# Array

## 1. LeetCode 1 2 Sum (easy)

- 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]
- **Constraints:**
  - `2 <= nums.length <= 10^4`
  - `-10^9 <= nums[i] <= 10^9`
  - `-10^9 <= target <= 10^9`
  - **Only one valid answer exists.**
- **Follow-up:** Can you come up with an algorithm that is less than `O(n^2)` time complexity?

## Solution 1: brute force

```java
public int[] twoSum(int[] nums, int target) {
    for (int i = 0; i < nums.length; i++) {
        for (int j = i + 1; j < nums.length; j++) {
            if (nums[i] + nums[j] == target) {
                return new int[]{i, j};
            }
        }
    }
    return null; // throw new IllegalArgumentException("No two sum solution");
}
```

Time Complexity: O(n^2)

Space Complexity: O(1)

## Solution 2: one pass hashmap

```java
public int[] twoSum(int[] nums, int target) {
    Map<Integer, Integer> map = new HashMap<>();
    for (int i = 0; i < nums.length; i++) {
        int complement = target - nums[i];
        if (map.containsKey(complement)) {
            return new int[]{map.get(complement), i};
        }
        map.put(nums[i], i);
    }
    return null;
}
```

Time Complexity: O(n)

Space Complexity: O(n)

# 2. LeetCode 121 Best Time to Buy and Sell Stock (easy)

- You are given an array `prices` where `prices[i]` is the price of a given stock on the `ith` 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.
- **Constraints:**
  - `1 <= prices.length <= 10^5`
  - `0 <= prices[i] <= 10^4`

## Solution

```java
public int maxProfit(int[] prices) {
    if (prices == null || prices.length == 0) {
        return 0;
    }
    int minPriceSoFar = Integer.MAX_VALUE;
    int maxProfit = 0;
    for (int price : prices) {
        minPriceSoFar = Math.min(minPriceSoFar, price);
        maxProfit = Math.max(maxProfit, price - minPriceSoFar);
    }
    return maxProfit;
}
```

Time Complexity: O(n)

Space Complexity: O(1)

# 3. LeetCode 217 Contains Duplicate (easy)

- 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
- **Constraints:**
  - `1 <= nums.length <= 10^5`
  - `-10^9 <= nums[i] <= 10^9`

## Solution 1: sort

```java
public boolean containsDuplicate(int[] nums) {
    if (nums == null || nums.length == 0) {
        return false;
    }
    Arrays.sort(nums);
    for (int i = 0; i < nums.length - 1; i++) {
        if (nums[i] == nums[i + 1]) {
            return true;
        }
    }
    return false;
}
```

Time Complexity: O(nlogn)

Space Complexity: O(1)

## Solution 2: hashset

```java
public boolean containsDuplicate(int[] nums) {
    if (nums == null || nums.length == 0) {
        return false;
    }
    Set<Integer> set = new HashSet<>();
    for (int num : nums) {
        if (set.contains(num)) {
            return true;
        }
        set.add(num);
    }
    return false;
}
```

Time Complexity: O(n)

Space Complexity: O(n)

## 4. LeetCode 238 Product of Array Except Self (medium)

- Given an integer array `nums`, return *an array* `answer` *such that* `answer[i]` *is equal to the product of all the elements of* `nums` *except* `nums[i]`.
- The product of any prefix or suffix of `nums` is **guaranteed** to fit in a **32-bit** integer.
- You must write an algorithm that runs in `O(n)` time and without using the division operation.
- **Example 1:**
  - **Input:** nums = [1,2,3,4]
  - **Output:** [24,12,8,6]
- **Example 2:**
  - **Input:** nums = [-1,1,0,-3,3]
  - **Output:** [0,0,9,0,0]
- **Constraints:**
  - `2 <= nums.length <= 10^5`
  - `-30 <= nums[i] <= 30`
  - The product of any prefix or suffix of `nums` is **guaranteed** to fit in a **32-bit** integer.
- **Follow up:** Can you solve the problem in `O(1)` extra space complexity? (The output array **does not** count as extra space for space complexity analysis.)

## Solution

```java
public int[] productExceptSelf(int[] nums) {
    if (nums == null || nums.length == 0) {
        throw new IllegalArgumentException("illegal input array");
    }
    int[] result = new int[nums.length];
    result[0] = 1;
    for (int i = 1; i < nums.length; i++) {
        result[i] = result[i - 1] * nums[i - 1];
    }
    int right = 1;
    for (int i = nums.length - 1; i >= 0; i--) {
        result[i] = result[i] * right;
        right *= nums[i];
```

```
    }
    return result;
}
```

Time Complexity: O(n)

Space Complexity: O(1)

# 5. LeetCode 53 Maximum Subarray (easy)

- Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest sum and return *its sum*.
- A **subarray** is a **contiguous** part of an array.
- **Example 1:**
  - **Input:** nums = [-2,1,-3,4,-1,2,1,-5,4]
  - **Output:** 6
  - **Explanation:** [4,-1,2,1] has the largest sum = 6.
- **Example 2:**
  - **Input:** nums = [1]
  - **Output:** 1
- **Example 3:**
  - **Input:** nums = [5,4,-1,7,8]
  - **Output:** 23
- **Constraints:**
  - `1 <= nums.length <= 10^5`
  - `-10^4 <= nums[i] <= 10^4`
- **Follow up:** If you have figured out the `O(n)` solution, try coding another solution using the **divide and conquer** approach, which is more subtle.

## Solution: dynamic programming

```
public int maxSubArray(int[] nums) {
    if (nums == null || nums.length == 0) {
        return 0;
    }
    int lastMax = nums[0];
    int globalMax = nums[0];
    for (int i = 1; i < nums.length; i++) {
        lastMax = Math.max(lastMax + nums[i], nums[i]);
        // 继承遗产 or 另起炉灶
        globalMax = Math.max(globalMax, lastMax);
    }
    return globalMax;
}
```

Time Complexity: O(n)

Space Complexity: O(1)

# 6. LeetCode 152 Maximum Product Subarray (medium)

- Given an integer array `nums`, find a contiguous non-empty subarray within the array that has the largest product, and return *the product*.
- The test cases are generated so that the answer will fit in a **32-bit** integer.
- A **subarray** is a contiguous subsequence of the array.
- **Example 1:**
  - **Input:** nums = [2,3,-2,4]
  - **Output:** 6
  - **Explanation:** [2,3] has the largest product 6.
- **Example 2:**
  - **Input:** nums = [-2,0,-1]
  - **Output:** 0
  - **Explanation:** The result cannot be 2, because [-2,-1] is not a subarray.
- **Constraints:**
  - `1 <= nums.length <= 2 * 10^4`
  - `-10 <= nums[i] <= 10`
  - The product of any prefix or suffix of `nums` is **guaranteed** to fit in a **32-bit** integer.

## Solution

```java
public int maxProduct(int[] nums) {
    if (nums == null || nums.length == 0) {
        return 0;
    }
    int lastMin = nums[0];
    int lastMax = nums[0];
    int globalMax = nums[0];
    for (int i = 1; i < nums.length; i++) {
        int temp = lastMin;
        lastMin = Math.min(Math.min(lastMin * nums[i], lastMax * nums[i]), nums[i]);
        lastMax = Math.max(Math.max(temp * nums[i], lastMax * nums[i]), nums[i]);
        globalMax = Math.max(globalMax, lastMax);
    }
    return globalMax;
}
```

Time Complexity: O(n)

Space Complexity: O(1)

# 7. LeetCode 153 Find Minimum in Rotated Sorted Array (medium)

- Suppose an array of length `n` sorted in ascending order is **rotated** between `1` and `n` times. For example, the array `nums = [0,1,2,4,5,6,7]` might become:
  - `[4,5,6,7,0,1,2]` if it was rotated `4` times.
  - `[0,1,2,4,5,6,7]` if it was rotated `7` times.
- Notice that **rotating** an array `[a[0], a[1], a[2], ..., a[n-1]]` 1 time results in the array `[a[n-1], a[0], a[1], a[2], ..., a[n-2]]`.
- Given the sorted rotated array `nums` of **unique** elements, return *the minimum element of this array*.
- You must write an algorithm that runs in `O(log n) time.`

- **Example 1:**
  - **Input:** nums = [3,4,5,1,2]
  - **Output:** 1
  - **Explanation:** The original array was [1,2,3,4,5] rotated 3 times.
- **Example 2:**
  - **Input:** nums = [4,5,6,7,0,1,2]
  - **Output:** 0
  - **Explanation:** The original array was [0,1,2,4,5,6,7] and it was rotated 4 times.
- **Example 3:**
  - **Input:** nums = [11,13,15,17]
  - **Output:** 11
  - **Explanation:** The original array was [11,13,15,17] and it was rotated 4 times.
- **Constraints:**
  - `n == nums.length`
  - `1 <= n <= 5000`
  - `-5000 <= nums[i] <= 5000`
  - All the integers of `nums` are **unique**.
  - `nums` is sorted and rotated between `1` and `n` times.

## Solution: binary search

1. while condition
   - right > left + 1
2. update left
   - left = mid
3. update right
   - right = mid

```java
public int findMin(int[] nums) {
    if (nums == null || nums.length == 0) {
        throw new IllegalArgumentException("illegal input array");
    }
    if (nums.length == 1 || nums[0] < nums[nums.length - 1]) {
        return nums[0];
    }
    int left = 0, right = nums.length - 1;
    while (right > left + 1) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < nums[right]) {
            right = mid;
        } else {
            left = mid;
        }
    }
    return Math.min(nums[left], nums[right]);
}
```

Time Complexity: O(logn)

Space Complexity: O(1)

# 8. LeetCode 33 Search in Rotated Sorted Array (medium)

- There is an integer array `nums` sorted in ascending order (with **distinct** values).
- Prior to being passed to your function, `nums` is **possibly rotated** at an unknown pivot index `k` (`1 <= k < nums.length`) such that the resulting array is `[nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]` (**0-indexed**). For example, `[0,1,2,4,5,6,7]` might be rotated at pivot index `3` and become `[4,5,6,7,0,1,2]`.
- Given the array `nums` **after** the possible rotation and an integer `target`, return *the index of* `target` *if it is in* `nums`*, or* `-1` *if it is not in* `nums`.
- You must write an algorithm with `O(log n)` runtime complexity.
- **Example 1:**
  - **Input:** nums = [4,5,6,7,0,1,2], target = 0
  - **Output:** 4
- **Example 2:**
  - **Input:** nums = [4,5,6,7,0,1,2], target = 3
  - **Output:** -1
- **Example 3:**
  - **Input:** nums = [1], target = 0
  - **Output:** -1
- **Constraints:**
  - `1 <= nums.length <= 5000`
  - `-10^4 <= nums[i] <= 10^4`
  - All values of `nums` are **unique**.
  - `nums` is an ascending array that is possibly rotated.
  - `-10^4 <= target <= 10^4`

## Solution: binary search

```java
public int search(int[] nums, int target) {
    if (nums == null || nums.length == 0) {
        return -1;
    }
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        if (nums[left] == target) {
            return left;
        }
        if (nums[right] == target) {
            return right;
        }
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            return mid;
        } else if (nums[mid] > nums[left]) {
            if (target < nums[mid] && target > nums[left]) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        } else {
```

```
            if (target > nums[mid] && target < nums[right]) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
    }
    return -1;
}
```

Time Complexity: O(logn)

Space Complexity: O(1)

# 9. LeetCode 15 3 Sum (medium)

- Given an integer array nums, return all the triplets `[nums[i], nums[j], nums[k]]` such that `i != j`, `i != k`, and `j != k`, and `nums[i] + nums[j] + nums[k] == 0`.
- Notice that the solution set must not contain duplicate triplets.
- **Example 1:**
  - **Input:** nums = [-1,0,1,2,-1,-4]
  - **Output:** `[[-1,-1,2],[-1,0,1]]`
- **Example 2:**
  - **Input:** nums = []
  - **Output:** []
- **Example 3:**
  - **Input:** nums = [0]
  - **Output:** []
- **Constraints:**
  - `0 <= nums.length <= 3000`
  - `-10^5 <= nums[i] <= 10^5`

## Solution: sort

```
public List<List<Integer>> threeSum(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();
    if (nums == null || nums.length < 3) {
        return result;
    }
    return allTriples(nums, 0);
}

private List<List<Integer>> allTriples(int[] array, int target) {
    List<List<Integer>> result = new ArrayList<>();
    Arrays.sort(array);
    for (int i = 0; i < array.length - 2; i++) {
        if (i > 0 && array[i] == array[i - 1]) {
            continue;
        }
        int left = i + 1;
        int right = array.length - 1;
        while (left < right) {
            int temp = array[left] + array[right];
```

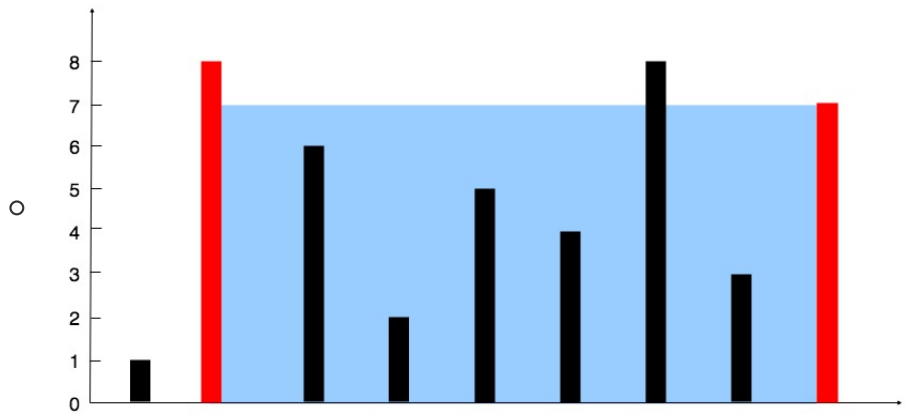
```
            if (temp + array[i] == target) {
                result.add(Arrays.asList(array[i], array[left], array[right]));
                left++;
                while (left < right && array[left] == array[left - 1]) {
                    left++;
                }
            } else if (temp + array[i] < target) {
                left++;
            } else {
                right--;
            }
        }
    }
    return result;
}
```

Time Complexity: O(n^2)

Space Complexity: O(1)

## 10. LeetCode 11 Container With Most Water (medium)

- You are given an integer array `height` of length `n`. There are `n` vertical lines drawn such that the two endpoints of the `ith` line are `(i, 0)` and `(i, height[i])`.
- Find two lines that together with the x-axis form a container, such that the container contains the most water.
- Return *the maximum amount of water a container can store*.
- **Notice** that you may not slant the container.
- **Example 1:**




  - **Input:** height = [1,8,6,2,5,4,8,3,7]
  - **Output:** 49
  - **Explanation:** The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49.
- **Example 2:**
  - **Input:** height = [1,1]
  - **Output:** 1
- **Constraints:**
  - `n == height.length`
  - `2 <= n <= 10^5`
  - `0 <= height[i] <= 10^4`

# Solution

```java
public int maxArea(int[] height) {
    if (height == null || height.length == 0) {
        return 0;
    }
    int result = 0;
    int left = 0, right = height.length - 1;
    while (left < right) {
        result = Math.max(result, (right - left) * Math.min(height[left],
height[right]));
        if (height[left] > height[right]) {
            right--;
        } else {
            left++;
        }
    }
    return result;
}
```

Time Complexity: O(n)

Space Complexity: O(1)

# Binary

- bit operation
  1. `&` (bitwise AND)
  2. `|` (bitwise OR)
  3. `~` (bitwise NOT)
  4. `^` (bitwise XOR)
  5. `<<` (left shift): 右侧补充零
  6. `>>` (right shift): 左侧补充原先的符号位
  7. `>>>`: 无符号右移, 忽略符号位, 空位都以 0 补齐
- building blocks: k = 0 -> least significant bit (从右往左数第 k 位)
  1. (**bit tester**) Given an integer x, test whether x's k-th bit is one.
     `int bit_k = (x >> k) & 1;`
  2. (**bit setter**) Given an integer x, set x's k-th bit to 1. (把第 k 位设成1, 其他位都不变)
     `x |= (1 << k);`
  3. (**bit resetter**) Given an integer x, set x's k-th bit to 0. (把第 k 位设成0, 其他位都不变)
     `x &= ~(1 << k);`

## 1. LeetCode 371 Sum of Two Integers (medium)

- Given two integers `a` and `b`, return *the sum of the two integers without using the operators* `+` *and* `-`.
- **Example 1:**
  - **Input:** a = 1, b = 2
  - **Output:** 3
- **Example 2:**
  - **Input:** a = 2, b = 3
  - **Output:** 5
- **Constraints:**
  - `-1000 <= a, b <= 1000`

### Solution: bit manipulation

```java
public int getSum(int a, int b) {
    while (b != 0) {
        int answer = a ^ b;
        int carry = (a & b) << 1;
        a = answer;
        b = carry;
    }
    return a;
}
```

Time Complexity: O(1)

Space Complexity: O(1)

## 2. LeetCode 191 Number of 1 Bits (easy)

- Write a function that takes 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 = 00000000000000000000000000001011
  - **Output:** 3
  - **Explanation:** The input binary string **00000000000000000000000000001011** has a total of three '1' bits.
- **Example 2:**
  - **Input:** n = 00000000000000000000000010000000
  - **Output:** 1
  - **Explanation:** The input binary string **00000000000000000000000010000000** has a total of one '1' bit.
- **Example 3:**
  - **Input:** n = 11111111111111111111111111111101
  - **Output:** 31
  - **Explanation:** The input binary string **11111111111111111111111111111101** has a total of thirty one '1' bits.
- **Constraints:**
  - The input must be a **binary string** of length `32`.
- **Follow up:** If this function is called many times, how would you optimize it?

## Solution: bit tester

```java
public int hammingWeight(int n) {
    int count = 0;
    for (int k = 0; k < 32; k++) {
        count += (n >> k) & 1;
    }
    return count;
}
```

Time Complexity: O(1)

Space Complexity: O(1)

# 3. LeetCode 338 Counting Bits (easy)

- Given an integer `n`, return *an array* `ans` *of length* `n + 1` *such that for each* `i` (`0 <= i <= n`), `ans[i]` *is the* ***number of*** `1`***'s*** *in the binary representation of* `i`.
- **Example 1:**
  - **Input:** n = 2
  - **Output:** [0,1,1]
  - **Explanation:**
    - 0 --> 0
    - 1 --> 1
    - 2 --> 10

- **Example 2:**
  - **Input:** n = 5
  - **Output:** [0,1,1,2,1,2]
  - **Explanation:**
    - 0 --> 0
    - 1 --> 1
    - 2 --> 10
    - 3 --> 11
    - 4 --> 100
    - 5 --> 101
- **Constraints:**
  - `0 <= n <= 10^5`
- **Follow up:**
  - It is very easy to come up with a solution with a runtime of `O(nlogn)`. Can you do it in linear time `O(n)` and possibly in a single pass?
  - Can you do it without using any built-in function (i.e., like `__builtin_popcount` in C++)?

## Solution: dynamic programming

```java
public int[] countBits(int n) {
    if (n < 0) {
        return null;
    }
    int[] result = new int[n + 1];
    for (int i = 1; i <= n; i++) {
        result[i] = result[i >> 1] + (i & 1);
    }
    return result;
}
```

Time Complexity: O(n)

Space Complexity: O(1)

# 4. LeetCode 268 Missing Number (easy)

- 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 since it does not appear in nums.
- **Constraints:**
    - `n == nums.length`
    - `1 <= n <= 10^4`
    - `0 <= nums[i] <= n`
    - All the numbers of `nums` are **unique**.
- **Follow up:** Could you implement a solution using only `O(1)` extra space complexity and `O(n)` runtime complexity?

# Solution 1: sort

```java
public int missingNumber(int[] nums) {
    if (nums == null || nums.length == 0) {
        return -1;
    }
    Arrays.sort(nums);
    if (nums[nums.length - 1] != nums.length) {
        return nums.length;
    }
    for (int i = 0; i < nums.length; i++) {
        if (nums[i] != i) {
            return i;
        }
    }
    return -1;
}
```

Time Complexity: O(nlogn)

Space Complexity: O(1)

# Solution 2: hashset

```java
public int missingNumber(int[] nums) {
    if (nums == null || nums.length == 0) {
        return -1;
    }
    Set<Integer> set = new HashSet<>();
    for (int num : nums) {
        set.add(num);
    }
    for (int i = 0; i <= nums.length; i++) {
        if (!set.contains(i)) {
            return i;
        }
    }
    return -1;
}
```

Time Complexity: O(n)

Space Complexity: O(n)

## Solution 3: sum

```java
public int missingNumber(int[] nums) {
    if (nums == null || nums.length == 0) {
        return -1;
    }
    long target = (nums.length + 0L) * (nums.length + 1) / 2;
    long sum = 0L;
    for (int num : nums) {
        sum += num;
    }
    return (int) (target - sum);
}
```

Time Complexity: O(n)

Space Complexity: O(1)

## Solution 4: bit manipulation (recommended)

step 1: XOR every element in input -> temp_result

step 2: temp_result XOR from 1 to n -> missing number

```java
public int missingNumber(int[] nums) {
    if (nums == null || nums.length == 0) {
        return -1;
    }
    int xor = 0;
    for (int num : nums) {
        xor ^= num;
    }
    for (int i = 1; i <= nums.length; i++) {
        xor ^= i;
    }
    return xor;
}
```

Time Complexity: O(n)

Space Complexity: O(1)

# 5. LeetCode 190 Reverse Bits (easy)

- 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 = 11111111111111111111111111111101
    - **Output:** 3221225471 (10111111111111111111111111111111)
    - **Explanation:** The input binary string **11111111111111111111111111111101** represents the unsigned integer 4294967293, so return 3221225471 which its binary representation is **10111111111111111111111111111111**.
- **Constraints:**
    - The input must be a **binary string** of length `32`
- **Follow up:** If this function is called many times, how would you optimize it?

# Solution

```java
public int reverseBits(int n) {
    int i = 0, j = 31;
    while (i < j) {
        n = swap(n, i, j);
        i++;
        j--;
    }
    return n;
}

private int swap(int x, int i, int j) {
    int bit_i = (x >> i) & 1;
    int bit_j = (x >> j) & 1;
    if (bit_i == bit_j) {
        return x;
    }
    return x ^ ((1 << i) + (1 << j));
}
```

Time Complexity: O(1)

Space Complexity: O(1)

# Dynamic Programming

**表象上填表格, 实质上用空间换取时间**

1. base case: M[0]
2. induction rule
    - 英文物理意义: M[i] represents what
    - 数学表达式: relationship between M[i] and M[i - 1], etc.

## 1. LeetCode 70 Climbing Stairs (easy)

- 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
- **Constraints:**
    - `1 <= n <= 45`

### Solution

1. base case
    - M[0] = 0
    - M[1] = 1
    - M[2] = 2
2. induction rule
    - M[i] represents how many distinct ways to climb to step i
    - M[i] = M[i - 1] + M[i - 2]

```java
public int climbStairs(int n) {
    // assumption: n >= 0
    if (n == 0 || n == 1 || n == 2) {
        return n;
    }
    int[] M = new int[n + 1];
    M[1] = 1;
    M[2] = 2;
    for (int i = 3; i <= n; i++) {
        M[i] = M[i - 1] + M[i - 2];
    }
    return M[n];
}
```

Time Complexity: O(n)

Space Complexity: O(n)

## Solution 1: space optimized dynamic programming

```java
public int climbStairs(int n) {
    // assumption: n >= 0
    if (n == 0 || n == 1 || n == 2) {
        return n;
    }
    int twoStep = 1;
    int oneStep = 2;
    int result = oneStep;
    for (int i = 3; i <= n; i++) {
        result = oneStep + twoStep;
        twoStep = oneStep;
        oneStep = result;
    }
    return result;
}
```

Time Complexity: O(n)

Space Complexity: O(1)

## Solution 2: matrix multiplication

```java
public int climbStairs(int n) {
    int[][] q = {{1, 1}, {1, 0}};
    int[][] result = power(q, n);
    return result[0][0];
}

private int[][] power(int[][] a, int n) {
    int[][] result = {{1, 0}, {0, 1}};
    while (n > 0) {
        if ((n & 1) == 1) {
            result = multiply(result, a);
        }
        n >>= 1;
        a = multiply(a, a);
```

```
    }
    return result;
}

private int[][] multiply(int[][] a, int[][] b) {
    int[][] result = new int[2][2];
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            result[i][j] = a[i][0] * b[0][j] + a[i][1] * b[1][j];
        }
    }
    return result;
}
```

Time Complexity: O(logn)

Space Complexity: O(1)

# Solution 3: Fibonacci formula

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

Time Complexity: O(logn) // pow method

Space Complexity: O(1)

# 2. LeetCode 322 Coin Change (medium)

- You are given an integer array `coins` representing coins of different denominations and an integer `amount` representing a total amount of money.
- Return *the fewest number of coins that you need to make up that amount*. If that amount of money cannot be made up by any combination of the coins, return `-1`.
- You may assume that you have an infinite number of each kind of coin.
- **Example 1:**
  - **Input:** coins = [1,2,5], amount = 11
  - **Output:** 3
  - **Explanation:** 11 = 5 + 5 + 1
- **Example 2:**
  - **Input:** coins = [2], amount = 3
  - **Output:** -1
- **Example 3:**
  - **Input:** coins = [1], amount = 0
  - **Output:** 0
- **Constraints:**
  - `1 <= coins.length <= 12`
  - `1 <= coins[i] <= 2^31 - 1`
  - `0 <= amount <= 10^4`

## Solution

1. base case: M[0] = 0
2. induction rule
    - M[i] represents the fewest number of coins to make up i
    - M[i] = Math.min(M[i], M[i - j] + 1), j is coin denomination

```java
public int coinChange(int[] coins, int amount) {
    // assumption: amount << Integer.MAX_VALUE
    if (coins == null || coins.length == 0 || amount < 0) {
        return -1;
    }
    int[] M = new int[amount + 1];
    Arrays.fill(M, amount + 1); // Integer.MAX_VALUE ??? why wrong ? overflow ?
    M[0] = 0;
    for (int i = 1; i <= amount; i++) {
        for (int j = 0; j < coins.length; j++) {
            if (coins[j] <= i) {
                M[i] = Math.min(M[i], M[i - coins[j]] + 1);
            }
        }
    }
    return M[amount] > amount ? -1 : M[amount];
}
```

Time Complexity: O(mn)

Space Complexity: O(n)

## 3. LeetCode 300 Longest Increasing Subsequence (medium)

- Given an integer array `nums`, return the length of the longest strictly increasing subsequence.
- A **subsequence** is a sequence that can be derived from an array by deleting some or no elements without changing the order of the remaining elements. For example, `[3,6,2,7]` is a subsequence of the array `[0,3,1,6,2,2,7]`.
- **Example 1:**
    - **Input:** nums = [10,9,2,5,3,7,101,18]
    - **Output:** 4
    - **Explanation:** The longest increasing subsequence is [2,3,7,101], therefore the length is 4.
- **Example 2:**
    - **Input:** nums = [0,1,0,3,2,3]
    - **Output:** 4
- **Example 3:**
    - **Input:** nums = [7,7,7,7,7,7,7]
    - **Output:** 1
- **Constraints:**
    - `1 <= nums.length <= 2500`
    - `-10^4 <= nums[i] <= 10^4`
- **Follow up:** Can you come up with an algorithm that runs in `O(nlogn)` time complexity?

# Solution 1: dynamic programming

| nums | 10 | 9 | 2 | 5 | 3 | 7 | 101 | 18 |
|---|---|---|---|---|---|---|---|---|
| M | 1 | 1 | 1 | 2 | 2 | 3 | 4 | 4 |

1. base case: M[0] = 1
2. induction rule
    - M[i] represents the length of the longest strictly increasing subsequence stopping at index i
    - M[i] = Math.max(M[i], M[j] + 1), nums[j] < nums[i], 0 <= j < i

```java
public int lengthOfLIS(int[] nums) {
    if (nums == null || nums.length == 0) {
        return 0;
    }
    int[] M = new int[nums.length];
    Arrays.fill(M, 1);
    int max = 1;
    for (int i = 1; i < nums.length; i++) {
        for (int j = 0; j < i; j++) {
            if (nums[j] < nums[i]) {
                M[i] = Math.max(M[i], M[j] + 1);
                max = Math.max(max, M[i]);
            }
        }
    }
    return max;
}
```

Time Complexity: O(n^2)

Space Complexity: O(n)

# Solution 2: binary search

```java
public int lengthOfLIS(int[] nums) {
    // ??? why
    List<Integer> subsequence = new ArrayList<>();
    subsequence.add(nums[0]);
    for (int i = 1; i < nums.length; i++) {
        if (nums[i] > subsequence.get(subsequence.size() - 1)) {
            subsequence.add(nums[i]);
        } else {
            int j = binarySearch(subsequence, nums[i]);
            subsequence.set(j, nums[i]);
        }
    }
    return subsequence.size();
}

private int binarySearch(List<Integer> list, int target) {
    int left = 0, right = list.size() - 1;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (list.get(mid) == target) {
```

```
            return mid;
        } else if (list.get(mid) < target) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    return left;
}
```

Time Complexity: O(nlogn)

Space Complexity: O(n)

# 4. LeetCode 1143 Longest Common Subsequence (medium)

- Given two strings `text1` and `text2`, return *the length of their longest* ***common subsequence***. If there is no **common subsequence**, return `0`.
- A **subsequence** of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.
  - For example, `"ace"` is a subsequence of `"abcde"`.
- A **common subsequence** of two strings is a subsequence that is common to both strings.
- **Example 1:**
  - **Input:** text1 = "abcde", text2 = "ace"
  - **Output:** 3
  - **Explanation:** The longest common subsequence is "ace" and its length is 3.
- **Example 2:**
  - **Input:** text1 = "abc", text2 = "abc"
  - **Output:** 3
  - **Explanation:** The longest common subsequence is "abc" and its length is 3.
- **Example 3:**
  - **Input:** text1 = "abc", text2 = "def"
  - **Output:** 0
  - **Explanation:** There is no such common subsequence, so the result is 0.
- **Constraints:**
  - `1 <= text1.length, text2.length <= 1000`
  - `text1` and `text2` consist of only lowercase English characters.

## Solution

| | _ | a | b | c | d | e |
|---|---|---|---|---|---|---|
| _ | 0 | 0 | 0 | 0 | 0 | 0 |
| a | 0 | 1 | 1 | 1 | 1 | 1 |
| c | 0 | 1 | 1 | 2 | 2 | 2 |
| e | 0 | 1 | 1 | 2 | 2 | 3 |

1. base case: `M[i][0] = m[0][j] = 0`

2. induction rule:
    - `M[i][j]` represents the length of longest common subsequence stopping at text1[i] and text2[j]
    - `M[i][j] = M[i - 1][j - 1] + 1` if text1[i] == text2[j]
    - `M[i][j] = Math.max(M[i - 1][j], M[i][j - 1])` otherwise

```java
public int longestCommonSubsequence(String text1, String text2) {
    if (text1 == null || text1.length() == 0 || text2 == null || text2.length() == 0) {
        return 0;
    }
    int[][] M = new int[text1.length() + 1][text2.length() + 1];
    for (int i = 1; i <= text1.length(); i++) {
        for (int j = 1; j <= text2.length(); j++) {
            if (text1.charAt(i - 1) == text2.charAt(j - 1)) {
                M[i][j] = M[i - 1][j - 1] + 1;
            } else {
                M[i][j] = Math.max(M[i - 1][j], M[i][j - 1]);
            }
        }
    }
    return M[text1.length()][text2.length()];
}
```

Time Complexity: O(mn)

Space Complexity: O(mn)

# 5. LeetCode 139 Word Break Problem (medium)

- Given a string `s` and a dictionary of strings `wordDict`, return `true` if `s` can be segmented into a space-separated sequence of one or more dictionary words.
- **Note** that the same word in the dictionary may be reused multiple times in the segmentation.
- **Example 1:**
    - **Input:** s = "leetcode", wordDict = ["leet","code"]
    - **Output:** true
    - **Explanation:** Return true because "leetcode" can be segmented as "leet code".
- **Example 2:**
    - **Input:** s = "applepenapple", wordDict = ["apple","pen"]
    - **Output:** true
    - **Explanation:** Return true because "applepenapple" can be segmented as "apple pen apple".
        - Note that you are allowed to reuse a dictionary word.
- **Example 3:**
    - **Input:** s = "catsandog", wordDict = ["cats","dog","sand","and","cat"]
    - **Output:** false
- **Constraints:**
    - `1 <= s.length <= 300`
    - `1 <= wordDict.length <= 1000`
    - `1 <= wordDict[i].length <= 20`
    - `s` and `wordDict[i]` consist of only lowercase English letters.
    - All the strings of `wordDict` are **unique**.

# Solution

s = "leetcode", wordDict = ["leet", "code"]

| s | | l | e | e | t | c | o | d | e |
|---|---|---|---|---|---|---|---|---|---|
| M | true | false | false | false | true | false | false | false | true |

1. base case: M[0] = true
2. induction rule
    - M[i] represents whether we can partition the first i letters of the input into words
    - M[i] = OR{M[j] for all 0 <= j < i and input[j..i) is a word}

```java
public boolean wordBreak(String s, List<String> wordDict) {
    if (s == null || s.length() == 0 || wordDict == null || wordDict.size() == 0) {
        return false;
    }
    Set<String> set = new HashSet<>();
    for (String word : wordDict) {
        set.add(word);
    }
    boolean[] M = new boolean[s.length() + 1];
    M[0] = true;
    for (int i = 1; i <= s.length(); i++) {
        for (int j = 0; j < i; j++) {
            if (M[j] && set.contains(s.substring(j, i))) {
                M[i] = true;
                break;
            }
        }
    }
    return M[s.length()];
}
```

Time Complexity: O(n^3) // string API is very slow

Space Complexity: O(n)

# 6. LeetCode 377 Combination Sum (medium)

- Given an array of **distinct** integers `nums` and a target integer `target`, return *the number of possible combinations that add up to* `target`.
- The test cases are generated so that the answer can fit in a **32-bit** integer.
- **Example 1:**
    - **Input:** nums = [1,2,3], target = 4
    - **Output:** 7
    - **Explanation:** The possible combination ways are:
        - (1, 1, 1, 1)
        - (1, 1, 2)
        - (1, 2, 1)
        - (1, 3)
        - (2, 1, 1)

    - (2, 2)
    - (3, 1)
    - Note that different sequences are counted as different combinations.
- **Example 2:**
    - **Input:** nums = [9], target = 3
    - **Output:** 0
- **Constraints:**
    - `1 <= nums.length <= 200`
    - `1 <= nums[i] <= 1000`
    - All the elements of `nums` are **unique**.
    - `1 <= target <= 1000`
- **Follow up:** What if negative numbers are allowed in the given array? How does it change the problem? What limitation we need to add to the question to allow negative numbers?

# Solution

nums = [1, 2, 3], target = 4

| target | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| M | 1 | 1 | 2 | 4 | 7 |

1. base case: M[0] = 1
2. induction rule
    - M[i] represents the number of possible combinations to add up to i
    - M[i] = sum(M[i - nums[j])

```
public int combinationSum4(int[] nums, int target) {
    if (nums == null || nums.length == 0) {
        return 0;
    }
    int[] M = new int[target + 1];
    M[0] = 1;
    for (int i = 1; i <= target; i++) {
        for (int num : nums) {
            if (num <= i) {
                M[i] += M[i - num];
            }
        }
    }
    return M[target];
}
```

Time Complexity: O(n)

Space Complexity: O(n)

# 7. LeetCode 198 House Robber (medium)

- 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.
- **Constraints:**
  - `1 <= nums.length <= 100`
  - `0 <= nums[i] <= 400`

## Solution

1. base case
   - M[0] = nums[0]
   - M[1] = Math.max(nums[0], nums[1])
2. induction rule
   - M[i] represents the maximum amount of money you can rob stopping at index i
   - M[i] = Math.max(M[i - 2] + nums[i], M[i - 1])

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

Time Complexity: O(n)

Space Complexity: O(n)

## Solution: space optimized

```
public int rob(int[] nums) {
    if (nums == null || nums.length == 0) {
        return 0;
```

```
    }
    if (nums.length == 1) {
        return nums[0];
    }
    int twoStep = nums[0];
    int oneStep = Math.max(nums[0], nums[1]);
    int result = oneStep;
    for (int i = 2; i < nums.length; i++) {
        result = Math.max(twoStep + nums[i], oneStep);
        twoStep = oneStep;
        oneStep = result;
    }
    return result;
}
```

Time Complexity: O(n)

Space Complexity: O(1)

## 8. LeetCode 213 House Robber II (medium)

- You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. All houses at this place are **arranged in a circle.** That means the first house is the neighbor of the last one. Meanwhile, adjacent houses have a security system 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 = [2,3,2]
  - **Output:** 3
  - **Explanation:** You cannot rob house 1 (money = 2) and then rob house 3 (money = 2), because they are adjacent houses.
- **Example 2:**
  - **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 3:**
  - **Input:** nums = [1,2,3]
  - **Output:** 3
- **Constraints:**
  - `1 <= nums.length <= 100`
  - `0 <= nums[i] <= 1000`

## Solution

- two subproblems:
  1. rob 0 to nums.length - 2
  2. rob 1 to nums.length - 1

```java
public int rob(int[] nums) {
    if (nums == null || nums.length == 0) {
        return 0;
    }
    if (nums.length == 1) {
        return nums[0];
    }
    return Math.max(rob(nums, 0, nums.length - 2), rob(nums, 1, nums.length -1));
}

private int rob(int[] nums, int left, int right) {
    int include = 0, exclude = 0;
    for (int i = left; i <= right; i++) {
        int oneStep = include, twoStep = exclude;
        include = twoStep + nums[i];
        exclude = Math.max(oneStep, twoStep);
    }
    return Math.max(include, exclude);
}
```

Time Complexity: O(n)

Space Complexity: O(1)

## 9. LeetCode 91 Decode Ways (medium)

- A message containing letters from `A-Z` can be **encoded** into numbers using the following mapping:

```
'A' -> "1"
'B' -> "2"
...
'Z' -> "26"
```

- To **decode** an encoded message, all the digits must be grouped then mapped back into letters using the reverse of the mapping above (there may be multiple ways). For example, `"11106"` can be mapped into:
  - `"AAJF"` with the grouping `(1 1 10 6)`
  - `"KJF"` with the grouping `(11 10 6)`
- Note that the grouping `(1 11 06)` is invalid because `"06"` cannot be mapped into `'F'` since `"6"` is different from `"06"`.
- Given a string `s` containing only digits, return *the **number** of ways to **decode** it*.
- The test cases are generated so that the answer fits in a **32-bit** integer.
- **Example 1:**
  - **Input:** s = "12"
  - **Output:** 2
  - **Explanation:** "12" could be decoded as "AB" (1 2) or "L" (12).
- **Example 2:**
  - **Input:** s = "226"
  - **Output:** 3
  - **Explanation:** "226" could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2 2 6).
- **Example 3:**
  - **Input:** s = "06"
  - **Output:** 0

- **Explanation:** "06" cannot be mapped to "F" because of the leading zero ("6" is different from "06").

- **Constraints:**
  - `1 <= s.length <= 100`
  - `s` contains only digits and may contain leading zero(s).

## Solution

```java
public int numDecodings(String s) {
    if (s == null || s.length() == 0) {
        return 0;
    }
    int[] M = new int[s.length() + 1];
    M[0] = 1;
    M[1] = s.charAt(0) != '0' ? 1 : 0;
    for (int i = 2; i <= s.length(); i++) {
        if (s.charAt(i - 1) == '0') {
            if (s.charAt(i - 2) == '1' || s.charAt(i - 2) == '2') {
                M[i] = M[i - 2];
            }
        } else if ((s.charAt(i - 2) == '1' && s.charAt(i - 1) != '0') || (s.charAt(i - 2) == '2' && s.charAt(i - 1) > '0' && s.charAt(i - 1) < '7')) {
            M[i] = M[i - 1] + M[i - 2];
        } else {
            M[i] = M[i - 1];
        }
    }
    return M[s.length()];
}
```

Time Complexity: O(n)

Space Complexity: O(n)

## 10. LeetCode 62 Unique Paths (medium)

- There is a robot on an `m x n` grid. The robot is initially located at the **top-left corner** (i.e., `grid[0][0]`). The robot tries to move to the **bottom-right corner**(i.e., `grid[m - 1][n - 1]`). The robot can only move either down or right at any point in time.
- Given the two integers `m` and `n`, return *the number of possible unique paths that the robot can take to reach the bottom-right corner*.
- The test cases are generated so that the answer will be less than or equal to `2 * 10^9`.
- **Example 1:**
  - 

  - **Input:** m = 3, n = 7
  - **Output:** 28
- **Example 2:**

    - **Input:** m = 3, n = 2
    - **Output:** 3
    - **Explanation:** From the top-left corner, there are a total of 3 ways to reach the bottom-right corner:
        1. Right -> Down -> Down
        2. Down -> Down -> Right
        3. Down -> Right -> Down
- **Constraints:**
    - `1 <= m, n <= 100`

# Solution 1: dynamic programming

| | 1 | 1 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 3 | 6 | 10 | 15 | 21 | 28 |

```java
public int uniquePaths(int m, int n) {
    // assumption: m > 0, n > 0
    int[][] M = new int[m][n];
    for (int[] row : M) {
        Arrays.fill(row, 1);
    }
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            M[i][j] = M[i - 1][j] + M[i][j - 1];
        }
    }
    return M[m - 1][n - 1];
}
```

Time Complexity: O(mn)

Space Complexity: O(mn)

# Solution 2: math `combination(m + n, m) = (m + n)! / (m! * n!)`

```java
public int uniquePaths(int m, int n) {
    if (m == 1 || n == 1) {
        return 1;
    }
    if (m < n) {
        return uniquePaths(n, m);
    }
    m--;
    n--;
    long result = 1;
    for (int i = m + 1, j = 1; i <= m + n; i++, j++) {
        result *= i;
        result /= j;
    }
    return (int)result;
```

```
}
```

Time Complexity: O(min(m, n))

Space Complexity: O(1)

# 11. LeetCode 55 Jump Game (medium)

- You are given an integer array `nums`. You are initially positioned at the array's **first index**, and each element in the array represents your maximum jump length at that position.
- Return `true` *if you can reach the last index, or* `false` *otherwise*.
- **Example 1:**
  - **Input:** nums = [2,3,1,1,4]
  - **Output:** true
  - **Explanation:** Jump 1 step from index 0 to 1, then 3 steps to the last index.
- **Example 2:**
  - **Input:** nums = [3,2,1,0,4]
  - **Output:** false
  - **Explanation:** You will always arrive at index 3 no matter what. Its maximum jump length is 0, which makes it impossible to reach the last index.
- **Constraints:**
  - `1 <= nums.length <= 10^4`
  - `0 <= nums[i] <= 10^5`

## Solution 1: dynamic programming

1. base case: M[n - 1] = true
2. induction rule
   - M[i] represents whether we could reach the target from index i
   - M[i] = true, if there exists a j, where M[j] == true AND j <= i + input[i]
   - M[i] = false, otherwise

```
public boolean canJump(int[] nums) {
    if (nums == null || nums.length == 0) {
        return false;
    }
    if (nums.length == 1) {
        return true;
    }
    boolean[] M = new boolean[nums.length];
    for (int i = nums.length - 2; i >= 0; i--) {
        if (i + nums[i] >= nums.length - 1) {
            M[i] = true;
        } else {
            for (int j = nums[i]; j >= 1; j--) {
                if (M[j + i]) {
                    M[i] = true;
                    break;
                }
            }
        }
    }
```

```
    return M[0];
}
```

Time Complexity: O(n^2)

Space Complexity: O(n)

# Solution 2: greedy

```
public boolean canJump(int[] nums) {
    if (nums == null || nums.length == 0) {
        return false;
    }
    if (nums.length == 1) {
        return true;
    }
    int lastPosition = nums.length - 1;
    for (int i = nums.length - 1; i >= 0; i--) {
        if (i + nums[i] >= lastPosition) {
            lastPosition = i;
        }
    }
    return lastPosition == 0;
}
```

Time Complexity: O(n)

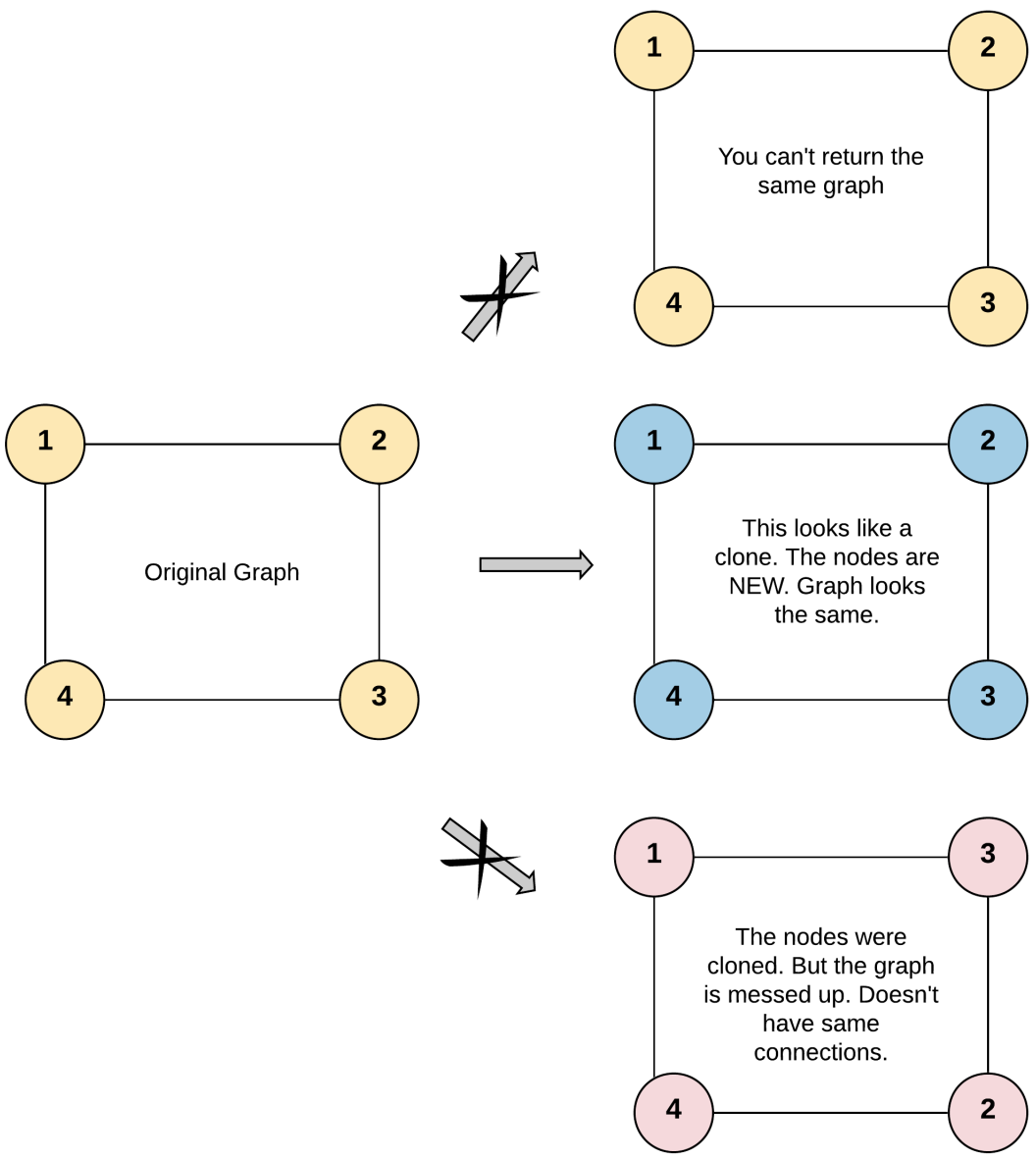
Space Complexity: O(1)

# Graph

## 1. LeetCode 133 Clone Graph (medium)

- Given a reference of a node in a **connected** undirected graph.
- Return a **deep copy** (clone) of the graph.
- Each node in the graph contains a value (`int`) and a list (`List[Node]`) of its neighbors.

```
class Node {
    public int val;
    public List<Node> neighbors;
}
```

- **Test case format:**
- For simplicity, each node's value is the same as the node's index (1-indexed). For example, the first node with `val == 1`, the second node with `val == 2`, and so on. The graph is represented in the test case using an adjacency list.
- **An adjacency list** is a collection of unordered **lists** used to represent a finite graph. Each list describes the set of neighbors of a node in the graph.
- The given node will always be the first node with `val = 1`. You must return the **copy of the given node** as a reference to the cloned graph.
- **Example 1:**
  - 

  - **Input:** adjList = `[[2,4],[1,3],[2,4],[1,3]]`
  - **Output:** `[[2,4],[1,3],[2,4],[1,3]]`
  - **Explanation:** There are 4 nodes in the graph.
    - 1st node (val = 1)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).

    - 2nd node (val = 2)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).
    - 3rd node (val = 3)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).
    - 4th node (val = 4)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).
- **Example 2:**


    - **Input:** adjList = `[[]]`
    - **Output:** `[[]]`
    - **Explanation:** Note that the input contains one empty list. The graph consists of only one node with val = 1 and it does not have any neighbors.
- **Example 3:**
    - **Input:** adjList = `[]`
    - **Output:** `[]`
    - **Explanation:** This an empty graph, it does not have any nodes.
- **Constraints:**
    - The number of nodes in the graph is in the range `[0, 100]`.
    - `1 <= Node.val <= 100`
    - `Node.val` is unique for each node.
    - There are no repeated edges and no self-loops in the graph.
    - The Graph is connected and all nodes can be visited starting from the given node.

## Solution 1: dfs

```java
public Node cloneGraph(Node node) {
    if (node == null) {
        return null;
    }
    Map<Node, Node> map = new HashMap<>();
    Node cloneNode = new Node(node.val);
    map.put(node, cloneNode);
    for (Node neighbor : node.neighbors) {
        if (!map.containsKey(neighbor)) {
            map.put(neighbor, new Node(neighbor.val));
            DFS(neighbor, map);
        }
        cloneNode.neighbors.add(map.get(neighbor));
    }
    return cloneNode;
}

private void DFS(Node node, Map<Node, Node> map) {
    Node copy = map.get(node);
    for (Node neighbor : node.neighbors) {
        if (!map.containsKey(neighbor)) {
            map.put(neighbor, new Node(neighbor.val));
            DFS(neighbor, map);
        }
        copy.neighbors.add(map.get(neighbor));
    }
```

```
}
```

Time Complexity: O(node + edge)

Space Complexity: O(node)

## Solution 2: bfs

```
public Node cloneGraph(Node node) {
    if (node == null) {
        return null;
    }
    Map<Node, Node> map = new HashMap<>();
    Queue<Node> queue = new ArrayDeque<>();
    Node cloneNode = new Node(node.val);
    map.put(node, cloneNode);
    queue.offer(node);
    while (!queue.isEmpty()) {
        Node old = queue.poll();
        for (Node neighbor : old.neighbors) {
            if (!map.containsKey(neighbor)) {
                map.put(neighbor, new Node(neighbor.val));
                queue.offer(neighbor);
            }
            map.get(old).neighbors.add(map.get(neighbor));
        }
    }
    return cloneNode;
}
```

Time Complexity: O(node + edge)

Space Complexity: O(node)

# 2. LeetCode 207 Course Schedule (medium)

- There are a total of `numCourses` courses you have to take, labeled from `0` to `numCourses - 1`. You are given an array `prerequisites` where `prerequisites[i] = [ai, bi]` indicates that you **must** take course `bi` first if you want to take course `ai`.
  - For example, the pair `[0, 1]`, indicates that to take course `0` you have to first take course `1`.
- Return `true` if you can finish all courses. Otherwise, return `false`.
- **Example 1:**
  - **Input:** numCourses = 2, prerequisites = `[[1,0]]`
  - **Output:** true
  - **Explanation:** There are a total of 2 courses to take.
    - To take course 1 you should have finished course 0. So it is possible.
- **Example 2:**
  - **Input:** numCourses = 2, prerequisites = `[[1,0],[0,1]]`
  - **Output:** false
  - **Explanation:** There are a total of 2 courses to take.
    - To take course 1 you should have finished course 0, and to take course 0 you should also have

finished course 1. So it is impossible.

- **Constraints:**
  - `1 <= numCourses <= 2000`
  - `0 <= prerequisites.length <= 5000`
  - `prerequisites[i].length == 2`
  - `0 <= ai, bi < numCourses`
  - All the pairs prerequisites[i] are **unique**.

DAG (directed acyclic graph) 有向无环图 -> topological ordering 拓扑排序

# Solution 1: dfs + memo (otherwise time limit exceeded)

```java
public boolean canFinish(int numCourses, int[][] prerequisites) {
    if (numCourses <= 0) {
        return false;
    }
    if (prerequisites == null || prerequisites.length == 0) {
        return true;
    }
    List[] graph = new List[numCourses];
    for (int i = 0; i < numCourses; i++) {
        graph[i] = new ArrayList<Integer>();
    }
    for (int i = 0; i < prerequisites.length; i++) {
        graph[prerequisites[i][1]].add(prerequisites[i][0]);
    }
    boolean[] visited = new boolean[numCourses];
    boolean[] memo = new boolean[numCourses];
    for (int i = 0; i < numCourses; i++) {
        if (!DFS(graph, visited, i, memo)) {
            return false;
        }
    }
    return true;
}

private boolean DFS(List[] graph, boolean[] visited, int course, boolean[] memo) {
    if (visited[course]) {
        // cycle
        return false;
    }
    if (memo[course]) {
        return true;
    }
    visited[course] = true;
    for (int i = 0; i < graph[course].size(); i++) {
        if (!DFS(graph, visited, (int)graph[course].get(i), memo)) {
            return false;
        }
    }
    visited[course] = false;
    memo[course] = true;
    return true;
}
```

Time Complexity: O(node + edge)

Space Complexity: O(node + edge)

## Solution 2: bfs

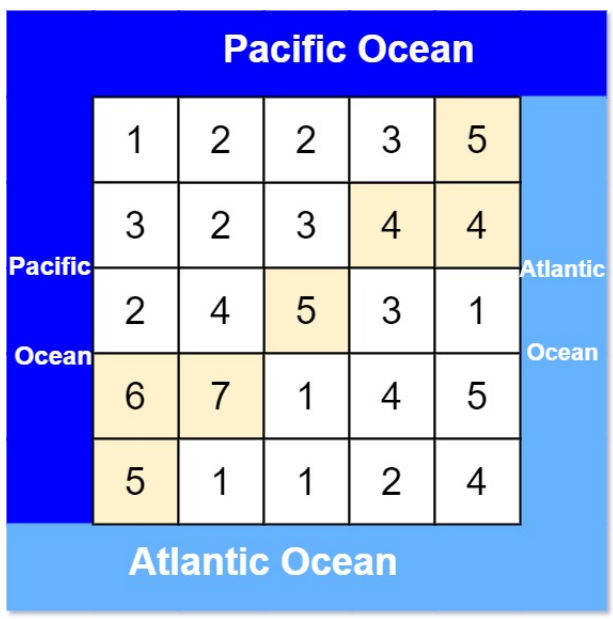
```java
public boolean canFinish(int numCourses, int[][] prerequisites) {
    if (numCourses <= 0) {
        return false;
    }
    if (prerequisites == null || prerequisites.length == 0) {
        return true;
    }
    List[] graph = new List[numCourses];
    for (int i = 0; i < numCourses; i++) {
        graph[i] = new ArrayList<Integer>();
    }
    int[] degree = new int[numCourses];
    for (int i = 0; i < prerequisites.length; i++) {
        graph[prerequisites[i][0]].add(prerequisites[i][1]);
        degree[prerequisites[i][1]]++;
    }
    Queue<Integer> queue = new ArrayDeque<>();
    for (int i = 0; i < numCourses; i++) {
        if (degree[i] == 0) {
            queue.offer(i);
        }
    }
    int count = 0;
    while (!queue.isEmpty()) {
        int course = (int)queue.poll();
        count++;
        for (int i = 0; i < graph[course].size(); i++) {
            int next = (int)graph[course].get(i);
            degree[next]--;
            if (degree[next] == 0) {
                queue.offer(next);
            }
        }
    }
    return count == numCourses;
}
```

Time Complexity: O(node + edge)

Space Complexity: O(node + edge)

# 3. LeetCode 417 Pacific Atlantic Water Flow (medium)

- There is an `m x n` rectangular island that borders both the **Pacific Ocean** and **Atlantic Ocean**.
  The **Pacific Ocean** touches the island's left and top edges, and the **Atlantic Ocean** touches the island's right and bottom edges.
- The island is partitioned into a grid of square cells. You are given an `m x n` integer matrix `heights` where `heights[r][c]` represents the **height above sea level** of the cell at coordinate `(r, c)`.

- The island receives a lot of rain, and the rain water can flow to neighboring cells directly north, south, east, and west if the neighboring cell's height is **less than or equal to** the current cell's height. Water can flow from any cell adjacent to an ocean into the ocean.
- Return *a **2D list** of grid coordinates* `result` *where* `result[i] = [ri, ci]` *denotes that rain water can flow from cell* `(ri, ci)` *to **both** the Pacific and Atlantic oceans*.
- **Example 1:**
  - 

  - **Input:** heights = `[[1,2,2,3,5],[3,2,3,4,4],[2,4,5,3,1],[6,7,1,4,5],[5,1,1,2,4]]`
  - **Output:** `[[0,4],[1,3],[1,4],[2,2],[3,0],[3,1],[4,0]]`
- **Example 2:**
  - **Input:** heights = `[[2,1],[1,2]]`
  - **Output:** `[[0,0],[0,1],[1,0],[1,1]]`
- **Constraints:**
  - `m == heights.length`
  - `n == heights[r].length`
  - `1 <= m, n <= 200`
  - `0 <= heights[r][c] <= 10^5`

## Solution: backtracking

```java
public List<List<Integer>> pacificAtlantic(int[][] heights) {
    List<List<Integer>> result = new ArrayList<>();
    if (heights == null || heights.length == 0) {
        return result;
    }
    int m = heights.length, n = heights[0].length;
    byte[][] dp = new byte[m][n];
    for (int i = 0; i < m; i++) {
        helper(i, 0, 1, heights[i][0], heights, dp, result);
        helper(i, n - 1, 2, heights[i][n - 1], heights, dp, result);
    }
    for (int j = 0; j < n; j++) {
        helper(0, j, 1, heights[0][j], heights, dp, result);
        helper(m - 1, j, 2, heights[m - 1][j], heights, dp, result);
    }
    return result;
}

private void helper(int i, int j, int ocean, int h, int[][] heights, byte[][] dp,
List<List<Integer>> result) {
```

```
20      if (i < 0 || i >= heights.length || j < 0 || j >= heights[0].length || (dp[i][j] &
ocean) > 0 || heights[i][j] < h) {
21          return;
22      }
23      dp[i][j] += ocean;
24      if (dp[i][j] == 3) {
25          result.add(Arrays.asList(i, j));
26      }
27      helper(i - 1, j, ocean, heights[i][j], heights, dp, result);
28      helper(i + 1, j, ocean, heights[i][j], heights, dp, result);
29      helper(i, j - 1, ocean, heights[i][j], heights, dp, result);
30      helper(i, j + 1, ocean, heights[i][j], heights, dp, result);
31  }
```

Time Complexity: O(mn)

Space Complexity: O(mn)

# 4. LeetCode 200 Number of Islands (medium)

- Given an `m x n` 2D binary grid `grid` which represents a map of `'1'`s (land) and `'0'`s (water), return *the number of islands*.
- An **island** is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.
- **Example 1:**
  - **Input:**

```
grid = [
  ["1","1","1","1","0"],
  ["1","1","0","1","0"],
  ["1","1","0","0","0"],
  ["0","0","0","0","0"]
]
```

  - **Output:** 1
- **Example 2:**
  - **Input:**

```
grid = [
  ["1","1","0","0","0"],
  ["1","1","0","0","0"],
  ["0","0","1","0","0"],
  ["0","0","0","1","1"]
]
```

  **Output:** 3
- **Constraints:**
  - `m == grid.length`
  - `n == grid[i].length`
  - `1 <= m, n <= 300`
  - `grid[i][j]` is `'0'` or `'1'`.

# Solution 1: dfs

```java
public int numIslands(char[][] grid) {
    if (grid == null || grid.length == 0) {
        return 0;
    }
    int result = 0;
    for (int i = 0; i < grid.length; i++) {
        for (int j = 0; j < grid[0].length; j++) {
            if (grid[i][j] == '1') {
                result++;
                dfs(grid, i, j);
            }
        }
    }
    return result;
}

private void dfs(char[][] grid, int i, int j) {
    if (i < 0 || j < 0 || i >= grid.length || j >= grid[0].length || grid[i][j] == '0') {
        return;
    }
    grid[i][j] = '0';
    dfs(grid, i - 1, j);
    dfs(grid, i + 1, j);
    dfs(grid, i, j - 1);
    dfs(grid, i, j + 1);
}
```

Time Complexity: O(mn)

Space Complexity: O(mn)

# Solution 2: bfs

```java
public int numIslands(char[][] grid) {
    if (grid == null || grid.length == 0) {
        return 0;
    }
    int result = 0, len = grid[0].length;
    for (int i = 0; i < grid.length; i++) {
        for (int j = 0; j < grid[0].length; j++) {
            if (grid[i][j] == '1') {
                result++;
                grid[i][j] = '0';
                Queue<Integer> q = new ArrayDeque<>();
                q.offer(i * len + j);
                while (!q.isEmpty()) {
                    int id = q.poll();
                    int row = id / len;
                    int col = id % len;
                    if (row > 0 && grid[row - 1][col] == '1') {
                        q.offer((row - 1) * len + col);
                        grid[row - 1][col] = '0';
                    }
                    if (row + 1 < grid.length && grid[row + 1][col] == '1') {
```

```
                        q.offer((row + 1) * len + col);
                        grid[row + 1][col] = '0';
                    }
                    if (col > 0 && grid[row][col - 1] == '1') {
                        q.offer(row * len + col - 1);
                        grid[row][col - 1] = '0';
                    }
                    if (col + 1 < grid[0].length && grid[row][col + 1] == '1') {
                        q.offer(row * len + col + 1);
                        grid[row][col + 1] = '0';
                    }
                }
            }
        }
    }
    return result;
}
```

Time Complexity: O(mn)

Space Complexity: O(min(m, n))

# 5. LeetCode 128 Longest Consecutive Sequence (medium)

- Given an unsorted array of integers `nums`, return *the length of the longest consecutive elements sequence.*
- You must write an algorithm that runs in `O(n)` time.
- **Example 1:**
  - **Input:** nums = [100,4,200,1,3,2]
  - **Output:** 4
  - **Explanation:** The longest consecutive elements sequence is `[1, 2, 3, 4]`. Therefore its length is 4.
- **Example 2:**
  - **Input:** nums = [0,3,7,2,5,8,4,6,0,1]
  - **Output:** 9
- **Constraints:**
  - `0 <= nums.length <= 10^5`
  - `-10^9 <= nums[i] <= 10^9`

## Solution

```
public int longestConsecutive(int[] nums) {
    if (nums == null || nums.length == 0) {
        return 0;
    }
    Set<Integer> set = new HashSet<>();
    for (int num : nums) {
        set.add(num);
    }
    int result = 0;
    for (int num : set) {
        if (!set.contains(num - 1)) {
            int current = num, temp = 1;
            while (set.contains(current + 1)) {
                temp++;
```

```
                current++;
            }
            result = Math.max(result, temp);
        }
    }
    return result;
}
```

Time Complexity: O(n)

Space Complexity: O(n)

# 6. LeetCode 269 Alien Dictionary (hard)

- There is a new alien language that uses the English alphabet. However, the order among the letters is unknown to you.
- You are given a list of strings `words` from the alien language's dictionary, where the strings in `words` are **sorted lexicographically** by the rules of this new language.
- Return *a string of the unique letters in the new alien language sorted in **lexicographically increasing order** by the new language's rules. If there is no solution, return `""`. If there are multiple solutions, return **any of them***.
- A string `s` is **lexicographically smaller** than a string `t` if at the first letter where they differ, the letter in `s` comes before the letter in `t` in the alien language. If the first `min(s.length, t.length)` letters are the same, then `s` is smaller if and only if `s.length < t.length`.
- **Example 1:**
  - **Input:** words = ["wrt","wrf","er","ett","rftt"]
  - **Output:** "wertf"
- **Example 2:**
  - **Input:** words = ["z","x"]
  - **Output:** "zx"
- **Example 3:**
  - **Input:** words = ["z","x","z"]
  - **Output:** ""
  - **Explanation:** The order is invalid, so return `""`.
- **Constraints:**
  - `1 <= words.length <= 100`
  - `1 <= words[i].length <= 100`
  - `words[i]` consists of only lowercase English letters.

## Solution 1: dfs

```
public String alienOrder(String[] words) {
    if (words == null || words.length == 0) {
        return "";
    }
    Map<Character, List<Character>> graph = new HashMap<>();
    for (String word : words) {
        for (char c : word.toCharArray()) {
            graph.put(c, new ArrayList<>());
        }
    }
```

```
    for (int i = 0; i < words.length - 1; i++) {
        String word1 = words[i];
        String word2 = words[i + 1];
        if (word1.length() > word2.length() && word1.startsWith(word2)) {
            return "";
        }
        for (int j = 0; j < Math.min(word1.length(), word2.length()); j++) {
            if (word1.charAt(j) != word2.charAt(j)) {
                graph.get(word2.charAt(j)).add(word1.charAt(j));
                break;
            }
        }
    }
    Map<Character, Boolean> visited = new HashMap<>();
    StringBuilder sb = new StringBuilder();
    for (Character c : graph.keySet()) {
        if (!dfs(graph, c, visited, sb)) {
            return "";
        }
    }
    return sb.toString();
}

private boolean dfs(Map<Character, List<Character>> graph, Character c, Map<Character,
Boolean> visited, StringBuilder sb) {
    if (visited.containsKey(c)) {
        return visited.get(c);
    }
    visited.put(c, false);
    for (Character next : graph.get(c)) {
        if (!dfs(graph, next, visited, sb)) {
            return false;
        }
    }
    visited.put(c, true);
    sb.append(c);
    return true;
}
```

Time Complexity: O(total length of all words)

Space Complexity: O(1)

## Solution 2: bfs

```
public String alienOrder(String[] words) {
    if (words == null || words.length == 0) {
        return "";
    }
    Map<Character, List<Character>> graph = new HashMap<>();
    Map<Character, Integer> degree = new HashMap<>();
    for (String word : words) {
        for (char c : word.toCharArray()) {
            graph.put(c, new ArrayList<>());
            degree.put(c, 0);
        }
    }
```

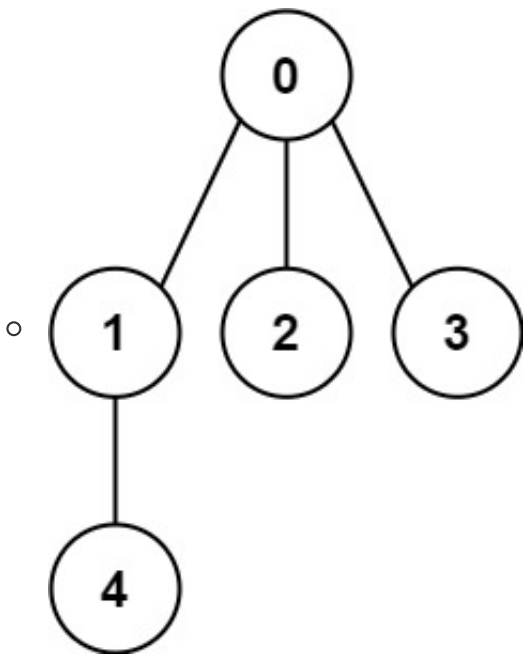
```
    for (int i = 0; i < words.length - 1; i++) {
        String word1 = words[i];
        String word2 = words[i + 1];
        if (word1.length() > word2.length() && word1.startsWith(word2)) {
            return "";
        }
        for (int j = 0; j < Math.min(word1.length(), word2.length()); j++) {
            if (word1.charAt(j) != word2.charAt(j)) {
                graph.get(word1.charAt(j)).add(word2.charAt(j));
                degree.put(word2.charAt(j), degree.get(word2.charAt(j)) + 1);
                break;
            }
        }
    }
    Queue<Character> queue = new ArrayDeque<>();
    for (Character c : degree.keySet()) {
        if (degree.get(c).equals(0)) {
            queue.offer(c);
        }
    }
    StringBuilder sb = new StringBuilder();
    while (!queue.isEmpty()) {
        Character c = queue.poll();
        sb.append(c);
        for (Character next : graph.get(c)) {
            degree.put(next, degree.get(next) - 1);
            if (degree.get(next).equals(0)) {
                queue.offer(next);
            }
        }
    }
    if (sb.length() != degree.size()) {
        return "";
    }
    return sb.toString();
}
```

Time Complexity: O(total length of all words)

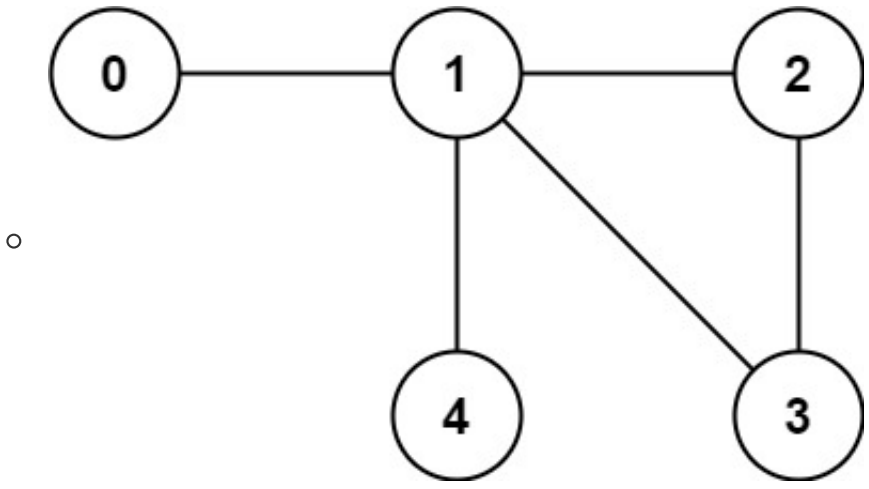
Space Complexity: O(1)

# 7. LeetCode 261 Graph Valid Tree (medium)

- You have a graph of `n` nodes labeled from `0` to `n - 1`. You are given an integer n and a list of `edges` where `edges[i] = [ai, bi]` indicates that there is an undirected edge between nodes `ai` and `bi` in the graph.
- Return `true` *if the edges of the given graph make up a valid tree, and* `false` *otherwise*.
- **Example 1:**




    - **Input:** n = 5, edges = `[[0,1],[0,2],[0,3],[1,4]]`
    - **Output:** true
- **Example 2:**




    - **Input:** n = 5, edges = `[[0,1],[1,2],[2,3],[1,3],[1,4]]`
    - **Output:** false
- **Constraints:**
    - `1 <= n <= 2000`
    - `0 <= edges.length <= 5000`
    - `edges[i].length == 2`
    - `0 <= ai, bi < n`
    - `ai != bi`
    - There are no self-loops or repeated edges.

## Solution 1: dfs

```java
public boolean validTree(int n, int[][] edges) {
    if (edges == null || edges.length != n - 1) {
        return false;
    }
    Map<Integer, List<Integer>> graph = new HashMap<>();
    for (int i = 0; i < n; i++) {
        graph.put(i, new ArrayList<>());
    }
    for (int[] edge : edges) {
        graph.get(edge[0]).add(edge[1]);
        graph.get(edge[1]).add(edge[0]);
    }
    Set<Integer> visited = new HashSet<>();
    dfs(0, graph, visited);
    return visited.size() == n;
}

```

```java
private void dfs(int node, Map<Integer, List<Integer>> graph, Set<Integer> visited) {
    if (visited.contains(node)) {
        return;
    }
    visited.add(node);
    for (int neighbor : graph.get(node)) {
        dfs(neighbor, graph, visited);
    }
}
```

Time Complexity: O(n)

Space Complexity: O(n)

## Solution 2: bfs

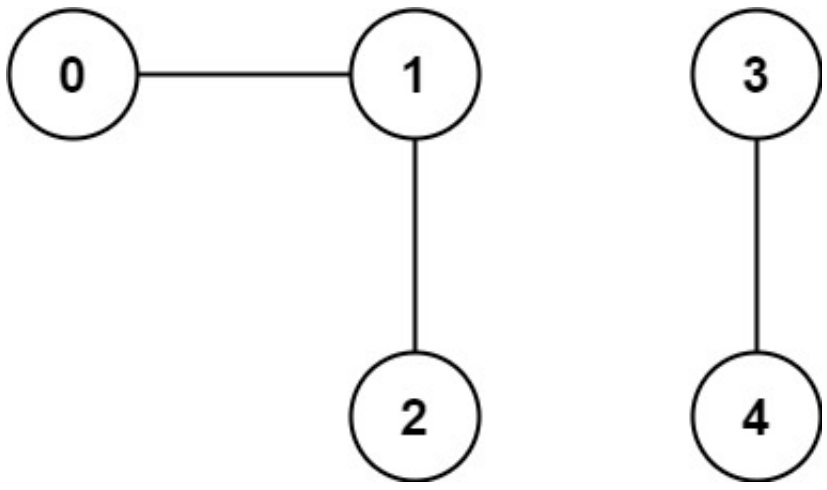
```java
public boolean validTree(int n, int[][] edges) {
    if (edges == null || edges.length != n - 1) {
        return false;
    }
    Map<Integer, List<Integer>> graph = new HashMap<>();
    for (int i = 0; i < n; i++) {
        graph.put(i, new ArrayList<>());
    }
    for (int[] edge : edges) {
        graph.get(edge[0]).add(edge[1]);
        graph.get(edge[1]).add(edge[0]);
    }
    Queue<Integer> queue = new ArrayDeque<>();
    Set<Integer> visited = new HashSet<>();
    queue.offer(0);
    visited.add(0);
    while (!queue.isEmpty()) {
        int node = queue.poll();
        for (int neighbor : graph.get(node)) {
            if (visited.contains(neighbor)) {
                continue;
            }
            queue.offer(neighbor);
            visited.add(neighbor);
        }
    }
    return visited.size() == n;
}
```

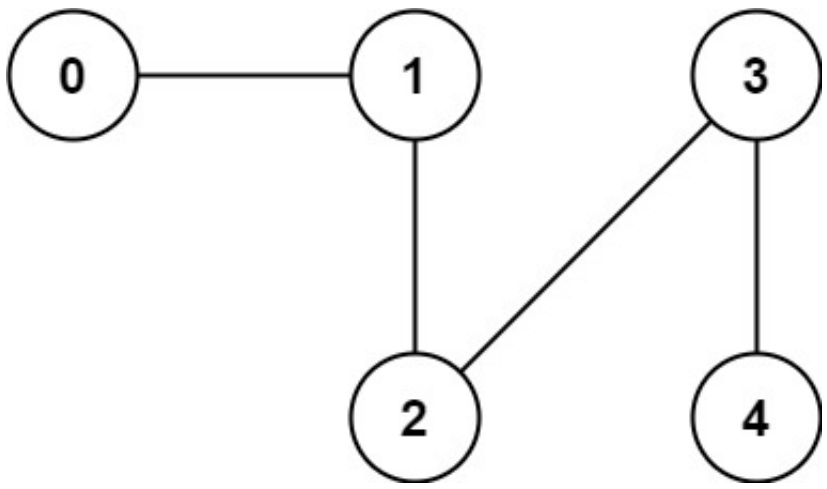
Time Complexity: O(n)

Space Complexity: O(n)

# 8. LeetCode 323 Number of Connected Components in an Undirected Graph (medium)

- You have a graph of `n` nodes. You are given an integer `n` and an array `edges` where `edges[i] = [ai, bi]` indicates that there is an edge between `ai` and `bi` in the graph.
- Return *the number of connected components in the graph*.

- **Example 1:**
  - 

  - **Input:** n = 5, edges = `[[0,1],[1,2],[3,4]]`
  - **Output:** 2
- **Example 2:**
  - 

  - **Input:** n = 5, edges = `[[0,1],[1,2],[2,3],[3,4]]`
  - **Output:** 1
- **Constraints:**
  - `1 <= n <= 2000`
  - `1 <= edges.length <= 5000`
  - `edges[i].length == 2`
  - `0 <= ai <= bi < n`
  - `ai != bi`
  - There are no repeated edges.

## Solution: dfs

```java
public int countComponents(int n, int[][] edges) {
    Map<Integer, List<Integer>> graph = new HashMap<>();
    for (int i = 0; i < n; i++) {
        graph.put(i, new ArrayList<>());
    }
    for (int[] edge : edges) {
        graph.get(edge[0]).add(edge[1]);
        graph.get(edge[1]).add(edge[0]);
    }
    Set<Integer> visited = new HashSet<>();
    int result = 0;
    for (int i = 0; i < n; i++) {
        if (!visited.contains(i)) {
            result++;
            dfs(i, graph, visited);
        }
    }
    return result;
}
```

```
20
21 private void dfs(int node, Map<Integer, List<Integer>> graph, Set<Integer> visited) {
22     visited.add(node);
23     for (int neighbor : graph.get(node)) {
24         if (!visited.contains(neighbor)) {
25             dfs(neighbor, graph, visited);
26         }
27     }
28 }
```

Time Complexity: O(node + edge)

Space Complexity: O(node + edge)

# Heap

```java
PriorityQueue<Integer> minHeap = new PriorityQueue<>();
PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());
```

# 1. LeetCode 23 Merge K Sorted Lists (hard)

- You are given an array of `k` linked-lists `lists`, each linked-list is sorted in ascending order.
- *Merge all the linked-lists into one sorted linked-list and return it.*
- **Example 1:**
  - **Input:** lists = `[[1,4,5],[1,3,4],[2,6]]`
  - **Output:** [1,1,2,3,4,4,5,6]
- **Example 2:**
  - **Input:** lists = `[]`
  - **Output:** []
- **Example 3:**
  - **Input:** lists = `[[]]`
  - **Output:** []
- **Constraints:**
  - `k == lists.length`
  - `0 <= k <= 10^4`
  - `0 <= lists[i].length <= 500`
  - `-10^4 <= lists[i][j] <= 10^4`
  - `lists[i]` is sorted in **ascending order**.
  - The sum of `lists[i].length` will not exceed `10^4`.

## Solution: priority queue (min heap)

```java
public ListNode mergeKLists(ListNode[] lists) {
    if (lists == null || lists.length == 0) {
        return null;
    }
    PriorityQueue<ListNode> pq = new PriorityQueue<>(lists.length, (n1, n2) -> {
        if (n1.val == n2.val) {
            return 0;
        }
        return n1.val < n2.val ? -1 : 1;
    });
    ListNode dummy = new ListNode(0);
    ListNode tail = dummy;
    for (ListNode node : lists) {
        if (node != null) {
            pq.offer(node);
        }
    }
    while (!pq.isEmpty()) {
        tail.next = pq.poll();
        tail = tail.next;
        if (tail.next != null) {
            pq.offer(tail.next);
```

```
            }
        }
        return dummy.next;
    }
```

Time Complexity: O(nlogk)

Space Complexity: O(k)

# 2. LeetCode 347 Top K Frequent Elements (medium)

- Given an integer array `nums` and an integer `k`, return *the* `k` *most frequent elements*. You may return the answer in **any order**.
- **Example 1:**
  - **Input:** nums = [1,1,1,2,2,3], k = 2
  - **Output:** [1,2]
- **Example 2:**
  - **Input:** nums = [1], k = 1
  - **Output:** [1]
- **Constraints:**
  - `1 <= nums.length <= 10^5`
  - `k` is in the range `[1, the number of unique elements in the array]`.
  - It is **guaranteed** that the answer is **unique**.
- **Follow up:** Your algorithm's time complexity must be better than `O(nlogn)`, where n is the array's size.

## Solution 1: hashmap + heap

```
public int[] topKFrequent(int[] nums, int k) {
    if (nums == null || nums.length == 0) {
        return null;
    }
    if (nums.length <= k) {
        return nums;
    }
    Map<Integer, Integer> map = new HashMap<>();
    for (int num : nums) {
        map.put(num, map.getOrDefault(num, 0) + 1);
    }
    PriorityQueue<Map.Entry<Integer, Integer>> minHeap = new PriorityQueue<>(k, new
Comparator<Map.Entry<Integer, Integer>>() {
        @Override
        public int compare(Map.Entry<Integer, Integer> e1, Map.Entry<Integer, Integer>
e2) {
            return e1.getValue().compareTo(e2.getValue());
        }
    });
    for (Map.Entry<Integer, Integer> entry : map.entrySet()) {
        if (minHeap.size() < k) {
            minHeap.offer(entry);
        } else if (entry.getValue() > minHeap.peek().getValue()) {
            minHeap.poll();
            minHeap.offer(entry);
        }
    }
```

```
    int[] result = new int[minHeap.size()];
    for (int i = minHeap.size() - 1; i >= 0; i--) {
        result[i] = minHeap.poll().getKey();
    }
    return result;
}
```

Time Complexity: O(nlogk)

Space Complexity: O(n)

## Solution 2: quick select

```
public int[] topKFrequent(int[] nums, int k) {
    if (nums == null || nums.length == 0) {
        return null;
    }
    if (nums.length <= k) {
        return nums;
    }
    Map<Integer, Integer> map = new HashMap<>();
    for (int num : nums) {
        map.put(num, map.getOrDefault(num, 0) + 1);
    }
    int n = map.size();
    int[] unique = new int[n];
    int i = 0;
    for (int num : map.keySet()) {
        unique[i] = num;
        i++;
    }
    quickselect(0, n - 1, n - k, map, unique);
    return Arrays.copyOfRange(unique, n - k, n);
}

private void quickselect(int left, int right, int k_smallest, Map<Integer, Integer> map,
int[] unique) {
    if (left == right) {
        return;
    }
    int pivot = left + new Random().nextInt(right - left);
    pivot = partition(left, right, pivot, map, unique);
    if (pivot == k_smallest) {
        return;
    } else if (pivot < k_smallest) {
        quickselect(pivot + 1, right, k_smallest, map, unique);
    } else {
        quickselect(left, pivot - 1, k_smallest, map, unique);
    }
}

private int partition(int left, int right, int pivot, Map<Integer, Integer> map, int[]
unique) {
    int frequency = map.get(unique[pivot]);
    swap(unique, pivot, right);
    int index = left;
    for (int i = left; i < right; i++) {
```

```
        if (map.get(unique[i]) < frequency) {
            swap(unique, index, i);
            index++;
        }
    }
    swap(unique, index, right);
    return index;
}

private void swap(int[] array, int x, int y) {
    int temp = array[x];
    array[x] = array[y];
    array[y] = temp;
}
```

Time Complexity: O(n)

Space Complexity: O(n)

## 3. LeetCode 295 Find Median from Data Stream

- The **median** is the middle value in an ordered integer list. If the size of the list is even, there is no middle value and the median is the mean of the two middle values.
  - For example, for `arr = [2,3,4]`, the median is `3`.
  - For example, for `arr = [2,3]`, the median is `(2 + 3) / 2 = 2.5`.
- Implement the MedianFinder class:
  - `MedianFinder()` initializes the `MedianFinder` object.
  - `void addNum(int num)` adds the integer `num` from the data stream to the data structure.
  - `double findMedian()` returns the median of all elements so far. Answers within `10-5` of the actual answer will be accepted.
- **Example 1:**
  - **Input**
    - ["MedianFinder", "addNum", "addNum", "findMedian", "addNum", "findMedian"]
    - `[[], [1], [2], [], [3], []]`
  - **Output**
    - [null, null, null, 1.5, null, 2.0]
  - **Explanation**

```
MedianFinder medianFinder = new MedianFinder();
medianFinder.addNum(1);    // arr = [1]
medianFinder.addNum(2);    // arr = [1, 2]
medianFinder.findMedian(); // return 1.5 (i.e., (1 + 2) / 2)
medianFinder.addNum(3);    // arr[1, 2, 3]
medianFinder.findMedian(); // return 2.0
```

- **Constraints:**
  - `-10^5 <= num <= 10^5`
  - There will be at least one element in the data structure before calling `findMedian`.
  - At most `5 * 10^4` calls will be made to `addNum` and `findMedian`.
- **Follow up:**
  - If all integer numbers from the stream are in the range `[0, 100]`, how would you optimize your

solution?

- If `99%` of all integer numbers from the stream are in the range `[0, 100]`, how would you optimize your solution?

# Solution

```java
class MedianFinder {
    PriorityQueue<Integer> minHeap;
    PriorityQueue<Integer> maxHeap;

    public MedianFinder() {
        minHeap = new PriorityQueue<>();
        maxHeap = new PriorityQueue<>(Collections.reverseOrder());
    }

    public void addNum(int num) {
        maxHeap.offer(num);
        minHeap.offer(maxHeap.poll());
        if (minHeap.size() > maxHeap.size()) {
            maxHeap.offer(minHeap.poll());
        }
        // Time Complexity: O(logn)
    }

    public double findMedian() {
        if (minHeap.size() == maxHeap.size()) {
            return (double) (maxHeap.peek() + minHeap.peek()) * 0.5;
        } else {
            return maxHeap.peek();
        }
        // Time Complexity: O(1)
    }
}
```

# Interval

- `Arrays.sort()` in Java:
  - Time Complexity: O(nlogn)
  - Space Complexity: O(logn)

## 1. LeetCode 57 Insert Interval (medium)

- You are given an array of non-overlapping intervals `intervals` where `intervals[i] = [starti, endi]` represent the start and the end of the `ith` interval and `intervals` is sorted in ascending order by `starti`. You are also given an interval `newInterval = [start, end]` that represents the start and end of another interval.
- Insert `newInterval` into `intervals` such that `intervals` is still sorted in ascending order by `starti` and `intervals` still does not have any overlapping intervals (merge overlapping intervals if necessary).
- Return `intervals` *after the insertion*.
- **Example 1:**
  - **Input:** intervals = `[[1,3],[6,9]]`, newInterval = [2,5]
  - **Output:** `[[1,5],[6,9]]`
- **Example 2:**
  - **Input:** intervals = `[[1,2],[3,5],[6,7],[8,10],[12,16]]`, newInterval = [4,8]
  - **Output:** `[[1,2],[3,10],[12,16]]`
  - **Explanation:** Because the new interval [4,8] overlaps with [3,5],[6,7],[8,10].
- **Constraints:**
  - `0 <= intervals.length <= 10^4`
  - `intervals[i].length == 2`
  - `0 <= starti <= endi <= 10^5`
  - `intervals` is sorted by `starti` in **ascending** order.
  - `newInterval.length == 2`
  - `0 <= start <= end <= 10^5`

### Solution

```java
public int[][] insert(int[][] intervals, int[] newInterval) {
    if (intervals == null || newInterval == null || newInterval.length == 0) {
        return intervals;
    }
    List<int[]> result = new ArrayList<>();
    int i = 0;
    while (i < intervals.length && intervals[i][1] < newInterval[0]) {
        result.add(intervals[i]);
        i++;
    }
    while (i < intervals.length && intervals[i][0] <= newInterval[1]) {
        newInterval[0] = Math.min(newInterval[0], intervals[i][0]);
        newInterval[1] = Math.max(newInterval[1], intervals[i][1]);
        i++;
    }
    result.add(newInterval);
```

```
    while (i < intervals.length) {
        result.add(intervals[i]);
        i++;
    }
    return result.toArray(new int[result.size()][]);
}
```

Time Complexity: O(n)

Space Complexity: O(1)

# 2. LeetCode 56 Merge Intervals (medium)

- Given an array of `intervals` where `intervals[i] = [starti, endi]`, merge all overlapping intervals, and return *an array of the non-overlapping intervals that cover all the intervals in the input*.
- **Example 1:**
  - **Input:** intervals = `[[1,3],[2,6],[8,10],[15,18]]`
  - **Output:** `[[1,6],[8,10],[15,18]]`
  - **Explanation:** Since intervals [1,3] and [2,6] overlap, merge them into [1,6].
- **Example 2:**
  - **Input:** intervals = `[[1,4],[4,5]]`
  - **Output:** `[[1,5]]`
  - **Explanation:** Intervals [1,4] and [4,5] are considered overlapping.
- **Constraints:**
  - `1 <= intervals.length <= 10^4`
  - `intervals[i].length == 2`
  - `0 <= starti <= endi <= 10^4`

## Solution

```
public int[][] merge(int[][] intervals) {
    if (intervals == null || intervals.length == 0) {
        return intervals;
    }
    Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));
    List<int[]> result = new ArrayList<>();
    for (int[] interval : intervals) {
        if (result.isEmpty() || result.get(result.size() - 1)[1] < interval[0]) {
            result.add(interval);
        } else {
            result.get(result.size() - 1)[1] = Math.max(result.get(result.size() - 1)[1],
interval[1]);
        }
    }
    return result.toArray(new int[result.size()][]);
}
```

Time Complexity: O(nlogn)

Space Complexity: O(logn)

# 3. LeetCode 435 Non-overlapping Intervals (medium)

- Given an array of intervals `intervals` where `intervals[i] = [starti, endi]`, return *the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping*.
- **Example 1:**
  - **Input:** intervals = `[[1,2],[2,3],[3,4],[1,3]]`
  - **Output:** 1
  - **Explanation:** [1,3] can be removed and the rest of the intervals are non-overlapping.
- **Example 2:**
  - **Input:** intervals = `[[1,2],[1,2],[1,2]]`
  - **Output:** 2
  - **Explanation:** You need to remove two [1,2] to make the rest of the intervals non-overlapping.
- **Example 3:**
  - **Input:** intervals = `[[1,2],[2,3]]`
  - **Output:** 0
  - **Explanation:** You don't need to remove any of the intervals since they're already non-overlapping.
- **Constraints:**
  - `1 <= intervals.length <= 10^5`
  - `intervals[i].length == 2`
  - `-5 * 10^4 <= starti < endi <= 5 * 10^4`

## Solution

```java
public int eraseOverlapIntervals(int[][] intervals) {
    if (intervals == null || intervals.length == 0) {
        return 0;
    }
    Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));
    int result = 0, end = intervals[0][1];
    for (int i = 1; i < intervals.length; i++) {
        if (end > intervals[i][0]) {
            end = Math.min(end, intervals[i][1]);
            result++;
        } else {
            end = intervals[i][1];
        }
    }
    return result;
}
```

Time Complexity: O(nlogn)

Space Complexity: O(logn)

# 4. LeetCode 252 Meeting Rooms (easy)

- Given an array of meeting time `intervals` where `intervals[i] = [starti, endi]`, determine if a person could attend all meetings.
- **Example 1:**
  - **Input:** intervals = `[[0,30],[5,10],[15,20]]`
  - **Output:** false
- **Example 2:**

- **Input:** intervals = `[[7,10],[2,4]]`
- **Output:** true

- **Constraints:**
  - `0 <= intervals.length <= 10^4`
  - `intervals[i].length == 2`
  - `0 <= starti < endi <= 10^6`

## Solution 1

```java
public boolean canAttendMeetings(int[][] intervals) {
    if (intervals == null || intervals.length == 0) {
        return true;
    }
    Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));
    for (int i = 0; i < intervals.length - 1; i++) {
        if (intervals[i][1] > intervals[i + 1][0]) {
            return false;
        }
    }
    return true;
}
```

Time Complexity: O(nlogn)

Space Complexity: O(logn)

## Solution 2

```java
public boolean canAttendMeetings(int[][] intervals) {
    // assumption: start < end
    if (intervals == null || intervals.length == 0) {
        return true;
    }
    try {
        Arrays.sort(intervals, (a, b) -> {
            if (a[1] <= b[0]) { // a[0] < a[1] <= b[0]
                return -1;
            } else if (a[0] >= b[1]) { // b[0] < b[1] <= a[0]
                return 1;
            }
            throw new RuntimeException("overlap detected");
        });
        return true;
    } catch (RuntimeException e) {
        return false;
    }
}
```

Time Complexity: O(logn)

Space Complexity: O(n)

# 5. LeetCode 253 Meeting Rooms II (medium)

- Given an array of meeting time intervals `intervals` where `intervals[i] = [starti, endi]`, return *the minimum number of conference rooms required*.
- **Example 1:**
  - **Input:** intervals = `[[0,30],[5,10],[15,20]]`
  - **Output:** 2
- **Example 2:**
  - **Input:** intervals = `[[7,10],[2,4]]`
  - **Output:** 1
- **Constraints:**
  - `1 <= intervals.length <= 10^4`
  - `0 <= starti < endi <= 10^6`

# Solution

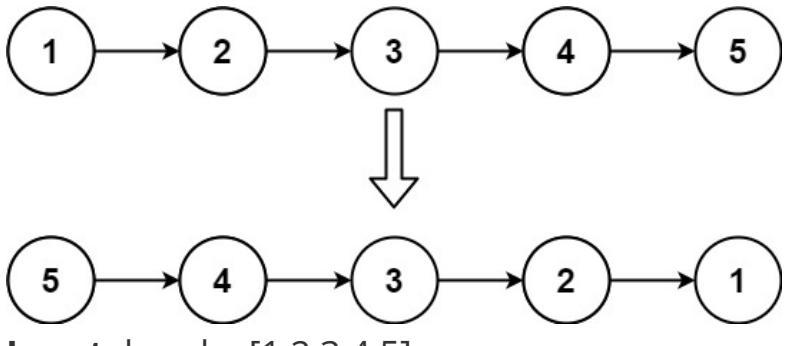
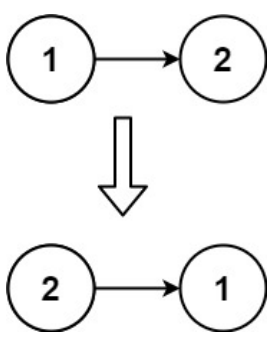
```java
public int minMeetingRooms(int[][] intervals) {
    if (intervals == null || intervals.length == 0) {
        return 0;
    }
    Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));
    PriorityQueue<Integer> minHeap = new PriorityQueue<>(intervals.length);
    minHeap.offer(intervals[0][1]);
    for (int i = 1; i < intervals.length; i++) {
        if (intervals[i][0] >= minHeap.peek()) {
            minHeap.poll();
        }
        minHeap.offer(intervals[i][1]);
    }
    return minHeap.size();
}
```

Time Complexity: O(nlogn)

Space Complexity: O(n)

# Linked List

## 1. LeetCode 206 Reverse a Linked List (easy)

- 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:** []
- **Constraints:**
  - The number of nodes in the list is the range `[0, 5000]`.
  - `-5000 <= Node.val <= 5000`
- **Follow up:** A linked list can be reversed either iteratively or recursively. Could you implement both?

## Solution 1: iterative

```java
public ListNode reverseList(ListNode head) {
    // corner case
    if (head == null) {
        return null;
    }
    ListNode previous = null;
    ListNode current = head;
    ListNode next = null;
    while (current != null) {
        next = current.next; // store next node
        current.next = previous; // reverse
        previous = current; // previous moves one step
        current = next; // next moves one step
    }
    return previous;
}
```

Time Complexity: O(n)

Space Complexity: O(1)

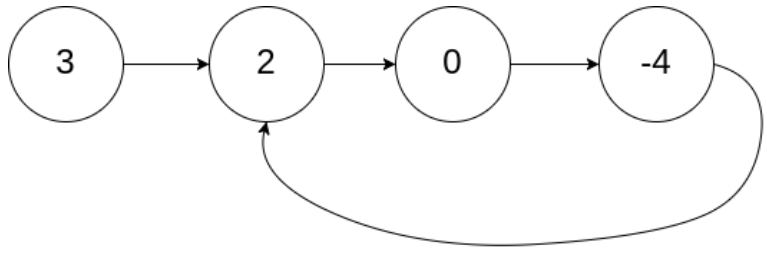
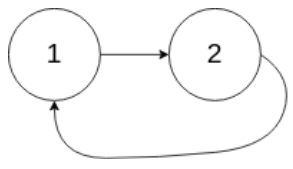
# Solution 2: recursive

```java
public ListNode reverseList(ListNode head) {
    // base case
    if (head == null || head.next == null) {
        return head;
    }
    // recursive step
    ListNode newHead = reverseList(head.next);
    head.next.next = head;
    head.next = null;
    return newHead;
}
```

Time Complexity: O(n)

Space Complexity: O(n)

# 2. LeetCode 141 Detect Cycle in a Linked List (easy)

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

    - `pos` is `-1` or a **valid index** in the linked-list.
- **Follow up:** Can you solve it using `O(1)` (i.e. constant) memory?

## Solution: two pointers

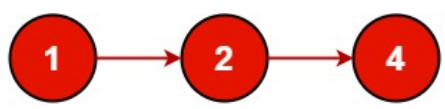
```java
public boolean hasCycle(ListNode head) {
    if (head == null) {
        return false;
    }
    ListNode slow = head;
    ListNode fast = head;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
        if (slow == fast) {
            return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

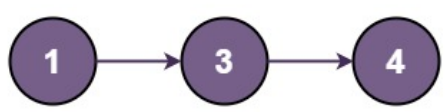
Time Complexity: O(n)

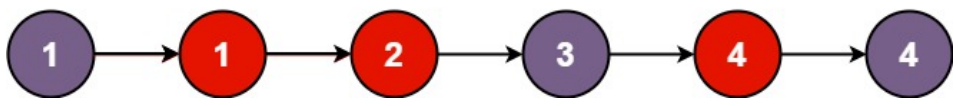
Space omplexity: O(1)

## 3. LeetCode 21 Merge Two Sorted Lists (easy)

- You are given the heads of two sorted linked lists `list1` and `list2`.
- Merge the two lists in a 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]
- **Constraints:**
    - The number of nodes in both lists is in the range `[0, 50]`.
    - `-100 <= Node.val <= 100`

- Both `list1` and `list2` are sorted in **non-decreasing** order.

## Solution: 谁小移谁

```java
public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
    if (list1 == null && list2 == null) {
        return null;
    }
    ListNode dummy = new ListNode(0);
    ListNode tail = dummy;
    while (list1 != null && list2 != null) {
        if (list1.val <= list2.val) {
            tail.next = list1;
            list1 = list1.next;
        } else {
            tail.next = list2;
            list2 = list2.next;
        }
        tail = tail.next;
    }
    if (list1 == null) {
        tail.next = list2;
    }
    if (list2 == null) {
        tail.next = list1;
    }
    return dummy.next;
}
```

Time Complexity: O(n)

Space Complexity: O(1)

## 4. LeetCode 23 Merge K Sorted Lists (hard)

- You are given an array of `k` linked-lists `lists`, each linked-list is sorted in ascending order.
- *Merge all the linked-lists into one sorted linked-list and return it.*
- **Example 1:**
  - **Input:** lists = `[[1,4,5],[1,3,4],[2,6]]`
  - **Output:** [1,1,2,3,4,4,5,6]
- **Example 2:**
  - **Input:** lists = `[]`
  - **Output:** []
- **Example 3:**
  - **Input:** lists = `[[]]`
  - **Output:** []
- **Constraints:**
  - `k == lists.length`
  - `0 <= k <= 10^4`
  - `0 <= lists[i].length <= 500`
  - `-10^4 <= lists[i][j] <= 10^4`
  - `lists[i]` is sorted in **ascending order**.

- The sum of `lists[i].length` will not exceed `10^4`.

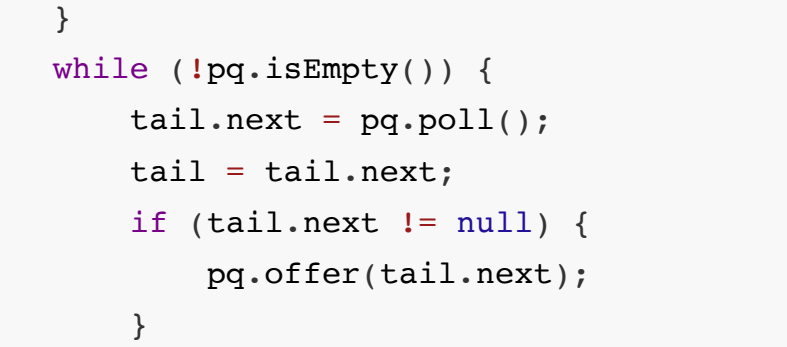
## Solution: priority queue (min heap)

```java
public ListNode mergeKLists(ListNode[] lists) {
    if (lists == null || lists.length == 0) {
        return null;
    }
    PriorityQueue<ListNode> pq = new PriorityQueue<>(lists.length, (n1, n2) -> {
        if (n1.val == n2.val) {
            return 0;
        }
        return n1.val < n2.val ? -1 : 1;
    });
    ListNode dummy = new ListNode(0);
    ListNode tail = dummy;
    for (ListNode node : lists) {
        if (node != null) {
            pq.offer(node);
        }
    }
    while (!pq.isEmpty()) {
        tail.next = pq.poll();
        tail = tail.next;
        if (tail.next != null) {
            pq.offer(tail.next);
        }
    }
    return dummy.next;
}
```

Time Complexity: O(nlogk)

Space Complexity: O(k)

## 5. LeetCode 19 Remove Nth Node From End Of List (medium)

- Given the `head` of a linked list, remove the `nth` 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]
- **Constraints:**
    - The number of nodes in the list is `sz`.
    - `1 <= sz <= 30`
    - `0 <= Node.val <= 100`
    - `1 <= n <= sz`
- **Follow up:** Could you do this in one pass?

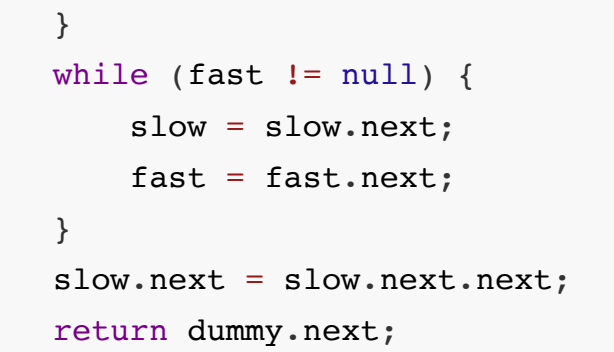
# Solution: like sliding window

```java
public ListNode removeNthFromEnd(ListNode head, int n) {
    // assumption: 1 <= n <= number of nodes in the list
    if (head == null) {
        return null;
    }
    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) {
        slow = slow.next;
        fast = fast.next;
    }
    slow.next = slow.next.next;
    return dummy.next;
}
```

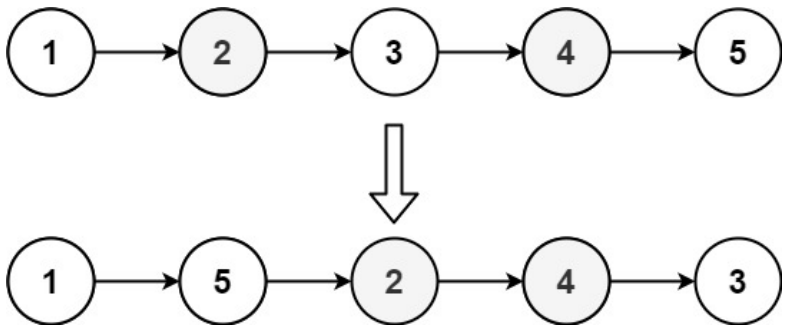
Time Complexity: O(n)

Space Complexity: O(1)

# 6. LeetCode 143 Reorder List (medium)

- You are given the head of a singly linked-list. The list can be represented as: `L0 → L1 → … → Ln - 1 → Ln`
- *Reorder the list to be on the following form:* `L0 → Ln → L1 → Ln - 1 → L2 → Ln - 2 → …`
- You may not modify the values in the list's nodes. Only nodes themselves may be changed.
- **Example 1:**
    - 

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




  - **Input:** head = [1,2,3,4,5]
  - **Output:** [1,5,2,4,3]
- **Constraints:**
  - The number of nodes in the list is in the range `[1, 5 * 10^4]`.
  - `1 <= Node.val <= 1000`

# Solution

```java
public void reorderList(ListNode head) {
    if (head == null || head.next == null) {
        return;
    }
    ListNode middle = findMiddle(head);
    ListNode one = head;
    ListNode two = middle.next;
    middle.next = null;
    head = merge(one, reverse(two));
    return;
}

private ListNode findMiddle(ListNode head) {
    if (head == null) {
        return null;
    }
    ListNode slow = head;
    ListNode fast = head;
    while (fast.next != null && fast.next.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }
    return slow;
}

private ListNode reverse(ListNode head) {
    if (head == null) {
        return null;
    }
    ListNode previous = null;
    ListNode current = head;
    ListNode next = null;
    while (current != null) {
        next = current.next;
        current.next = previous;
        previous = current;
        current = next;
    }
    return previous;
}

```

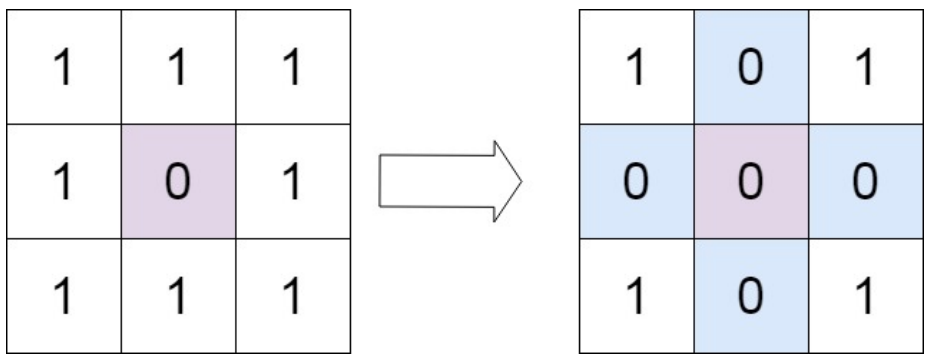
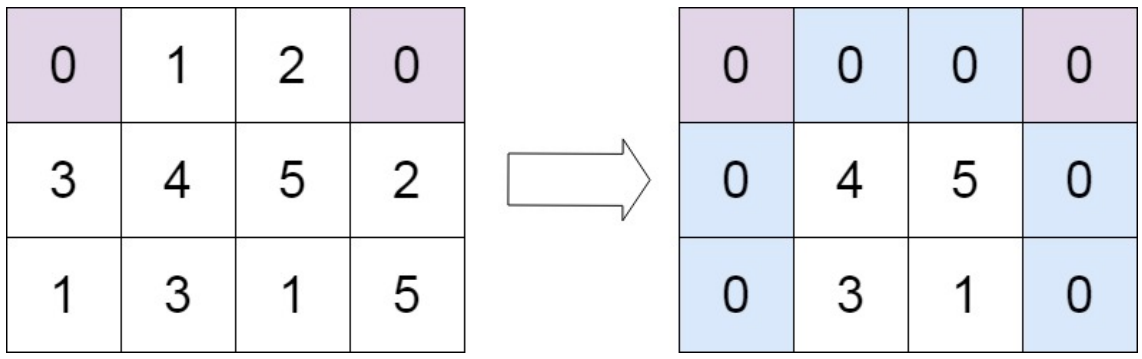
```
42 private ListNode merge(ListNode one, ListNode two) {
43     ListNode dummy = new ListNode(0);
44     ListNode current = dummy;
45     while (one != null && two != null) {
46         current.next = one;
47         one = one.next;
48         current.next.next = two;
49         two = two.next;
50         current = current.next.next;
51     }
52     if (one != null) {
53         current.next = one;
54     }
55     if (two != null) {
56         current.next = two;
57     }
58     return dummy.next;
59 }
```

Time Complexity: O(n)

Space Complexity: O(1)

# Matrix

## 1. LeetCode 73 Set Matrix Zeros (medium)

- Given an `m x n` integer matrix `matrix`, if an element is `0`, set its entire row and column to `0`'s.
- You must do it in place.
- **Example 1:**
  - 
  - **Input:** matrix = `[[1,1,1],[1,0,1],[1,1,1]]`
  - **Output:** `[[1,0,1],[0,0,0],[1,0,1]]`
- **Example 2:**
  - 
  - **Input:** matrix = `[[0,1,2,0],[3,4,5,2],[1,3,1,5]]`
  - **Output:** `[[0,0,0,0],[0,4,5,0],[0,3,1,0]]`
- **Constraints:**
  - `m == matrix.length`
  - `n == matrix[0].length`
  - `1 <= m, n <= 200`
  - `-2^31 <= matrix[i][j] <= 2^31 - 1`
- **Follow up:**
  - A straightforward solution using `O(mn)` space is probably a bad idea.
  - A simple improvement uses `O(m + n)` space, but still not the best solution.
  - Could you devise a constant space solution?

## Solution

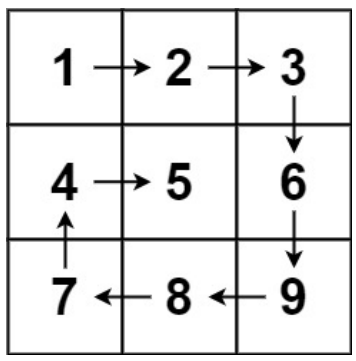
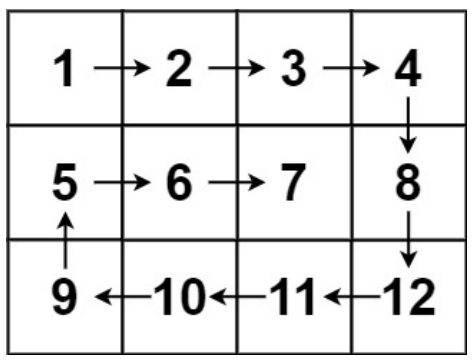
```java
public void setZeroes(int[][] matrix) {
    if (matrix == null || matrix.length == 0) {
        return;
    }
    int k = 0;
    while (k < matrix[0].length && matrix[0][k] != 0) {
        k++;
    }
    for (int i = 1; i < matrix.length; i++) {
        for (int j = 0; j < matrix[0].length; j++) {
            if (matrix[i][j] == 0) {
                matrix[i][0] = 0;
                matrix[0][j] = 0;
```

```
            }
        }
    }
    for (int i = 1; i < matrix.length; i++) {
        for (int j = matrix[0].length - 1; j >= 0; j--) {
            if (matrix[i][0] == 0 || matrix[0][j] == 0) {
                matrix[i][j] = 0;
            }
        }
    }
    if (k < matrix[0].length) {
        Arrays.fill(matrix[0], 0);
    }
    return;
}
```

Time Complexity: O(mn)

Space Complexity: O(1)

## 2. LeetCode 54 Spiral Matrix (medium)

- Given an `m x n` `matrix`, return *all elements of the* `matrix` *in spiral order*.
- **Example 1:**
  - 
  - **Input:** matrix = `[[1,2,3],[4,5,6],[7,8,9]]`
  - **Output:** [1,2,3,6,9,8,7,4,5]
- **Example 2:**
  - 
  - **Input:** matrix = `[[1,2,3,4],[5,6,7,8],[9,10,11,12]]`
  - **Output:** [1,2,3,4,8,12,11,10,9,5,6,7]
- **Constraints:**
  - `m == matrix.length`
  - `n == matrix[i].length`
  - `1 <= m, n <= 10`
  - `-100 <= matrix[i][j] <= 100`

## Solution

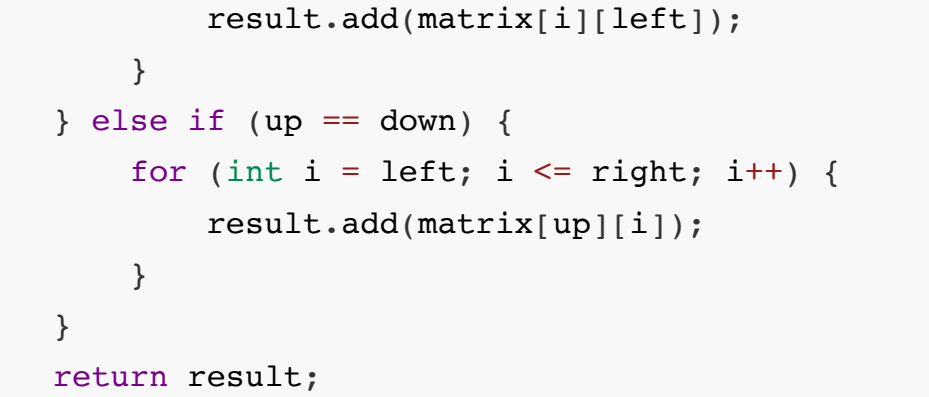
```
public List<Integer> spiralOrder(int[][] matrix) {
    List<Integer> result = new ArrayList<>();
    if (matrix == null || matrix.length == 0) {
```

```
        return result;
    }
    int m = matrix.length;
    int n = matrix[0].length;
    int left = 0, right = n - 1;
    int up = 0, down = m - 1;
    while (left < right && up < down) {
        for (int i = left; i <= right; i++) {
            result.add(matrix[up][i]);
        }
        for (int i = up + 1; i <= down - 1; i++) {
            result.add(matrix[i][right]);
        }
        for (int i = right; i >= left; i--) {
            result.add(matrix[down][i]);
        }
        for (int i = down - 1; i >= up + 1; i--) {
            result.add(matrix[i][left]);
        }
        left++;
        right--;
        up++;
        down--;
    }
    if (left == right) {
        for (int i = up; i <= down; i++) {
            result.add(matrix[i][left]);
        }
    } else if (up == down) {
        for (int i = left; i <= right; i++) {
            result.add(matrix[up][i]);
        }
    }
    return result;
}
```

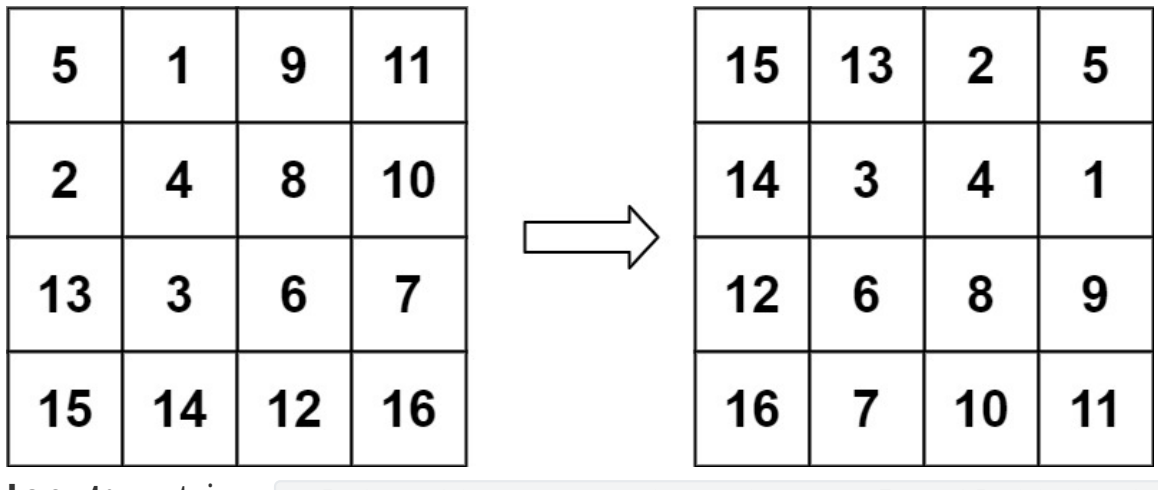
Time Complexity: O(mn)

Space Complexity: O(1)

# 3. LeetCode 48 Rotate Image (medium)

- You are given an `n x 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:**
  - 

  - **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]]`
- **Constraints:**
  - `n == matrix.length == matrix[i].length`
  - `1 <= n <= 20`
  - `-1000 <= matrix[i][j] <= 1000`

## Solution

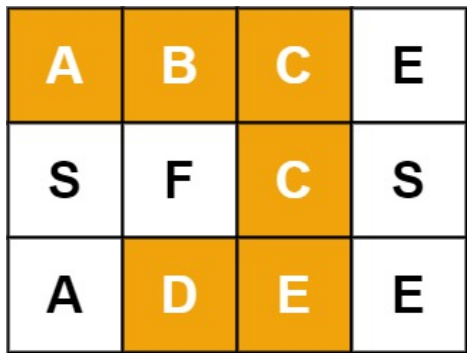
```java
public void rotate(int[][] matrix) {
    if (matrix == null || matrix.length == 0) {
        return;
    }
    for (int i = 0; i < matrix.length / 2; i++) {
        int[] temp = matrix[i];
        matrix[i] = matrix[matrix.length - 1 - i];
        matrix[matrix.length - 1 - i] = temp;
    }
    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < i; j++) {
            int temp = matrix[i][j];
            matrix[i][j] = matrix[j][i];
            matrix[j][i] = temp;
        }
    }
    return;
}
```

Time Complexity: O(n^2)

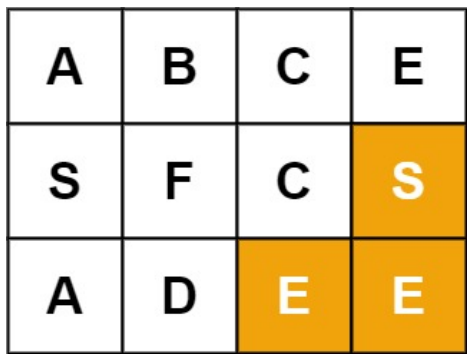
Space Complexity: O(1)

# 4. LeetCode 79 Word Search (medium)

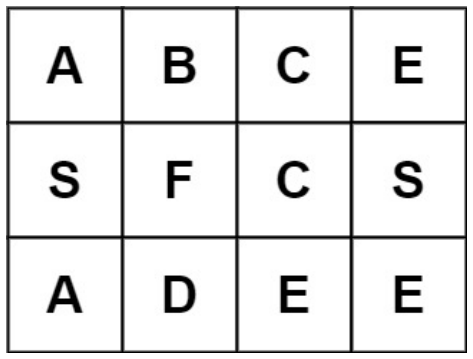
- Given an `m x n` grid of characters `board` and a string `word`, return `true` *if* `word` *exists in the grid*.
- The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.
- **Example 1:**

  - 

| A | B | C | E |
|---|---|---|---|
| S | F | C | S |
| A | D | E | E |

  - **Input:** board = `[["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]]`, word = "ABCCED"
  - **Output:** true
- **Example 2:**
  - 

| A | B | C | E |
|---|---|---|---|
| S | F | C | S |
| A | D | E | E |

  - **Input:** board = `[["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]]`, word = "SEE"
  - **Output:** true
- **Example 3:**
  - 

| A | B | C | E |
|---|---|---|---|
| S | F | C | S |
| A | D | E | E |

  - **Input:** board = `[["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]]`, word = "ABCB"
  - **Output:** false
- **Constraints:**
  - `m == board.length`
  - `n = board[i].length`
  - `1 <= m, n <= 6`
  - `1 <= word.length <= 15`
  - `board` and `word` consists of only lowercase and uppercase English letters.
- **Follow up:** Could you use search pruning to make your solution faster with a larger `board`?

## Solution: backtracking

```java
public boolean exist(char[][] board, String word) {
    if (board == null) {
        return false;
    }
    if (word == null || word.length() == 0) {
        return true;
    }
    boolean[][] visited = new boolean[board.length][board[0].length];
    for (int i = 0; i < board.length; i++) {
        for (int j = 0; j < board[0].length; j++) {
            if (board[i][j] == word.charAt(0) && helper(board, word, i, j, 0, visited)) {
                return true;
            }
        }
    }
```

```
    return false;
}

private boolean helper(char[][] board, String word, int i, int j, int index, boolean[][] visited) {
    if (index == word.length()) {
        return true;
    }
    if (i < 0 || i >= board.length || j < 0 || j >= board[0].length || board[i][j] != word.charAt(index) || visited[i][j]) {
        return false;
    }
    visited[i][j] = true;
    if (helper(board, word, i - 1, j, index + 1, visited) || helper(board, word, i + 1, j, index + 1, visited) || helper(board, word, i, j - 1, index + 1, visited) || helper(board, word, i, j + 1, index + 1, visited)) {
        return true;
    }
    visited[i][j] = false;
    return false;
}
```

Time Complexity: O(mn * 3^L)

Space Complexity: O(L)

# String

- char -> int
  1. all unique characters: `int index = c - 'a';`
  2. parse a string representation of positive integer: `int i = c - '0';`
- int -> char
  1. convert a digit into its string representation: `char c = (char) (digit + '0');`
  2. interpret an integer as ASCII code: `char c = (char) i;`

## 1. LeetCode 3 Longest Substring Without Repeating Characters (medium)

- Given a string `s`, find the length of the **longest substring** without repeating characters.
- **Example 1:**
  - **Input:** s = "abcabcbb"
  - **Output:** 3
  - **Explanation:** The answer is "abc", with the length of 3.
- **Example 2:**
  - **Input:** s = "bbbbb"
  - **Output:** 1
  - **Explanation:** The answer is "b", with the length of 1.
- **Example 3:**
  - **Input:** s = "pwwkew"
  - **Output:** 3
  - **Explanation:** The answer is "wke", with the length of 3.
    - Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.
- **Constraints:**
  - `0 <= s.length <= 5 * 10^4`
  - `s` consists of English letters, digits, symbols and spaces.

### Solution

```java
public int lengthOfLongestSubstring(String s) {
    if (s == null || s.length() == 0) {
        return 0;
    }
    int result = 0;
    int[] cache = new int[256]; // assumption: all characters are from ACSII
    for (int slow = 0, fast = 0; fast < s.length(); fast++) {
        slow = cache[s.charAt(fast)] > 0 ? Math.max(slow, cache[s.charAt(fast)]) : slow;
        cache[s.charAt(fast)] = fast + 1;
        result = Math.max(result, fast - slow + 1);
    }
    return result;
}
```

Time Complexity: O(n)

Space Complexity: O(1)

# 2. LeetCode 424 Longest Repeating Character Replacement (medium)

- You are given a string `s` and an integer `k`. You can choose any character of the string and change it to any other uppercase English character. You can perform this operation at most `k` times.
- Return *the length of the longest substring containing the same letter you can get after performing the above operations*.
- **Example 1:**
  - **Input:** s = "ABAB", k = 2
  - **Output:** 4
  - **Explanation:** Replace the two 'A's with two 'B's or vice versa.
- **Example 2:**
  - **Input:** s = "AABABBA", k = 1
  - **Output:** 4
  - **Explanation:** Replace the one 'A' in the middle with 'B' and form "AABBBBA". The substring "BBBB" has the longest repeating letters, which is 4.
- **Constraints:**
  - `1 <= s.length <= 10^5`
  - `s` consists of only uppercase English letters.
  - `0 <= k <= s.length`

## Solution

```java
public int characterReplacement(String s, int k) {
    if (s == null || s.length() == 0) {
        return 0;
    }
    int[] cache = new int[26]; // assumption: s consists of only uppercase English letters
    int result = 0, maxCount = 0;
    for (int slow = 0, fast = 0; fast < s.length(); fast++) {
        maxCount = Math.max(maxCount, ++cache[s.charAt(fast) - 'A']);
        while (maxCount + k < fast - slow + 1) {
            cache[s.charAt(slow) - 'A']--;
            slow++;
        }
        result = Math.max(result, fast - slow + 1);
    }
    return result;
}
```

Time Complexity: O(n)

Space Complexity: O(n)

# 3. LeetCode 76 Minimum Window Substring (hard)

- Given two strings `s` and `t` of lengths `m` and `n` respectively, return *the* ***minimum window substring*** *of* `s` *such that every character in* `t` *(****including duplicates****) is included in the window. If there is no such substring__, return the empty string* `""`.

- The testcases will be generated such that the answer is **unique**.
- A **substring** is a contiguous sequence of characters within the string.
- **Example 1:**
  - **Input:** s = "ADOBECODEBANC", t = "ABC"
  - **Output:** "BANC"
  - **Explanation:** The minimum window substring "BANC" includes 'A', 'B', and 'C' from string t.
- **Example 2:**
  - **Input:** s = "a", t = "a"
  - **Output:** "a"
  - **Explanation:** The entire string s is the minimum window.
- **Example 3:**
  - **Input:** s = "a", t = "aa"
  - **Output:** ""
  - **Explanation:** Both 'a's from t must be included in the window. Since the largest window of s only has one 'a', return empty string.
- **Constraints:**
  - `m == s.length`
  - `n == t.length`
  - `1 <= m, n <= 10^5`
  - `s` and `t` consist of uppercase and lowercase English letters.
- **Follow up:** Could you find an algorithm that runs in `O(m + n)` time?

## Solution

```java
public String minWindow(String s, String t) {
    if (s == null || t == null) {
        return null;
    }
    if (s.length() == 0 || t.length() == 0) {
        return "";
    }
    Map<Character, Integer> map = new HashMap<>();
    for (int i = 0; i < t.length(); i++) {
        map.put(t.charAt(i), map.getOrDefault(t.charAt(i), 0) + 1);
    }
    int[] result = {-1, 0, 0};
    int left = 0, right = 0, count = 0;
    Map<Character, Integer> window = new HashMap<>();
    while (right < s.length()) {
        char c = s.charAt(right);
        window.put(c, window.getOrDefault(c, 0) + 1);
        if (map.containsKey(c) && map.get(c).equals(window.get(c))) {
            count++;
        }
        while (left <= right && count == map.size()) {
            c = s.charAt(left);
            if (result[0] == -1 || right - left + 1 < result[0]) {
                result[0] = right - left + 1;
                result[1] = left;
                result[2] = right;
            }
            window.put(c, window.get(c) - 1);
```

```
            if (map.containsKey(c) && window.get(c).intValue() < map.get(c).intValue()) {
                count--;
            }
            left++;
        }
        right++;
    }
    return result[0] == -1 ? "" : s.substring(result[1], result[2] + 1);
}
```

Time Complexity: O(m + n)

Space Complexity: O(m + n)

# 4. LeetCode 242 Valid Anagram (easy)

- 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
- **Constraints:**
  - `1 <= s.length, t.length <= 5 * 10^4`
  - `s` and `t` consist of lowercase English letters.
- **Follow up:** What if the inputs contain Unicode characters? How would you adapt your solution to such a case?

## Solution

```
public boolean isAnagram(String s, String t) {
    // assumption: s and t are not null
    if (s.length() != t.length()) {
        return false;
    }
    int[] cache = new int[256]; // assumption: all characters are from ACSII
    for (int i = 0; i < s.length(); i++) {
        cache[s.charAt(i)]++;
    }
    for (int i = 0; i < t.length(); i++) {
        cache[t.charAt(i)]--;
        if (cache[t.charAt(i)] < 0) {
            return false;
        }
    }
    return true;
}
```

Time Complexity: O(n)

Space Complexity: O(1)

# 5. LeetCode 49 Group Anagrams (medium)

- Given an array of strings `strs`, group **the anagrams** together. You can return the answer in **any order**.
- 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:** strs = ["eat","tea","tan","ate","nat","bat"]
  - **Output:** `[["bat"],["nat","tan"],["ate","eat","tea"]]`
- **Example 2:**
  - **Input:** strs = [""]
  - **Output:** `[[""]]`
- **Example 3:**
  - **Input:** strs = ["a"]
  - **Output:** `[["a"]]`
- **Constraints:**
  - `1 <= strs.length <= 10^4`
  - `0 <= strs[i].length <= 100`
  - `strs[i]` consists of lowercase English letters.

## Solution 1

```java
public List<List<String>> groupAnagrams(String[] strs) {
    if (strs == null || strs.length == 0) {
        return new ArrayList<>();
    }
    Map<String, List<String>> result = new HashMap<>();
    for (String s : strs) {
        char[] array = s.toCharArray();
        Arrays.sort(array);
        String key = new String(array);
        if (!result.containsKey(key)) {
            result.put(key, new ArrayList<>());
        }
        result.get(key).add(s);
    }
    return new ArrayList<>(result.values());
}
```

Time Complexity: O(nLlogL)

Space Complexity: O(nL)

## Solution 2

```java
public List<List<String>> groupAnagrams(String[] strs) {
    if (strs == null || strs.length == 0) {
        return new ArrayList<>();
    }
    Map<String, List> result = new HashMap<>();
```

```
    int[] count = new int[26]; // assumption: strs[i] consists of lowercase English letters
    for (String s : strs) {
        Arrays.fill(count, 0);
        for (char c : s.toCharArray()) {
            count[c - 'a']++;
        }
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < 26; i++) {
            sb.append('#');
            sb.append(count[i]);
        }
        String key = sb.toString();
        if (!result.containsKey(key)) {
            result.put(key, new ArrayList());
        }
        result.get(key).add(s);
    }
    return new ArrayList(result.values());
}
```

Time Complexity: O(nL)

Space Complexity: O(nL)

## 6. LeetCode 20 Valid Parentheses (easy)

- 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.
- **Example 1:**
  - **Input:** s = `"()"`
  - **Output:** true
- **Example 2:**
  - **Input:** s = `"()[]{}"`
  - **Output:** true
- **Example 3:**
  - **Input:** s = `"(]"`
  - **Output:** false
- **Constraints:**
  - `1 <= s.length <= 10^4`
  - `s` consists of parentheses only `'()[]{}'`.

## Solution

```
public boolean isValid(String s) {
    if (s == null || s.length() == 0) {
        return true;
    }
    Map<Character, Character> map = new HashMap<>();
```

```
    map.put(')', '(');
    map.put(']', '[');
    map.put('}', '{');
    Deque<Character> stack = new ArrayDeque<>();
    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        if (map.containsKey(c)) {
            if (stack.isEmpty()) {
                return false;
            }
            if (stack.pollFirst() != map.get(c)) {
                return false;
            }
        } else {
            stack.offerFirst(c);
        }
    }
    return stack.isEmpty();
}
```

Time Complexity: O(n)

Space Complexity: O(n)

## 7. LeetCode 125 Valid Palindrome (easy)

- 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.
- **Constraints:**
  - `1 <= s.length <= 2 * 10^5`
  - `s` consists only of printable ASCII characters.

## Solution

```
public boolean isPalindrome(String s) {
    if (s == null || s.length() == 0) {
        return true;
```

```
    }
    for (int left = 0, right = s.length() - 1; left < right; left++, right--) {
        while (left < right && !Character.isLetterOrDigit(s.charAt(left))) {
            left++;
        }
        while (left < right && !Character.isLetterOrDigit(s.charAt(right))) {
            right--;
        }
        if (Character.toLowerCase(s.charAt(left)) !=
Character.toLowerCase(s.charAt(right))) {
            return false;
        }
    }
    return true;
}
```

Time Complexity: O(n)

Space Complexity: O(1)

# 8. LeetCode 5 Longest Palindromic Substring (medium)

- Given a string `s`, return *the longest palindromic substring* in `s`.
- **Example 1:**
  - **Input:** s = "babad"
  - **Output:** "bab"
  - **Explanation:** "aba" is also a valid answer.
- **Example 2:**
  - **Input:** s = "cbbd"
  - **Output:** "bb"
- **Constraints:**
  - `1 <= s.length <= 1000`
  - `s` consist of only digits and English letters.

## Solution: expand from center

```
public String longestPalindrome(String s) {
    if (s == null || s.length() < 2) {
        return s;
    }
    char[] array = s.toCharArray();
    int start = 0, end = 0;
    for (int i = 0; i < array.length; i++) {
        int maxLength = Math.max(expand(array, i, i), expand(array, i, i + 1));
        if (end - start < maxLength) {
            start = i - (maxLength - 1) / 2;
            end = i + maxLength / 2;
        }
    }
    return s.substring(start, end + 1);
}

private int expand(char[] array, int i, int j) {
    while (i >= 0 && j < array.length && array[i] == array[j]) {
```

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

Time Complexity: O(n^2)

Space Complexity: O(1)

# 9. LeetCode 647 Palindromic Substrings (medium)

- Given a string `s`, return *the number of **palindromic substrings** in it*.
- A string is a **palindrome** when it reads the same backward as forward.
- A **substring** is a contiguous sequence of characters within the string.
- **Example 1:**
  - **Input:** s = "abc"
  - **Output:** 3
  - **Explanation:** Three palindromic strings: "a", "b", "c".
- **Example 2:**
  - **Input:** s = "aaa"
  - **Output:** 6
  - **Explanation:** Six palindromic strings: "a", "a", "a", "aa", "aa", "aaa".
- **Constraints:**
  - `1 <= s.length <= 1000`
  - `s` consists of lowercase English letters.

## Solution

```
public int countSubstrings(String s) {
    if (s == null || s.length() == 0) {
        return 0;
    }
    int result = 0;
    for (int i = 0; i < s.length(); i++) {
        int left = i - 1, right = i;
        while (right < s.length() - 1 && s.charAt(right) == s.charAt(right + 1)) {
            right++;
        }
        result += (right - left) * (right - left + 1) / 2;
        i = right++;
        while (left >= 0 && right < s.length() && s.charAt(left--) == s.charAt(right++))
{
            result++;
        }
    }
    return result;
}
```

Time Complexity: O(n)

Space Complexity: O(1)

# 10. LeetCode 271 Encode and Decode Strings (medium)

- Design an algorithm to encode **a list of strings** to **a string**. The encoded string is then sent over the network and is decoded back to the original list of strings.
- Machine 1 (sender) has the function:

```
string encode(vector<string> strs) {
  // ... your code
  return encoded_string;
}
```

- Machine 2 (receiver) has the function:

```
vector<string> decode(string s) {
  //... your code
  return strs;
}
```

- So Machine 1 does: `string encoded_string = encode(strs);` and Machine 2 does: `vector<string> strs2 = decode(encoded_string);`
- `strs2` in Machine 2 should be the same as `strs` in Machine 1.
- Implement the `encode` and `decode` methods.
- You are not allowed to solve the problem using any serialize methods (such as `eval`).
- **Example 1:**
  - **Input:** dummy_input = ["Hello","World"]
  - **Output:** ["Hello","World"]
  - **Explanation:**
    - Machine 1:

```
Codec encoder = new Codec();
String msg = encoder.encode(strs);
```

    - Machine 1 ---msg---> Machine 2
    - Machine 2:

```
Codec decoder = new Codec();
String[] strs = decoder.decode(msg);
```

- **Example 2:**
  - **Input:** dummy_input = [""]
  - **Output:** [""]
- **Constraints:**
  - `1 <= strs.length <= 200`
  - `0 <= strs[i].length <= 200`
  - `strs[i]` contains any possible characters out of `256` valid ASCII characters.
- **Follow up:** Could you write a generalized algorithm to work on any possible set of characters?

# Solution

```java
public String encode(List<String> strs) {
    StringBuilder sb = new StringBuilder();
    for (String s : strs) {
        sb.append(s.replace("#", "##")).append(" # ");
    }
    sb.deleteCharAt(sb.length() - 1);
    return sb.toString();
}

public List<String> decode(String s) {
    List<String> result = new ArrayList<>();
    String[] array = s.split(" # ", -1);
    for (String str : array) {
        result.add(str.replace("##", "#"));
    }
    return result;
}
```

# Solution 1: non-ASCII delimiter

```java
public String encode(List<String> strs) {
    if (strs.size() == 0) {
        return Character.toString((char)258);
    }
    String delimiter = Character.toString((char)257);
    StringBuilder sb = new StringBuilder();
    for (String str : strs) {
        sb.append(str).append(delimiter);
    }
    sb.deleteCharAt(sb.length() - 1);
    return sb.toString();
}

public List<String> decode(String s) {
    if (s.equals(Character.toString((char)258))) {
        return new ArrayList();
    }
    String delimiter = Character.toString((char)257);
    return Arrays.asList(s.split(delimiter, -1));
}
```

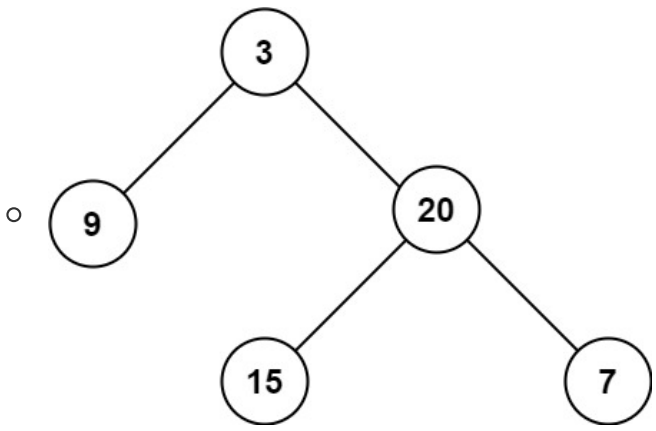
# Solution 2: chunked transfer encoding

```java
public String encode(List<String> strs) {
    StringBuilder sb = new StringBuilder();
    for (String str : strs) {
        sb.append(lenToString(str)).append(str);
    }
    return sb.toString();
}

private String lenToString(String s) {
    int len = s.length();
```

```
    char[] result = new char[4];
    for (int i = 3; i >= 0; i--) {
        result[3 - i] = (char) (len >> (i * 8) & 0xff); // ???
    }
    return new String(result);
}

public List<String> decode(String s) {
    List<String> result = new ArrayList<>();
    int i = 0;
    while (i < s.length()) {
        int len = stringToInt(s.substring(i, i + 4));
        i += 4;
        result.add(s.substring(i, i + len));
        i += len;
    }
    return result;
}

private int stringToInt(String s) {
    int result = 0;
    for (char c : s.toCharArray()) {
        result = (result << 8) + (int)c; // ???
    }
    return result;
}
```

# Tree

## 1. LeetCode 104 Maximum Depth of Binary Tree (easy)

- 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
- **Constraints:**
  - The number of nodes in the tree is in the range `[0, 10^4]`.
  - `-100 <= Node.val <= 100`

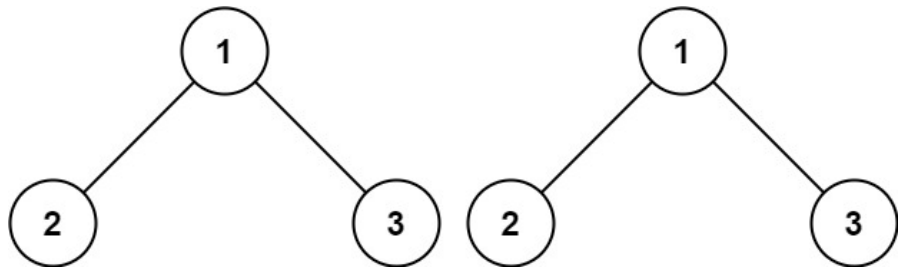
### Solution

```
public int maxDepth(TreeNode root) {
    if (root == null) {
        return 0;
    }
    return Math.max(maxDepth(root.left), maxDepth(root.right)) + 1;
}
```

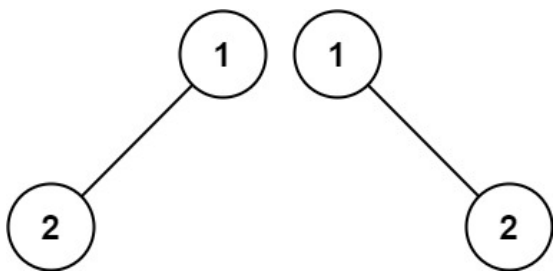
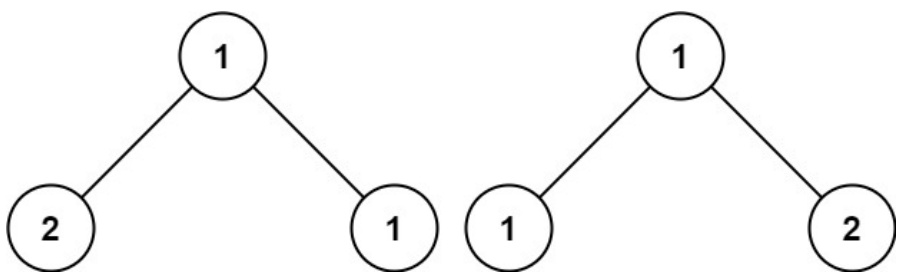
Time Complexity: O(n)

Space Complexity: O(height)

## 2. LeetCode 100 Same Tree (easy)

- Given the roots of two binary trees `p` and `q`, write a function to check if they are the same or not.
- Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.
- **Example 1:**
  - 

  - **Input:** p = [1,2,3], q = [1,2,3]

    - **Output:** true
- **Example 2:**
    - 
    - **Input:** p = [1,2], q = [1,null,2]
    - **Output:** false
- **Example 3:**
    - 
    - **Input:** p = [1,2,1], q = [1,1,2]
    - **Output:** false
- **Constraints:**
    - The number of nodes in both trees is in the range `[0, 100]`.
    - `-10^4 <= Node.val <= 10^4`

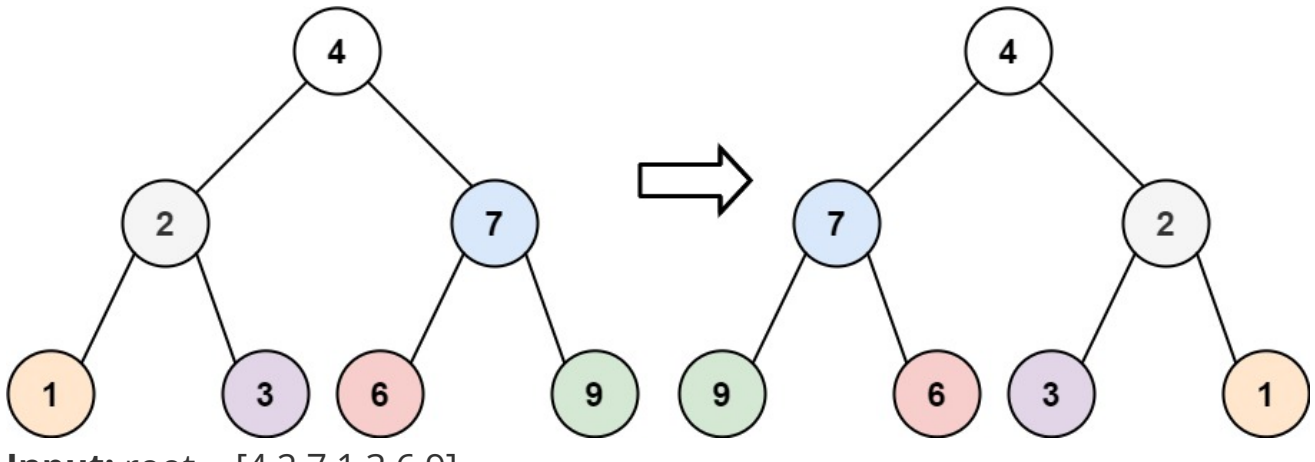
## Solution

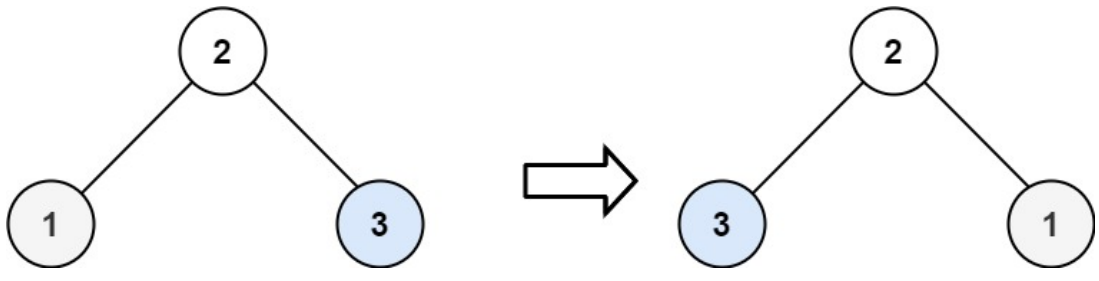
```java
public boolean isSameTree(TreeNode p, TreeNode q) {
    if (p == null && q == null) {
        return true;
    }
    if (p == null || q == null) {
        return false;
    }
    if (p.val != q.val) {
        return false;
    }
    return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
}
```

Time Complexity: O(n)

Space Complexity: O(height)

# 3. LeetCode 226 Invert/Flip Binary Tree (easy)

- Given the `root` of a binary tree, invert the tree, and return *its root*.
- **Example 1:**
    - 
    - **Input:** root = [4,2,7,1,3,6,9]

    - **Output:** [4,7,2,9,6,3,1]
- **Example 2:**
    - 
    - **Input:** root = [2,1,3]
    - **Output:** [2,3,1]
- **Example 3:**
    - **Input:** root = []
    - **Output:** []
- **Constraints:**
    - The number of nodes in the tree is in the range `[0, 100]`.
    - `-100 <= Node.val <= 100`

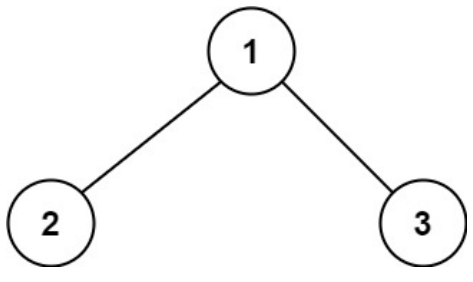
## Solution

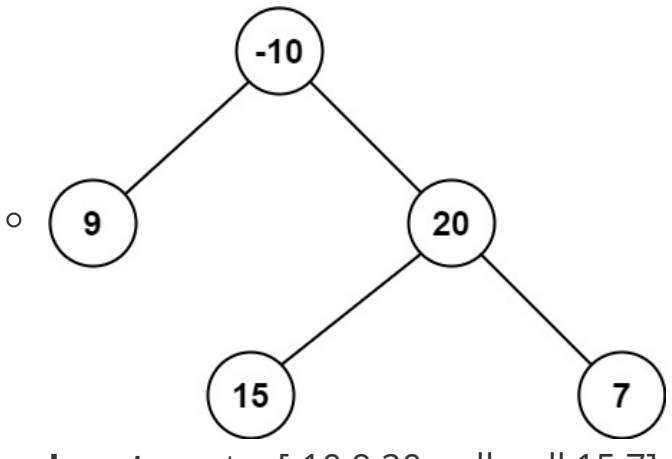
```java
public TreeNode invertTree(TreeNode root) {
    if (root == null) {
        return null;
    }
    TreeNode left = invertTree(root.left);
    TreeNode right = invertTree(root.right);
    root.left = right;
    root.right = left;
    return root;
}
```

Time Complexity: O(n)

Space Complexity: O(height)

# 4. LeetCode 124 Binary Tree Maximum Path Sum (hard)

- A **path** in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence **at most once**. Note that the path does not need to pass through the root.
- The **path sum** of a path is the sum of the node's values in the path.
- Given the `root` of a binary tree, return *the maximum **path sum** of any **non-empty** path*.
- **Example 1:**
    - 
    - **Input:** root = [1,2,3]
    - **Output:** 6
    - **Explanation:** The optimal path is 2 -> 1 -> 3 with a path sum of 2 + 1 + 3 = 6.
- **Example 2:**




    - **Input:** root = [-10,9,20,null,null,15,7]
    - **Output:** 42
    - **Explanation:** The optimal path is 15 -> 20 -> 7 with a path sum of 15 + 20 + 7 = 42.
- **Constraints:**
    - The number of nodes in the tree is in the range `[1, 3 * 10^4]`.
    - `-1000 <= Node.val <= 1000`

# Solution

1. what do you expect from your left-child or right-child?
    - left = max sum of half path in left subtree
    - right = max sum of half path in right subtree
2. what do you want to do in the current layer?
    - check and update result
3. what do you want to report to your parent?
    - return max positive sum of half path or prune

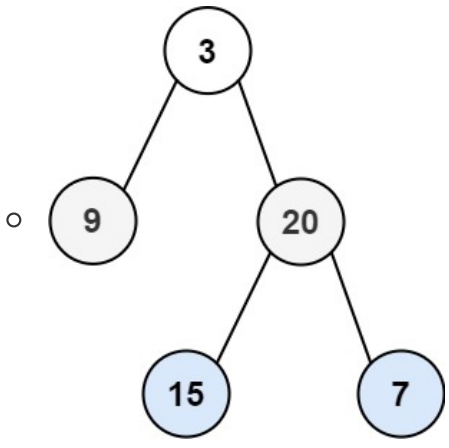
```java
public int maxPathSum(TreeNode root) {
    int[] result = {Integer.MIN_VALUE};
    helper(root, result);
    return result[0];
}

private int helper(TreeNode root, int[] result) {
    if (root == null) {
        return 0;
    }
    int left = helper(root.left, result);
    int right = helper(root.right, result);
    result[0] = Math.max(result[0], left + right + root.val);
    return Math.max(left, right) + root.val < 0 ? 0 : Math.max(left, right) + root.val;
}
```

Time Complexity: O(n)

Space Complexity: O(height)

# 5. LeetCode 102 Binary Tree Level Order Traversal (medium)

- 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:** `[]`
- **Constraints:**
    - The number of nodes in the tree is in the range `[0, 2000]`.
    - `-1000 <= Node.val <= 1000`

## Solution 1: bfs

```java
public List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> result = new ArrayList<>();
    if (root == null) {
        return result;
    }
    Queue<TreeNode> q = new ArrayDeque<>();
    q.offer(root);
    int level = 0;
    while (!q.isEmpty()) {
        result.add(new ArrayList<Integer>());
        int len = q.size();
        for (int i = 0; i < len; i++) {
            TreeNode node = q.poll();
            result.get(level).add(node.val);
            if (node.left != null) {
                q.offer(node.left);
            }
            if (node.right != null) {
                q.offer(node.right);
            }
        }
        level++;
    }
    return result;
}
```

Time Complexity: O(n)

Space Complexity: O(n)

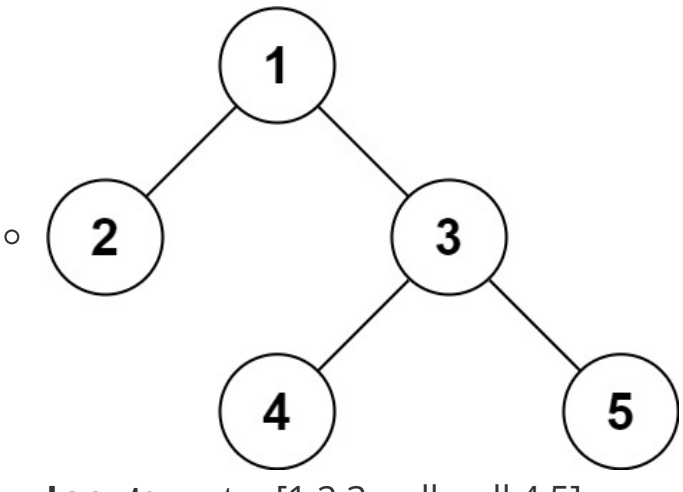
## Solution 2: dfs

```java
public List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> result = new ArrayList<>();
    if (root == null) {
        return result;
    }
    helper(root, 0, result);
    return result;
}

private void helper(TreeNode root, int level, List<List<Integer>> result) {
    if (result.size() == level) {
        result.add(new ArrayList<Integer>());
    }
    result.get(level).add(root.val);
    if (root.left != null) {
        helper(root.left, level + 1, result);
    }
    if (root.right != null) {
        helper(root.right, level + 1, result);
    }
}
```

Time Complexity: O(n)

Space Complexity: O(n)

## 6. LeetCode 297 Serialize and Deserialize Binary Tree (hard)

- Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.
- Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.
- **Clarification:** The input/output format is the same as how LeetCode serializes a binary tree. You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.
- **Example 1:**
  - 

  - **Input:** root = [1,2,3,null,null,4,5]
  - **Output:** [1,2,3,null,null,4,5]
- **Example 2:**

- **Input:** root = []
- **Output:** []

- **Constraints:**
  - The number of nodes in the tree is in the range `[0, 10^4]`.
  - `-1000 <= Node.val <= 1000`

# Solution 1: level-order iterative

```java
public String serialize(TreeNode root) {
    if (root == null) {
        return "#";
    }
    StringBuilder sb = new StringBuilder();
    Queue<TreeNode> q = new LinkedList<>(); // LinkedList to store null;
    q.offer(root);
    while (!q.isEmpty()) {
        TreeNode node = q.poll();
        if (node == null) {
            sb.append("# ");
            continue;
        }
        sb.append(node.val + " ");
        q.offer(node.left);
        q.offer(node.right);
    }
    return sb.toString();
}

public TreeNode deserialize(String data) {
    if (data == "#") {
        return null;
    }
    String[] array = data.split(" ");
    TreeNode root = new TreeNode(Integer.parseInt(array[0]));
    Queue<TreeNode> q = new LinkedList<>();
    q.offer(root);
    for (int i = 1; i < array.length; i++) {
        TreeNode parent = q.poll();
        if (!array[i].equals("#")) {
            TreeNode left = new TreeNode(Integer.parseInt(array[i]));
            parent.left = left;
            q.offer(left);
        }
        i++;
        if (!array[i].equals("#")) {
            TreeNode right = new TreeNode(Integer.parseInt(array[i]));
            parent.right = right;
            q.offer(right);
        }
    }
    return root;
}
```

Time Complexity: O(n)

Space Complexity: O(n)

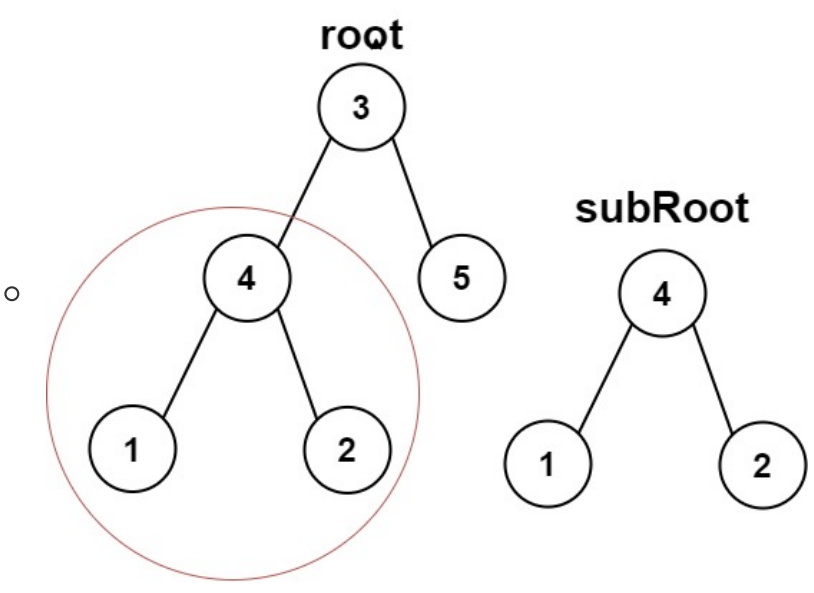
## Solution 2: pre-order recursive

```java
public String serialize(TreeNode root) {
    if (root == null) {
        return "#";
    }
    return root.val + "," + serialize(root.left) + "," + serialize(root.right);
}

public TreeNode deserialize(String data) {
    Queue<String> q = new LinkedList<>(Arrays.asList(data.split(",")));
    return helper(q);
}

private TreeNode helper(Queue<String> q) {
    String s = q.poll();
    if (s.equals("#")) {
        return null;
    }
    TreeNode root = new TreeNode(Integer.parseInt(s));
    root.left = helper(q);
    root.right = helper(q);
    return root;
}
```

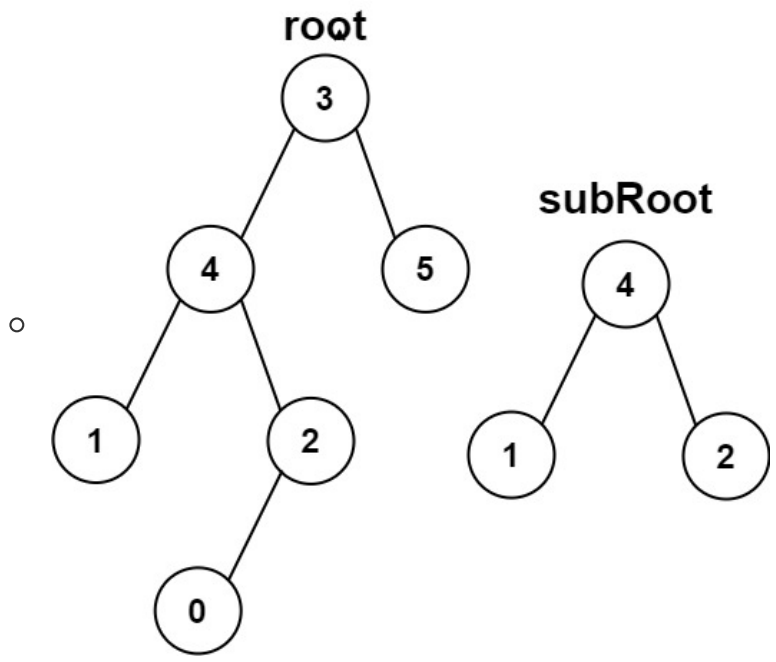
Time Complexity: O(n)

Space Complexity: O(n)

## 7. LeetCode 572 Subtree of Another Tree (easy)

- Given the roots of two binary trees `root` and `subRoot`, return `true` if there is a subtree of `root` with the same structure and node values of `subRoot` and `false` otherwise.
- A subtree of a binary tree `tree` is a tree that consists of a node in `tree` and all of this node's descendants. The tree `tree` could also be considered as a subtree of itself.
- **Example 1:**
  - 

  - **Input:** root = [3,4,5,1,2], subRoot = [4,1,2]
  - **Output:** true
- **Example 2:**




    - **Input:** root = [3,4,5,1,2,null,null,null,null,0], subRoot = [4,1,2]
    - **Output:** false
- **Constraints:**
    - The number of nodes in the `root` tree is in the range `[1, 2000]`.
    - The number of nodes in the `subRoot` tree is in the range `[1, 1000]`.
    - `-10^4 <= root.val <= 10^4`
    - `-10^4 <= subRoot.val <= 10^4`

## Solution

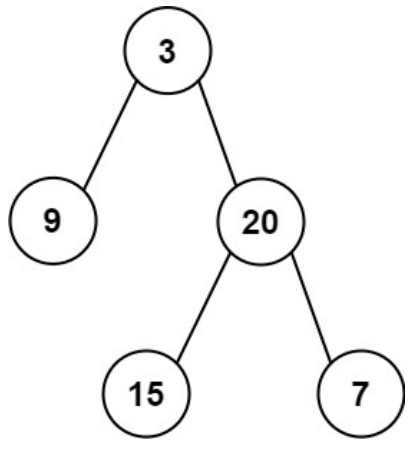
```java
public boolean isSubtree(TreeNode root, TreeNode subRoot) {
    if (root == null) {
        return subRoot == null;
    }
    return isSameTree(root, subRoot) || isSubtree(root.left, subRoot) ||
isSubtree(root.right, subRoot);
}

private boolean isSameTree(TreeNode p, TreeNode q) {
    if (p == null && q == null) {
        return true;
    }
    if (p == null || q == null) {
        return false;
    }
    if (p.val != q.val) {
        return false;
    }
    return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
}
```

Time Complexity: O(n)

Space Complexity: O(height)

# 8. LeetCode 105 Construct Binary Tree from Preorder and Inorder Traversal (medium)

- Given two integer arrays `preorder` and `inorder` where `preorder` is the preorder traversal of a binary tree and `inorder` is the inorder traversal of the same tree, construct and return *the binary tree*.
- **Example 1:**




    - **Input:** preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
    - **Output:** [3,9,20,null,null,15,7]
- **Example 2:**
    - **Input:** preorder = [-1], inorder = [-1]
    - **Output:** [-1]
- **Constraints:**
    - `1 <= preorder.length <= 3000`
    - `inorder.length == preorder.length`
    - `-3000 <= preorder[i], inorder[i] <= 3000`
    - `preorder` and `inorder` consist of **unique** values.
    - Each value of `inorder` also appears in `preorder`.
    - `preorder` is **guaranteed** to be the preorder traversal of the tree.
    - `inorder` is **guaranteed** to be the inorder traversal of the tree.

## Solution

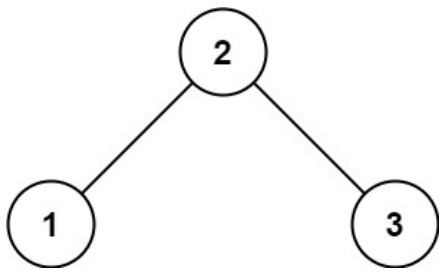
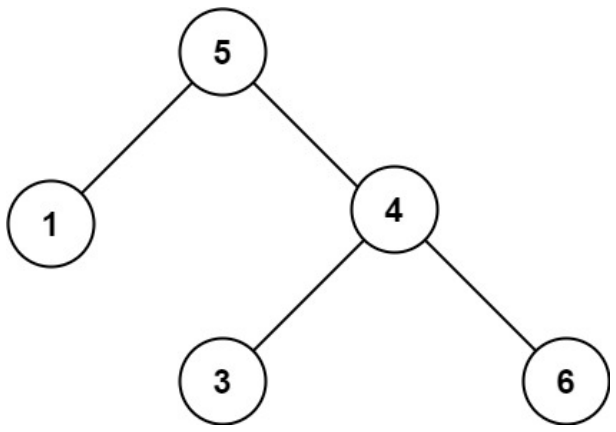
```java
public TreeNode buildTree(int[] preorder, int[] inorder) {
    Map<Integer, Integer> map = new HashMap<>();
    for (int i = 0; i < inorder.length; i++) {
        map.put(inorder[i], i);
    }
    return helper(preorder, map, 0, inorder.length - 1, 0, preorder.length - 1);
}

private TreeNode helper(int[] preorder, Map<Integer, Integer> map, int inLeft, int inRight, int preLeft, int preRight) {
    if (inLeft > inRight || preLeft > preRight) {
        return null;
    }
    TreeNode root = new TreeNode(preorder[preLeft]);
    int mid = map.get(root.val);
    root.left = helper(preorder, map, inLeft, mid - 1, preLeft + 1, preLeft + mid - inLeft);
    root.right = helper(preorder, map, mid + 1, inRight, preRight + mid - inRight + 1, preRight);
    return root;
}
```

Time Complexity: O(n)

Space Complexity: O(n)

# 9. LeetCode 98 Validate Binary Search Tree (medium)

- 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.
- **Constraints:**
  - The number of nodes in the tree is in the range `[1, 10^4]`.
  - `-2^31 <= Node.val <= 2^31 - 1`

## Solution 1: in-order recursive

```java
public boolean isValidBST(TreeNode root) {
    Integer[] previous = {null};
    return helper(root, previous);
}

private boolean helper(TreeNode root, Integer[] previous) {
    if (root == null) {
        return true;
    }
    if (!helper(root.left, previous)) {
        return false;
    }
    if (previous[0] != null && previous[0] >= root.val) {
        return false;
    }
    previous[0] = root.val;
    return helper(root.right, previous);
}
```

Time Complexity: O(n)

Space Complexity: O(n)

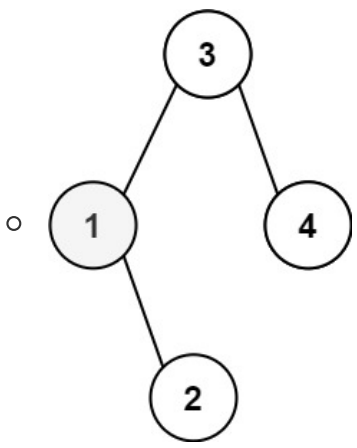
## Solution 2: in-order iterative

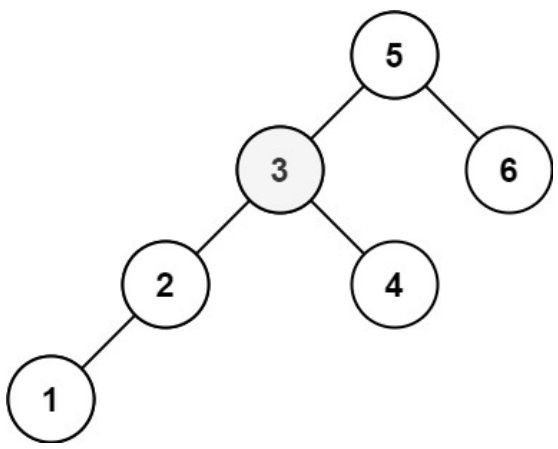
```java
public boolean isValidBST(TreeNode root) {
    if (root == null) {
        return true;
    }
    Deque<TreeNode> stack = new ArrayDeque<>();
    TreeNode helper = root;
    Integer previous = null;
    while (helper != null || !stack.isEmpty()) {
        if (helper != null) {
            stack.offerFirst(helper);
            helper = helper.left;
        } else {
            helper = stack.pollFirst();
            if (previous != null && previous >= helper.val) {
                return false;
            }
            previous = helper.val;
            helper = helper.right;
        }
    }
    return true;
}
```

Time Complexity: O(n)

Space Complexity: O(n)

## 10. LeetCode 230 Kth Smallest Element in a BST (medium)

- Given the `root` of a binary search tree, and an integer `k`, return *the* `kth` *smallest value (**1-indexed**) of all the values of the nodes in the tree*.
- **Example 1:**
  - 

  - **Input:** root = [3,1,4,null,2], k = 1
  - **Output:** 1
- **Example 2:**
  - 

  - **Input:** root = [5,3,6,2,4,null,null,1], k = 3

- **Output:** 3

- **Constraints:**
  - The number of nodes in the tree is `n`.
  - `1 <= k <= n <= 10^4`
  - `0 <= Node.val <= 10^4`
- **Follow up:** If the BST is modified often (i.e., we can do insert and delete operations) and you need to find the kth smallest frequently, how would you optimize?

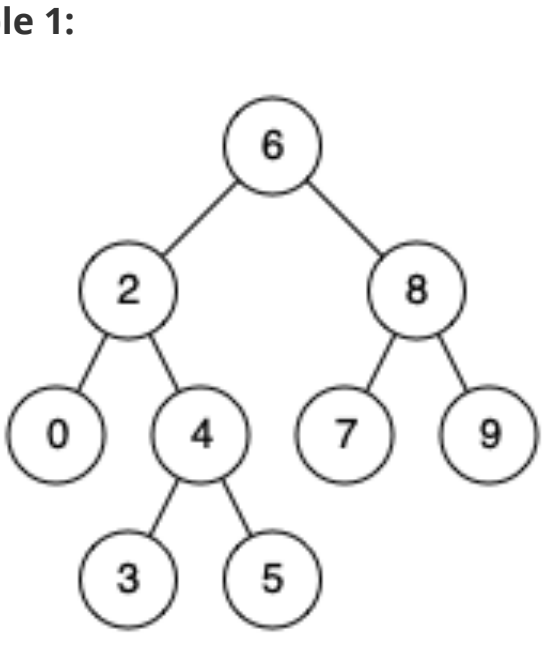
## Solution

```java
public int kthSmallest(TreeNode root, int k) {
    Deque<TreeNode> stack = new ArrayDeque<>();
    while (true) {
        while (root != null) {
            stack.offerFirst(root);
            root = root.left;
        }
        root = stack.pollFirst();
        if (--k == 0) {
            return root.val;
        }
        root = root.right;
    }
}
```

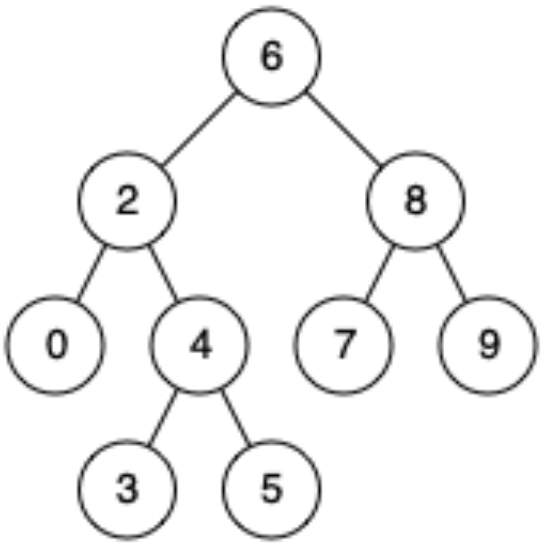
Time Complexity: O(height + k)

Space Complexity: O(height)

## 11. LeetCode 235 Lowest Common Ancestor of BST (easy)

- Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.
- According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes `p` and `q` as the lowest node in `T` that has both `p` and `q` as descendants (where we allow **a node to be a descendant of itself**).”
- **Example 1:**
  - 

  - **Input:** root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
  - **Output:** 6
  - **Explanation:** The LCA of nodes 2 and 8 is 6.
- **Example 2:**

- 

    - **Input:** root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
    - **Output:** 2
    - **Explanation:** The LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition.
- **Example 3:**
    - **Input:** root = [2,1], p = 2, q = 1
    - **Output:** 2
- **Constraints:**
    - The number of nodes in the tree is in the range `[2, 10^5]`.
    - `-10^9 <= Node.val <= 10^9`
    - All `Node.val` are **unique**.
    - `p != q`
    - `p` and `q` will exist in the BST.

## Solution 1

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    if (root == null || root == p || root == q) {
        return root;
    }
    TreeNode left = lowestCommonAncestor(root.left, p, q);
    TreeNode right = lowestCommonAncestor(root.right, p, q);
    if (left != null && right != null) {
        return root;
    }
    return left == null ? right : left;
}
```

Time Complexity: O(n)

Space Complexity: O(height)

## Soltuion 2

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    int small = Math.min(p.val, q.val);
    int large = Math.max(p.val, q.val);
    while (root != null) {
        if (root.val < small) {
            root = root.right;
        } else if (root.val > large) {
            root = root.left;
```

```
        } else {
            return root;
        }
    }
    return null;
}
```

Time Complexity: O(n)

Space Complexity: O(1)

# 12. LeetCode 208 Implement Trie (Prefix Tree) (medium)

- A **trie** (pronounced as "try") or **prefix tree** is a tree data structure used to efficiently store and retrieve keys in a dataset of strings. There are various applications of this data structure, such as autocomplete and spellchecker.
- Implement the Trie class:
  - `Trie()` Initializes the trie object.
  - `void insert(String word)` Inserts the string `word` into the trie.
  - `boolean search(String word)` Returns `true` if the string `word` is in the trie (i.e., was inserted before), and `false` otherwise.
  - `boolean startsWith(String prefix)` Returns `true` if there is a previously inserted string `word` that has the prefix `prefix`, and `false` otherwise.
- **Example 1:**
  - **Input**
    - ["Trie", "insert", "search", "search", "startsWith", "insert", "search"]
    - `[[], ["apple"], ["apple"], ["app"], ["app"], ["app"], ["app"]]`
  - **Output**
    - [null, null, true, false, true, null, true]
  - **Explanation**

```
Trie trie = new Trie();
trie.insert("apple");
trie.search("apple");   // return True
trie.search("app");     // return False
trie.startsWith("app"); // return True
trie.insert("app");
trie.search("app");     // return True
```

- **Constraints:**
  - `1 <= word.length, prefix.length <= 2000`
  - `word` and `prefix` consist only of lowercase English letters.
  - At most `3 * 10^4` calls **in total** will be made to `insert`, `search`, and `startsWith`.

# Solution

```
class Trie {
    static class TrieNode {
        Map<Character, TrieNode> children;
        boolean isWord;
        int count;

```

```
        public TrieNode() {
            children = new HashMap<>();
        }
    }

    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    public void insert(String word) {
        if (search(word)) {
            return;
        }
        TrieNode current = root;
        for (int i = 0; i < word.length(); i++) {
            TrieNode next = current.children.get(word.charAt(i));
            if (next == null) {
                next = new TrieNode();
                current.children.put(word.charAt(i), next);
            }
            current = next;
            current.count++;
        }
        current.isWord = true;
    }

    public boolean search(String word) {
        TrieNode current = root;
        for (int i = 0; i < word.length(); i++) {
            TrieNode next = current.children.get(word.charAt(i));
            if (next == null) {
                return false;
            }
            current = next;
        }
        return current.isWord;
    }

    public boolean startsWith(String prefix) {
        TrieNode current = root;
        for (int i = 0; i < prefix.length(); i++) {
            TrieNode next = current.children.get(prefix.charAt(i));
            if (next == null) {
                return false;
            }
            current = next;
        }
        return true;
    }
}
```

Time Complexity: O(length)

## 13. LeetCode 211 Add and Search Word (medium)

- Design a data structure that supports adding new words and finding if a string matches any previously added string.
- Implement the `WordDictionary` class:
  - `WordDictionary()` Initializes the object.
  - `void addWord(word)` Adds `word`to the data structure, it can be matched later.
  - `bool search(word)` Returns `true` if there is any string in the data structure that matches `word` or `false` otherwise. `word` may contain dots `'.'` where dots can be matched with any letter.
- **Example:**
  - **Input**
    - ["WordDictionary","addWord","addWord","addWord","search","search","search","search"]
    - `[[],["bad"],["dad"],["mad"],["pad"],["bad"],[".ad"],["b.."]]`
  - **Output**
    - [null,null,null,null,false,true,true,true]
  - **Explanation**

```
WordDictionary wordDictionary = new WordDictionary();
wordDictionary.addWord("bad");
wordDictionary.addWord("dad");
wordDictionary.addWord("mad");
wordDictionary.search("pad"); // return False
wordDictionary.search("bad"); // return True
wordDictionary.search(".ad"); // return True
wordDictionary.search("b.."); // return True
```

- **Constraints:**
  - `1 <= word.length <= 25`
  - `word` in `addWord` consists of lowercase English letters.
  - `word` in `search` consist of `'.'` or lowercase English letters.
  - There will be at most `3` dots in `word` for `search` queries.
  - At most `10^4` calls will be made to `addWord` and `search`.

# Solution

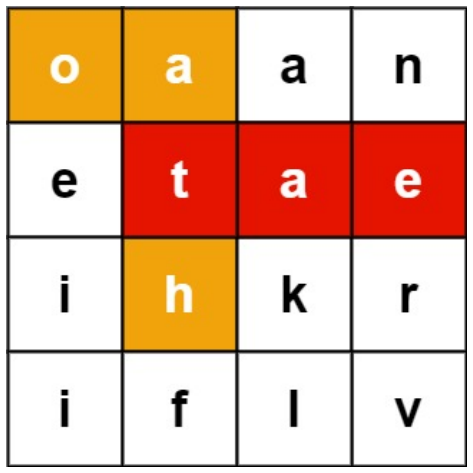
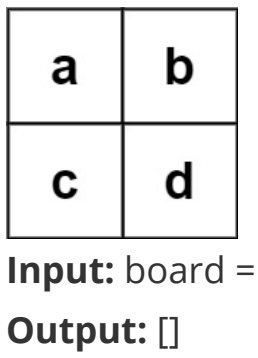
```
class WordDictionary {
    static class TrieNode {
        Map<Character, TrieNode> children;
        boolean isWord;
        int count;

        public TrieNode() {
            children = new HashMap<>();
        }
    }

    private TrieNode root;

    public WordDictionary() {
        root = new TrieNode();
    }

```

```java
    public void addWord(String word) {
        if (search(word)) {
            return;
        }
        TrieNode current = root;
        for (int i = 0; i < word.length(); i++) {
            TrieNode next = current.children.get(word.charAt(i));
            if (next == null) {
                next = new TrieNode();
                current.children.put(word.charAt(i), next);
            }
            current = next;
            current.count++;
        }
        current.isWord = true;
    }

    public boolean search(String word) {
        return search(word, root);
    }

    private boolean search(String word, TrieNode current) {
        for (int i = 0; i < word.length(); i++) {
            if (!current.children.containsKey(word.charAt(i))) {
                if (word.charAt(i) == '.') {
                    for (char c : current.children.keySet()) {
                        TrieNode next = current.children.get(c);
                        if (search(word.substring(i + 1), next)) {
                            return true;
                        }
                    }
                }
                return false;
            } else {
                current = current.children.get(word.charAt(i));
            }
        }
        return current.isWord;
    }
}
```

Time Complexity: O(length)

## 14. LeetCode 212 Word Search II (hard)

- Given an `m x n` `board` of characters and a list of strings `words`, return *all words on the board*.
- Each word must be constructed from letters of sequentially adjacent cells, where **adjacent cells** are horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.
- **Example 1:**

  - 
  - **Input:** board = `[["o","a","a","n"],["e","t","a","e"],["i","h","k","r"],["i","f","l","v"]]`, words = ["oath","pea","eat","rain"]
  - **Output:** ["eat","oath"]
- **Example 2:**
  - 
  - **Input:** board = `[["a","b"],["c","d"]]`, words = ["abcb"]
  - **Output:** []
- **Constraints:**
  - `m == board.length`
  - `n == board[i].length`
  - `1 <= m, n <= 12`
  - `board[i][j]` is a lowercase English letter.
  - `1 <= words.length <= 3 * 10^4`
  - `1 <= words[i].length <= 10`
  - `words[i]` consists of lowercase English letters.
  - All the strings of `words` are unique.

## Solution: trie

```java
class Solution {
    // assumption: words[i] consists of lowercase English letters
    static class TrieNode {
        TrieNode[] children = new TrieNode[26];
        String word;
    }

    public List<String> findWords(char[][] board, String[] words) {
        List<String> result = new ArrayList<>();
        TrieNode root = new TrieNode();
        for (String word : words) {
            TrieNode current = root;
            for (char c : word.toCharArray()) {
                int index = c - 'a';
                if (current.children[index] == null) {
                    current.children[index] = new TrieNode();
                }
                current = current.children[index];
            }
            current.word = word;
        }
        for (int i = 0; i < board.length; i++) {
```

```
            for (int j = 0; j < board[0].length; j++) {
                helper(board, i, j, root, result);
            }
        }
        return result;
    }

    private void helper(char[][] board, int i, int j, TrieNode node, List<String> result)
{
        char c = board[i][j];
        if (c == '#' || node.children[c - 'a'] == null) {
            return;
        }
        node = node.children[c - 'a'];
        if (node.word != null) {
            result.add(node.word);
            node.word = null;
        }
        board[i][j] = '#';
        if (i > 0) {
            helper(board, i - 1, j, node, result);
        }
        if (i < board.length - 1) {
            helper(board, i + 1, j, node, result);
        }
        if (j > 0) {
            helper(board, i, j - 1, node, result);
        }
        if (j < board[0].length - 1) {
            helper(board, i, j + 1, node, result);
        }
        board[i][j] = c;
    }
}
```

Time Complexity: O(mn * 4^L)

Space Complexity: O(wL)