSize);
    (*returnColumnSizes)[level] = levelSize;

    for (int i = 0; i < levelSize; i++) {
        currentNode = dequeue(queue);
        result[level][i] = currentNode->val;

        if (currentNode->left != NULL) {
            enqueue(queue, currentNode->left);
            nextLevelSize++;
        }
        if (currentNode->right != NULL) {
            enqueue(queue, currentNode->right);
            nextLevelSize++;
        }
    }

    levelSize = nextLevelSize;
    nextLevelSize = 0;
    level++;
}

*returnSize = level;
free(queue);
return result;
}

```

JAVA:

```

class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        Queue<TreeNode> q = new LinkedList<TreeNode>();
        int level = 0;
        if (root == null) return result;
        q.add(root);
        while (!q.isEmpty()) {
            int level_size = q.size();
            result.add(new ArrayList<Integer>());
            for (int i = 0; i < level_size; i++) {
                TreeNode node = q.remove();
            }
        }
    }
}

```

```

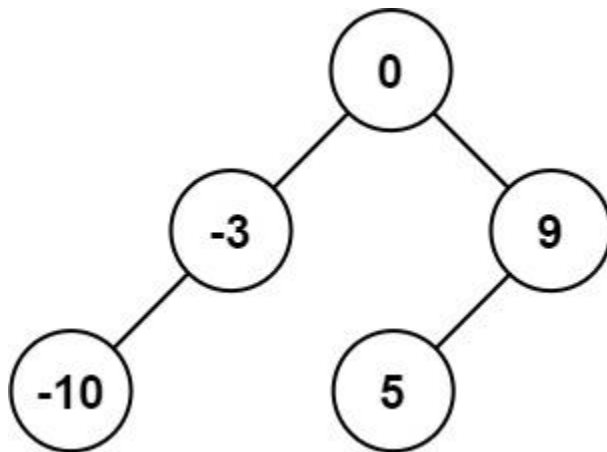
        result.get(level).add(node.val);
        if (node.left != null) q.add(node.left);
        if (node.right != null) q.add(node.right);
    }
    level++;
}
return result;
}
}

```

4.5 Sorted Array To Binary Search Tree

Given an integer array **nums** where the elements are sorted in **ascending order**, convert it to a **height-balanced** binary search tree.

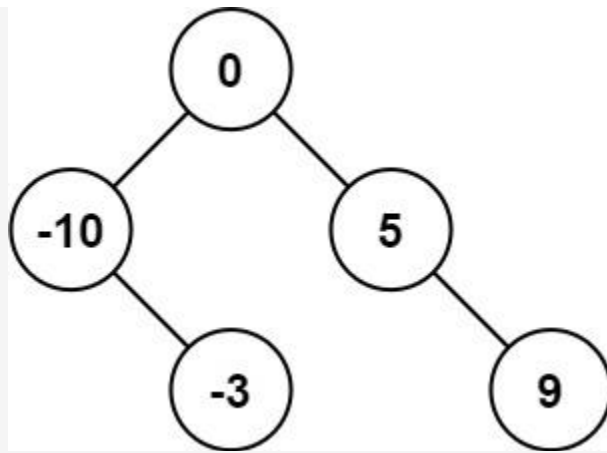
Example 1:



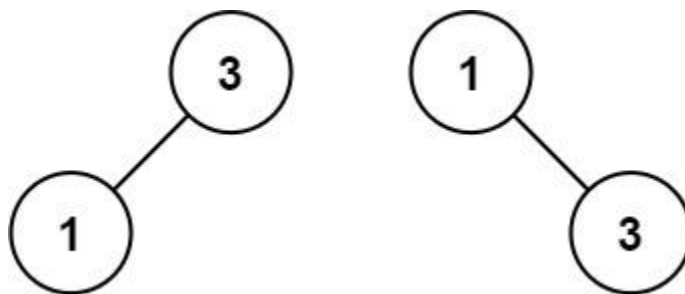
Input: nums = [-10,-3,0,5,9]

Output: [0,-3,9,-10,null,5]

Explanation: [0,-10,5,null,-3,null,9] is also accepted:



Example 2:



Input: nums = [1,3]

Output: [3,1]

Explanation: [1,null,3] and [3,1] are both height-balanced BSTs.

C:

```

struct TreeNode* helper(int* nums, int left, int right) {
    if (left > right)
        return NULL;

    int middle = (left + right) / 2;
    struct TreeNode* result = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    result->val = nums[middle];
    result->left = helper(nums, left, middle - 1);
    result->right = helper(nums, middle + 1, right);

    return result;
}

struct TreeNode* sortedArrayToBST(int* nums, int numsSize) {
    return helper(nums, 0, numsSize - 1);
}
  
```

JAVA:

```
class Solution {  
    int num[];  
    public TreeNode helper(int left, int right) {  
        if (left > right) return null;  
        int middle = (left+right)/2;  
        TreeNode result = new TreeNode(num[middle]);  
        result.left = helper(left,middle-1);  
        result.right = helper(middle+1,right);  
        return result;  
    }  
    public TreeNode sortedArrayToBST(int[] nums) {  
        this.num = nums;  
        return helper(0,nums.length-1);  
    }  
}
```

5. SEARCHING AND SORTING

5.1 Merge Two Sorted Arrays

You are given two integer arrays `nums1` and `nums2`, sorted in **non-decreasing order**, and two integers `m` and `n`, representing the number of elements in `nums1` and `nums2` respectively.

Merge `nums1` and `nums2` into a single array sorted in **non-decreasing order**.

The final sorted array should not be returned by the function, but instead be *stored inside the array* `nums1`. To accommodate this, `nums1` has a length of `m + n`, where the first `m` elements denote the elements that should be merged, and the last `n` elements are set to `0` and should be ignored. `nums2` has a length of `n`.

Example 1:

Input: `nums1 = [1,2,3,0,0,0]`, `m = 3`, `nums2 = [2,5,6]`, `n = 3`

Output: `[1,2,2,3,5,6]`

Explanation: The arrays we are merging are `[1,2,3]` and `[2,5,6]`.

The result of the merge is `[1,2,2,3,5,6]` with the underlined elements coming from `nums1`.

Example 2:

Input: `nums1 = [1]`, `m = 1`, `nums2 = []`, `n = 0`

Output: `[1]`

Explanation: The arrays we are merging are `[1]` and `[]`.

The result of the merge is `[1]`.

Example 3:

Input: `nums1 = [0]`, `m = 0`, `nums2 = [1]`, `n = 1`

Output: `[1]`

Explanation: The arrays we are merging are `[]` and `[1]`.

The result of the merge is `[1]`.

Note that because `m = 0`, there are no elements in `nums1`. The `0` is only there to ensure the merge result can fit in `num`

C:

```
void merge(int* nums1, int nums1Size, int m, int* nums2, int nums2Size, int n){
    int i = m-1;
    int j = n-1;
    int temp = 0;
    int k = m+n-1;
    while (i >=0 && j >=0 ) {
        if (nums1[i] > nums2[j]) {
            nums1[k--] = nums1[i--];
        } else if (nums1[i] < nums2[j]) {
            nums1[k--] = nums2[j--];
        } else {
            nums1[k--] = nums2[j--];
        }
    }
    while (i >= 0) nums1[k--] = nums1[i--];
    while (j >= 0) nums1[k--] = nums2[j--];
}
```

JAVA:

```
class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        int i = m-1;
        int j = n-1;
        int temp = 0;
        int k = m+n-1;
        while (i >=0 && j >=0 ) {
            if (nums1[i] > nums2[j]) {
                nums1[k--] = nums1[i--];
            } else if (nums1[i] < nums2[j]) {
                nums1[k--] = nums2[j--];
            } else {
                nums1[k--] = nums2[j--];
            }
        }
        while (i >= 0) nums1[k--] = nums1[i--];
        while (j >= 0) nums1[k--] = nums2[j--];
    }
}
```

5.2 First Bad Version

You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.

Suppose you have n versions $[1, 2, \dots, n]$ and you want to find out the first bad one, which causes all the following ones to be bad.

You are given an API `bool isBadVersion(version)` which returns whether `version` is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

Example 1:

Input: $n = 5$, $bad = 4$

Output: 4

Explanation:

call `isBadVersion(3)` -> false

call `isBadVersion(5)` -> true

call `isBadVersion(4)` -> true

Then 4 is the first bad version.

Example 2:

Input: $n = 1$, $bad = 1$

Output: 1

C:

```
int firstBadVersion(int n) {
    int left = 0;
    int right = n-1;
    int middle = 0;
    while (left <= right) {
        middle = left + (right-left)/2;
        if (isBadVersion(middle)) right = middle-1;
        else left = middle+1;
    }
    return left;
}
```

JAVA:

```
public class Solution extends VersionControl {
    public int firstBadVersion(int n) {
        int mid = 0;
        int left = 0;
```

```
int right = n-1;

while (left <= right) {
    mid = left+(right-left)/2;
    if (isBadVersion(mid)) right = mid-1;
    else {
        left = mid+1;
    }
}
return left;
}
```

6. DESIGN

6.1 Shuffle An Array

Given an integer array `nums`, design an algorithm to randomly shuffle the array. All permutations of the array should be **equally likely** as a result of the shuffling.

Implement the `Solution` class:

- `Solution(int[] nums)` Initializes the object with the integer array `nums`.
- `int[] reset()` Resets the array to its original configuration and returns it.
- `int[] shuffle()` Returns a random shuffling of the array.

Example 1:

Input

```
["Solution", "shuffle", "reset", "shuffle"]
```

```
[[[1, 2, 3]], [], [], []]
```

Output

```
[null, [3, 1, 2], [1, 2, 3], [1, 3, 2]]
```

Explanation

```
Solution solution = new Solution([1, 2, 3]);
```

```
solution.shuffle();    // Shuffle the array [1,2,3] and return its result.
```

```
                        // Any permutation of [1,2,3] must be equally likely to be re  
turned.
```

```
                        // Example: return [3, 1, 2]
```

```
solution.reset();      // Resets the array back to its original configuration [1,2,3  
]. Return [1, 2, 3]
```

```
solution.shuffle();    // Returns the random shuffling of array [1,2,3]. Example: re  
turn [1, 3, 2]
```

C:

```
#include <stdlib.h>
```

```
#include <string.h>
```

```

typedef struct {
    int* original;
    int* copy;
    int* temp;
    int length;
} Solution;

void swap(int* arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

Solution* solutionCreate(int* nums, int numsSize) {
    Solution* obj = (Solution*)malloc(sizeof(Solution));
    obj->original = (int*)malloc(sizeof(int) * numsSize);
    obj->copy = (int*)malloc(sizeof(int) * numsSize);
    obj->temp = (int*)malloc(sizeof(int) * numsSize);
    obj->length = numsSize;

    memcpy(obj->original, nums, sizeof(int) * numsSize);
    memcpy(obj->copy, nums, sizeof(int) * numsSize);

    return obj;
}

int* solutionReset(Solution* obj, int* returnSize) {
    memcpy(obj->copy, obj->original, sizeof(int) * obj->length);
    *returnSize = obj->length;
    return obj->original;
}

int* solutionShuffle(Solution* obj, int* returnSize) {
    memcpy(obj->temp, obj->copy, sizeof(int) * obj->length);

    for (int i = 0; i < obj->length; i++) {
        int ra = obj->length - i;
        int j = i + rand() % ra;
        swap(obj->temp, i, j);
    }

    *returnSize = obj->length;
    return obj->temp;
}

void solutionFree(Solution* obj) {
    free(obj->original);

```

```

    free(obj->copy);
    free(obj->temp);
    free(obj);
}

```

JAVA:

```

class Solution {
    int[] original;
    int[] copy;
    Random rand = new Random();

    void swap(int i, int j) {
        int temp = copy[i];
        copy[i] = copy[j];
        copy[j] = temp;
    }

    public Solution(int[] nums) {
        copy = nums;
        original = nums.clone();
    }

    public int[] reset() {
        copy = original;
        original = original.clone();
        return original;
    }

    public int[] shuffle() {
        for (int i = 0; i < copy.length; i++) {
            int ra = copy.length-i;
            swap(i, (rand.nextInt(ra)+i));
        }
        return copy;
    }
}

```

6.2 MinStack

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the **MinStack** class:

- **MinStack()** initializes the stack object.
- **void push(int val)** pushes the element **val** onto the stack.
- **void pop()** removes the element on the top of the stack.
- **int top()** gets the top element of the stack.
- **int getMin()** retrieves the minimum element in the stack.

You must implement a solution with $O(1)$ time complexity for each function.

Example 1:

Input

```
["MinStack","push","push","push","getMin","pop","top","getMin"]  
[[],[-2],[0],[-3],[[],[],[],[]]]
```

Output

```
[null,null,null,null,-3,null,0,-2]
```

Explanation

```
MinStack minStack = new MinStack();  
minStack.push(-2);  
minStack.push(0);  
minStack.push(-3);  
minStack.getMin(); // return -3  
minStack.pop();  
minStack.top();    // return 0  
minStack.getMin(); // return -2
```

C:

```
typedef struct {  
    int* stack;  
    int* minStack;  
    int top;  
    int minTop;  
    int capacity;  
} MinStack;
```

```
MinStack* minStackCreate() {  
    MinStack* stack = (MinStack*)malloc(sizeof(MinStack));  
    stack->stack = (int*)malloc(sizeof(int));
```

```

    stack->minStack = (int*)malloc(sizeof(int));
    stack->top = -1;
    stack->minTop = -1;
    stack->capacity = 1;
    return stack;
}

void minStackPush(MinStack* obj, int val) {
    if (obj->top == obj->capacity - 1) {
        obj->capacity *= 2;
        obj->stack = (int*)realloc(obj->stack, sizeof(int) * obj->capacity);
    }
    obj->stack[++obj->top] = val;

    if (obj->minTop == -1 || val <= obj->minStack[obj->minTop]) {
        if (obj->minTop == obj->capacity - 1) {
            obj->minStack = (int*)realloc(obj->minStack, sizeof(int) * obj->capacity);
        }
        obj->minStack[++obj->minTop] = val;
    }
}

void minStackPop(MinStack* obj) {
    if (obj->stack[obj->top] == obj->minStack[obj->minTop]) {
        obj->minTop--;
    }
    obj->top--;
}

int minStackTop(MinStack* obj) {
    return obj->stack[obj->top];
}

int minStackGetMin(MinStack* obj) {
    return obj->minStack[obj->minTop];
}

void minStackFree(MinStack* obj) {
    free(obj->stack);
    free(obj->minStack);
    free(obj);
}

```

JAVA:

```

class MinStack {
    Stack<Integer> original;
    Stack<Integer> minimum;
}

```

```
public MinStack() {
    original = new Stack<Integer>();
    minimum = new Stack<Integer>();
}

public void push(int val) {
    original.push(val);
    if (minimum.isEmpty() || val <= minimum.peek()) {
        minimum.push(val);
    }
}

public void pop() {
    if (original.peek().equals(minimum.peek())) {
        minimum.pop();
    }
    original.pop();
}

public int top() {
    return original.peek();
}

public int getMin() {
    return minimum.peek();
}
}
```


7. MATH

7.1 Fizz Buzz

Given an integer n , return a string array `answer` (**1-indexed**) where:

- `answer[i] == "FizzBuzz"` if i is divisible by 3 and 5.
- `answer[i] == "Fizz"` if i is divisible by 3.
- `answer[i] == "Buzz"` if i is divisible by 5.
- `answer[i] == i` (as a string) if none of the above conditions are true.

Example 1:

Input: $n = 3$

Output: ["1", "2", "Fizz"]

Example 2:

Input: $n = 5$

Output: ["1", "2", "Fizz", "4", "Buzz"]

Example 3:

Input: $n = 15$

Output: ["1", "2", "Fizz", "4", "Buzz", "Fizz", "7", "8", "Fizz", "Buzz", "11", "Fizz", "13", "14", "FizzBuzz"]

C:

```
char ** fizzBuzz(int n, int* returnSize){
    *returnSize = n;
    char **result = (char **)malloc(sizeof(char *)*n);
    for (int i = 0; i < n; i++) {
        int by_3 = (i+1)%3;
        int by_5 = (i+1)%5;
        int by_15 = (i+1)%15;
        if (by_15 == 0) {
            result[i] = (char *)malloc(sizeof(char)*9);
            sprintf(result[i],"%s","FizzBuzz");
        } else if (by_3 == 0) {
            result[i] = (char *)malloc(sizeof(char)*5);
            result[i] = "Fizz";
        } else if (by_5 == 0) {
            result[i] = (char *)malloc(sizeof(char)*5);
            result[i] = "Buzz";
        } else {
            result[i] = (char *)malloc(sizeof(char)*2);
            result[i] = (char *)itoa(i+1, result[i], 10);
        }
    }
    return result;
}
```

```

        } else if (by_5 == 0) {
            result[i] = (char *)malloc(sizeof(char)*5);
            strcpy(result[i], "Buzz");
        } else {
            result[i] = (char *)malloc(sizeof(char)*10);
            sprintf(result[i], "%d", i+1);
        }
    }
    return result;
}

```

JAVA:

```

class Solution {
    public List<String> fizzBuzz(int n) {
        List<String> result = new ArrayList<String>();
        for (int i = 1; i <= n; i++) {
            int by_3 = i%3;
            int by_5 = i%5;
            int by_15 = i%15;
            //int flag = 0;
            if (by_15 == 0) {
                result.add("FizzBuzz");
            } else if (by_3 == 0) {
                result.add("Fizz");
            } else if (by_5 == 0) {
                result.add("Buzz");
            } else {
                result.add(Integer.toString(i));
            }
        }
        return result;
    }
}

```

7.2 Count Primes

Given an integer n , return the number of prime numbers that are strictly less than n .

Example 1:

Input: $n = 10$

Output: 4

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

Example 2:

Input: n = 0

Output: 0

Example 3:

Input: n = 1

Output: 0

C:

```
int countPrimes(int n){
    int i = 0;
    int j = 0;
    int count = 0;

    if (n == 0) return 0;
    bool num[n];

    for (int i = 0 ; i < n; i++) {
        num[i] = false;
    }

    if (n < 2) return 0;
    for (i = 2; i <= (int)sqrt(n); i++) {
        if (num[i] == false) {
            for (j = i*i; j < (n) ;j+=i) {
                num[j] = true;
            }
        }
    }

    for (i = 2; i < n; i++) {
        if (num[i] == false) count++;
    }
    return count;
}
```

JAVA:

```
class Solution {
    public int countPrimes(int n) {
        if (n < 0) return 0;
```

```

boolean num[] = new boolean[n];

int count = 0;
int i = 0;
int j = 0;

for (i = 0; i < n; i++) {
    num[i] = false;
}

for (i = 2; i <= (int)Math.sqrt(n); i++) {
    if (num[i] == false) {
        for (j = i*i; j < n; j+=i) {
            num[j] = true;
        }
    }
}

for (i = 2; i < n; i++) {
    if (num[i] == false) count++;
}
return count;
}
}

```

7.3 Power Of Three

Given an integer `n`, return `true` if it is a power of three. Otherwise, return `false`.

An integer `n` is a power of three, if there exists an integer `x` such that `n == 3x`.

Example 1:

Input: `n = 27`

Output: `true`

Explanation: `27 = 33`

Example 2:

Input: `n = 0`

Output: `false`

Explanation: There is no x where $3^x = 0$.

Example 3:

Input: $n = -1$

Output: false

Explanation: There is no x where $3^x = (-1)$.

C:

```
bool isPowerOfThree(int n){
    int i = 0;
    int res = 1;
    for (i = 0; i < 19; i++) res *= 3;
    return (n > 0 && (res%n == 0));
}
```

JAVA:

```
class Solution {
    public boolean isPowerOfThree(int n) {
        if (n < 1) return false;
        while (n % 3 == 0) {
            n /= 3;
        }
        return n == 1;
    }
}
```

7.4 Roman To Integer

Roman numerals are represented by seven different symbols: **I**, **V**, **X**, **L**, **C**, **D** and **M**.

| Symbol | Value |
|--------|-------|
| I | 1 |
| V | 5 |
| X | 10 |
| L | 50 |
| C | 100 |
| D | 500 |
| M | 1000 |

For example, 2 is written as **II** in Roman numeral, just two ones added together. 12 is written as **XII**, which is simply **X + II**. The number 27 is written as **XXVII**, which is **XX + V + II**.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not **IIII**. Instead, the number four is written as **IV**. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as **IX**. There are six instances where subtraction is used:

- **I** can be placed before **V** (5) and **X** (10) to make 4 and 9.
- **X** can be placed before **L** (50) and **C** (100) to make 40 and 90.
- **C** can be placed before **D** (500) and **M** (1000) to make 400 and 900.

Given a roman numeral, convert it to an integer.

Example 1:

Input: `s = "III"`

Output: 3

Explanation: `III = 3`.

Example 2:

Input: `s = "LVIII"`

Output: 58

Explanation: `L = 50, V = 5, III = 3`.

Example 3:

Input: `s = "MCMXCIV"`

Output: 1994

Explanation: `M = 1000, CM = 900, XC = 90 and IV = 4`.

is no x where $3^x = (-1)$.

C:

```
int romanToInt(char * s){
    int result = 0;
    int i = 0;
```

```

// Create a lookup table for Roman numerals
char roman[7] = {'I', 'V', 'X', 'L', 'C', 'D', 'M'};
int value[7] = {1, 5, 10, 50, 100, 500, 1000};
int length = strlen(s);

for (i = 0; i < length; i++) {
    // Find the corresponding value for the current Roman numeral
    int current = 0;
    for (int j = 0; j < 7; j++) {
        if (s[i] == roman[j]) {
            current = value[j];
            break;
        }
    }

    // Check if subtraction is required
    if (i < length - 1) {
        int next = 0;
        for (int j = 0; j < 7; j++) {
            if (s[i+1] == roman[j]) {
                next = value[j];
                break;
            }
        }

        if (current < next) {
            result += (next - current);
            i++;
            continue;
        }
    }

    // Add the current value to the result
    result += current;
}

return result;
}

```

JAVA:

```

class Solution {
    public int romanToInt(String s) {
        int result = 0;
        int i = 0;
        HashMap <Character, Integer> hm = new HashMap<Character, Integer>();
        hm.put('I',1);
        hm.put('V',5);

```

```

hm.put('X',10);
hm.put('L',50);
hm.put('C',100);
hm.put('D',500);
hm.put('M',1000);

for (i = 0; i < s.length(); i++) {
    if (s.charAt(i) == 'I') {
        if ((i < s.length()-1) && (s.charAt(i+1) == 'V' || s.charAt(i+1) == 'X')) {
            result += (hm.get(s.charAt(i+1)) - hm.get(s.charAt(i)));
            i++;
        } else {
            result += hm.get(s.charAt(i));
        }
    } else if (s.charAt(i) == 'X') {
        if ((i < s.length()-1) && (s.charAt(i+1) == 'L' || s.charAt(i+1) == 'C')) {
            result += (hm.get(s.charAt(i+1)) - hm.get(s.charAt(i)));
            i++;
        } else {
            result += hm.get(s.charAt(i));
        }
    } else if (s.charAt(i) == 'C') {
        if ((i < s.length()-1) && (s.charAt(i+1) == 'D' || s.charAt(i+1) == 'M')) {
            result += (hm.get(s.charAt(i+1)) - hm.get(s.charAt(i)));
            i++;
        } else {
            result += hm.get(s.charAt(i));
        }
    } else {
        result += hm.get(s.charAt(i));
    }
}

return result;
}
}

```


8. OTHERS

8.1 Number Of 1 Bits

Write a function that takes the binary representation of an unsigned integer and returns the number of '1' bits it has (also known as the [Hamming weight](#)).

Note:

- Note that in some languages, such as Java, there is no unsigned integer type. In this case, the input will be given as a signed integer type. It should not affect your implementation, as the integer's internal binary representation is the same, whether it is signed or unsigned.
- In Java, the compiler represents the signed integers using [2's complement notation](#). Therefore, in **Example 3**, the input represents the signed integer. **-3**.

Example 1:

Input: n = 000000000000000000000000000000001011

Output: 3

[illegible]

Example 2:

Input: n = 0000000000000000000000000000010000000

Output: 1

[illegible]

Example 3:

Input: n = 1111111111111111111111111111111101

Output: 31

[illegible]

"7

C:

```
int hammingWeight(uint32_t n) {
    int count = 0;
    while (n != 0) {
        n = (n & n-1);
        count++;
    }
    return count;
}
```

JAVA:

```
public class Solution {
    // you need to treat n as an unsigned value
    public int hammingWeight(int n) {
        int count = 0;
        for (int i = 0; i < 32; i++) {
            if ((n & 1) == 1) count++;
            n >>= 1;
        }
        return count;
    }
}
```

8.2 Hamming Distance

The [Hamming distance](#) between two integers is the number of positions at which the corresponding bits are different.

Given two integers x and y , return the **Hamming distance** between them.

Example 1:

Input: $x = 1, y = 4$

Output: 2

Explanation:

```
1   (0 0 0 1)
4   (0 1 0 0)
    ↑   ↑
```

The above arrows point to positions where the corresponding bits are different.

Example 2:

Input: x = 3, y = 1

Output: 1

C:

```
int hammingDistance(int x, int y){
    int result = x ^ y;
    int count = 0;
    while (result != 0) {
        result = result & (result-1);
        count++;
    }
    return count;
}
```

JAVA:

```
class Solution {
    public int hammingDistance(int x, int y) {
        int count = 0;
        for (int i = 0; i < 32; i++) {
            if ((x & 1) != (y & 1)) count++;

            x >>= 1;
            y >>= 1;
        }
        return count;
    }
}
```

8.3 Reverse Bits

Reverse bits of a given 32 bits unsigned integer.

Note:

- Note that in some languages, such as Java, there is no unsigned integer type. In this case, both input and output will be given as a signed integer type. They should not affect your implementation, as the integer's internal binary representation is the same, whether it is signed or unsigned.
- In Java, the compiler represents the signed integers using [2's complement notation](#). Therefore, in **Example 2** above, the input represents the signed integer -3 and the output represents the signed integer -1073741825.

Example 1:

Input: n = 00000010100101000001111010011100

Output: 964176192 (00111001011110000010100101000000)

Explanation: The input binary string 00000010100101000001111010011100 represents the unsigned integer 43261596, so return 964176192 which its binary representation is 00111001011110000010100101000000.

Example 2:

Input: n = 1111111111111111111111111111101

Output: 3221225471 (1011111111111111111111111111111)

Explanation: The input binary string 111111111111111111111111111101 represents the unsigned integer 4294967293, s

C:

```
uint32_t reverseBits(uint32_t n) {
    unsigned int result = 0;
    unsigned int power = 31;
    while (n != 0) {
        result = (result | ((n & 1) << power));
        power -= 1;
        n >>= 1;
    }
    return result;
}
```

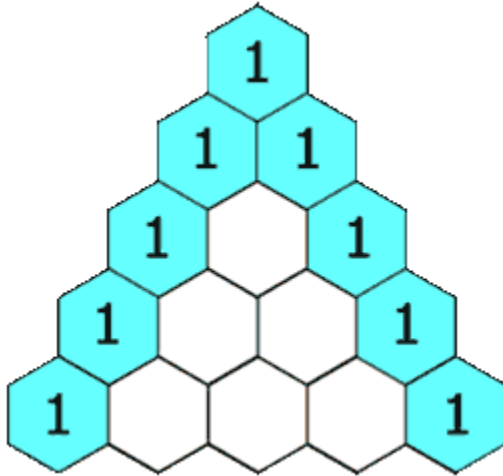
JAVA:

```
public class Solution {
    public int reverseBits(int n) {
        int result = 0;
        for (int i = 31; i >= 0; i--) {
            int temp = n&1;
            result = result | (temp<<i);
            n = n >> 1;
        }
        return result;
    }
}
```

8.4 Pascals Triangle

Given an integer `numRows`, return the first numRows of **Pascal's triangle**.

In **Pascal's triangle**, each number is the sum of the two numbers directly above it as shown:



Example 1:

Input: numRows = 5

Output: [[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]

Example 2:

Input: numRows = 1

Output: [[1]]

C:

```
int** generate(int numRows, int* returnSize, int** returnColumnSizes){
```

```
    *returnSize = numRows;
    *returnColumnSizes = (int *)malloc(sizeof(int)*numRows);
    int **result = (int **) malloc (sizeof(int *)*numRows);
```

```
    int i = 0;
    int j = 0;
    for (i = 0; i < numRows; i++) {
        (*returnColumnSizes)[i] = i+1;
        result[i] = (int *) malloc (sizeof(int)*(i+1));
        result[i][0] = 1;
        result[i][i] = 1;
        for (j = 1; j < i; j++) {
            result[i][j] = result[i-1][j-1]+result[i-1][j];
        }
    }
}
```

```
    return result;
}
```

JAVA:

```
class Solution {
    public List<List<Integer>> generate(int numRows) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        result.add(new ArrayList<Integer>());
        result.get(0).add(1);
        for (int i = 1; i < numRows; i++) {
            List<Integer> row = new ArrayList<Integer>();
            List<Integer> prevrow = result.get(i-1);
            row.add(1);
            for (int j = 1; j < i; j++) {
                row.add(prevrow.get(j-1)+prevrow.get(j));
            }
            row.add(1);
            result.add(row);
        }
        return result;
    }
}
```

8.5 Valid Prantheses

Given a string `s` containing just the characters `'('`, `')'`, `'{'`, `'}'`, `'['` and `']'`, determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.
3. Every close bracket has a corresponding open bracket of the same type.

Example 1:

Input: `s = "()"`

Output: `true`

Example 2:

Input: `s = "()[]{}"`

Output: true

Example 3:

Input: s = "([)"

Output: false

C:

```
typedef struct stack_s {
    int top;
    int arr[10000];
}stack_t;

stack_t st;

int mapping[3][2] = {
    {'}', '{'},
    {'}', '{'},
    {'}', '('}
};

void push(int element) {
    st.arr[++st.top] = element;
}

int pop() {
    if(st.top == -1) return -1;
    else {
        return st.arr[st.top--];
    }
}

bool isValid(char * s){
    int i = 0;
    int j = 0;
    st.top = -1;

    for (i = 0; i < strlen(s); i++) {
        int flag = 0;
        int matched = 0;
        int original = s[i];
        for (j = 0; j < 3; j++) {
            if (mapping[j][0] == original) {
                flag = 0;
                int expected = mapping[j][1];
                int popped;
```

```

        if (st.top == -1) {
            popped = -1;
        } else {
            popped = pop();
            flag = 0;
            matched = 1;
        }
        if (popped != expected) {
            return false;
        }
    } else {
        flag = 1;
    }
}
if (flag == 1 && matched == 0) {
    push(s[i]);
    flag = 0;
}
}
if (st.top == -1) {
    return true;
}
return false;
}

```

JAVA:

```

class Solution {
    public boolean isValid(String s) {
        HashMap <Character,Character> hm = new HashMap<>();
        hm.put('}', '{');
        hm.put(')', '(');
        hm.put(']', '[');

        Stack<Character> st = new Stack<>();
        for (int i = 0; i < s.length(); i++) {
            if (!hm.containsKey(s.charAt(i))) {
                st.push(s.charAt(i));
            } else {
                Character popped;
                if (st.isEmpty()) {
                    popped = '#';
                } else {
                    popped = st.pop();
                }
                if (popped != hm.get(s.charAt(i))) return false;
            }
        }
    }
}

```



```

    }
    return st.isEmpty();
}
}

```

8.6 Missing Number

Given an array `nums` containing `n` distinct numbers in the range `[0, n]`, return *the only number in the range that is missing from the array*.

Example 1:

Input: `nums = [3,0,1]`

Output: 2

Explanation: `n = 3` since there are 3 numbers, so all numbers are in the range `[0,3]`.
2 is the missing number in the range since it does not appear in `nums`.

Example 2:

Input: `nums = [0,1]`

Output: 2

Explanation: `n = 2` since there are 2 numbers, so all numbers are in the range `[0,2]`.
2 is the missing number in the range since it does not appear in `nums`.

Example 3:

Input: `nums = [9,6,4,2,3,5,7,0,1]`

Output: 8

Explanation: `n = 9` since there are 9 numbers, so all numbers are in the range `[0,9]`.
8 is the missing number in the range.

C:

```

int missingNumber(int* nums, int numsSize){
    int sum = 0;
    int actual_sum = 0;
    int i = 0;
    for (i = 0; i < numsSize; i++) {
        sum += nums[i];
    }
}

```

```
    actual_sum = (numsSize*(numsSize+1))/2;  
    return actual_sum - sum;  
}
```

JAVA:

```
class Solution {  
    public int missingNumber(int[] nums) {  
        int missing = nums.length;  
        for (int i = 0; i < nums.length; i++) {  
            missing ^= i ^ nums[i];  
        }  
        return missing;  
    }  
}
```

9. DYNAMIC PROGRAMMING

9.1 Climbing Stairs

You are climbing a staircase. It takes n steps to reach 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: $n = 2$

Output: 2

Explanation: There are two ways to climb to the top.

1. 1 step + 1 step
2. 2 steps

Example 2:

Input: $n = 3$

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

C:

```
int climbStairs(int n){
    if (n == 0) return 0;
    if (n == 1) return 1;
    if (n == 2) return 2;
    int *result = (int *)malloc(sizeof(int)*(n+1));
    int i = 0;
    result[0] = 0;
    result[1] = 1;
    result[2] = 2;
    for (i = 3; i <= n; i++) {
        result[i] = result[i-1] + result[i-2];
    }
    return result[n];
}
```

```
}
```

JAVA:

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

9.2 Best Time To Buy And Sell Stock

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.

C:

```
int max(int a, int b) {
    return (a > b) ? a:b;
}

int maxProfit(int* prices, int pricesSize){
    int min_price = INT_MAX;
    int i = 0;
    int max_profit = 0;
    for (i = 0; i < pricesSize; i++) {
        if (min_price > prices[i]) min_price = prices[i];
        max_profit = max(max_profit, prices[i]-min_price);
    }
    return max_profit;
}
```

JAVA:

```
class Solution {
    public int maxProfit(int[] prices) {
        int min_price = Integer.MAX_VALUE;
        int max_profit = 0;
        for (int i = 0; i < prices.length; i++) {
            if (prices[i] < min_price) min_price = prices[i];
            if ((prices[i]-min_price) > max_profit){
                max_profit = prices[i]-min_price;
            }
        }
        return max_profit;
    }
}
```

9.3 Maximum SubArray

Given an integer array `nums`, find the subarray with the largest sum, and return *its sum*.

Example 1:

Input: `nums = [-2,1,-3,4,-1,2,1,-5,4]`

Output: 6

Explanation: The subarray `[4,-1,2,1]` has the largest sum 6.

Example 2:

Input: `nums = [1]`

Output: 1

Explanation: The subarray [1] has the largest sum 1.

Example 3:

Input: nums = [5,4,-1,7,8]

Output: 23

Explanation: The subarray [5,4,-1,7,8] has the largest sum 23.

C:

```
int max(int a, int b) {
    return (a > b) ? a:b;
}
int maxSubArray(int* nums, int numsSize){
    int i = 0;
    int current_array = 0;
    int max_sub_array = 0;
    current_array = nums[0];
    max_sub_array = nums[0];
    for (i = 1; i < numsSize; i++) {
        current_array = max(nums[i], nums[i]+current_array);
        max_sub_array = max(max_sub_array, current_array);
    }
    return max_sub_array;
}
```

JAVA:

```
class Solution {
    public int maxSubArray(int[] nums) {
        int max_sub_array = nums[0];
        int current_sub_array = nums[0];
        for (int i = 1; i < nums.length; i++) {
            current_sub_array = Math.max(nums[i], nums[i]+current_sub_array);
            max_sub_array = Math.max(current_sub_array, max_sub_array);
        }
        return max_sub_array;
    }
}
```

9.4 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 systems connected and **it will automatically contact the police if two adjacent houses were broken into on the same night.**

Given an integer array `nums` representing the amount of money of each house, return *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.

: The subarray [5,4,-1,7,8] has the largest sum 23.

C:

```
int max(int a, int b) {
    return (a > b) ? a:b;
}
int rob(int* nums, int numsSize){

    int i = 0;
    if (numsSize == 0) return 0;
    if (numsSize == 1) return nums[0];
    int* result = malloc(numsSize * sizeof(int));
    result[0] = nums[0];
    result[1] = max(nums[0],nums[1]);
    for (i = 2; i < numsSize; i++) {
        result[i] = max(result[i-1],result[i-2]+nums[i]);
    }
}
```

```
    return result[numsSize-1];
```

JAVA:

```
class Solution {  
    public int rob(int[] nums) {  
        int N = nums.length;  
        int result[] = new int[N];  
        if (N == 0) return 0;  
        if (N == 1) return nums[0];  
        if (N == 2) return Math.max(nums[0], nums[1]);  
        result[0] = nums[0];  
        result[1] = Math.max(nums[0], nums[1]);  
        for (int i = 2; i < nums.length; i++) {  
            result[i] = Math.max(result[i-1], result[i-2]+nums[i]);  
        }  
        return result[N-1];  
    }  
}
```