Technical Interview

Michael Fong

2020-04-05

Contents

Chapter 1

Introduction

1.1 Glossary of Algorithm

1.1.1 Depth First Traversal

: An algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node and explores as far as possible along each branch recursively.

1.1.2 Breadth First Traversal}

: An algorithm for traversing or searching tree or graph data structures. It starts at the tree root, and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

1.1.3 Topological Sorting

: Topological Sorting

1.1.4 Minimax

: (sometimes MinMax) is an algorithm for minimizing the possible loss for a worst case (maximum loss) scenario.

1.1.5 Recursion

: a method of solving a problem where the solution depends on solutions to smaller instances of the same problem

1.1.6 Backtrack

: a method trying to construct a solution to a computational problem incrementally, one small piece at a time. Whenever the algorithm needs to decide between multiple alternatives to the next component of the solution, it recursively evaluates **every** alternative and then chooses the best one.

1.1.7 Dynamic Programming

: a method to to simplify a complicated problem by breaking it down into simpler sub-problems in a recursive manner. You have a main problem (the root of your tree of subproblems), and subproblems (subtrees). The subproblems typically repeat and overlap. Common approach is implemented recursively or iteratively table-filling.

1.1.8 Memoization

: an optimization technique used to speed up programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again. The implementation can be of form recursive call (or some iterative equivalent)

1.1.9 Tabulation

: is an approach where you solve a dynamic programming problem by first filling up a table, and then compute the solution to the original problem based on the results in this table.

1.1.10 Divide and Conquer

: an algorithm to recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

1.1.11 Binary Search Tree

: a binary tree where nodes are ordered, where the keys in the left subtree are less then the key in its parent node, and the keys in the right subtree are greater the key in its parent node.

1.1.12 Merge sort

: a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves.

1.1.13 Bubble sort

: a sorting algorithm that compares adjacent pairs and swaps them if necessary, causing the items to "bubble" up toward their proper position.

1.1.14 Quick Sort

: a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot.

1.1.15 Minimum Heap

: a complete binary tree in which the value in each internal node is smaller than or equal to the values in the children of that node. Thus, the root node is the smallest element in the tree.

1.2 Dev Guide

1.3 Problem

1.3.1 Description

1.3.2 Example

1.3.3 Solution

1.3.3.1 Walkthrough

1.3.3.2 Analysis

1.3.3.3 Algorithm

1.3.4 Java Code

Chapter 2

Arrays

If we are enumerating the cartesian products over two arrays of n elements.

```
A = [A_1, A_2, ..., A_n]
B = [B_1, B_2, ..., B_n]
A1 X A2 = [(A_1, B_1), (A_1, B_2), ..., (A_n, B_n)]
```

The time complexity is O(n^2).

```
for(int i = 0; i < A.length; i++) {
    for(int j = 0; j < B.length; j++) {
        // process each (A_i, B_j)
    }
}
```

If we only need to enumerate part of the combination over two arrays while traversing both arrays simultaneously.

```
A = [A_1, A_2, ..., A_n]
B = [B_1, B_2, ..., B_n]
A1 X A2 = [(A_1, B_1), (A_2, B_1), (A_2, B_2), ..., (A_n, B_n)]
```

we could possibly reduce the complexity to O(n)

```
for(int i = 0, j = 0; i < A.length; j < B.length;) {
    int a = A[i];
    int b = B[j];

    if(a < b) {
        i++;
    } else if(a > b) {
        j++;
    }
```

```
}
```

2.1 Two Sums I / LeetCode 1 / Easy

2.1.1 Description

Given an array of integers, return indices of the two numbers such that they add up to a specific target. You may assume that each input would have exactly one solution, and you may not **use the same element twice**.

2.1.2 Examples

Given nums = [2, 7, 11, 15], target = 9. Because nums[0] + nums[1] = 2 + 7 = 9, return [0, 1].

2.1.3 Solution - with HashTable

2.1.3.1 Walkthrough

Use a HashMap to store each element and its associated index, that is <nums[i], i>. Then check if diff = (target - nums[i]) and return the pair of indices if existed.

2.1.3.2 Analysis

Each insertion and get operation from hashmap takes O(1) and there are n elements. The total time complexity costs O(n) and Auxiliary Space takes $O(n)$

2.1.3.3 Algorithm

2.1.4 Java Code - with HashTable

```
public int[] twoSum(int[] nums, int target) {
    Map<Integer, Integer> map = new HashMap<Integer, Integer>();

    //iterate through the indices
    for(int i = 0; i < nums.length; i++) {
        int diff = target - nums[i];
        Integer diffIdx = map.get(diff);
```

```
        if(map.containsKey(diff) && i != diffIdx) {
            //map contains diff value and its associated index
            return new int[]{i, diffIdx};
        } else {
            //store the value and index for later use
            map.put(nums[i], i);
        }
    }

    return null;
}
```

2.1.5 Solution - Searching Target Sum from Sorted Array

2.1.5.1 Walkthrough

This solution is valid only on **sorted** array. Have two indices pointed to left most and right most position of array. Start comparing the sum against the target. If sum meets, return the indices; otherwise, move indices inward.

2.1.5.2 Analysis

Time complexity is O(n) since every element is visited, and Auxiliary Space is O(1)

2.1.5.3 Algorithm

2.1.6 Java Code - Searching Target Sum from Sorted Array

```
public int[] twoSum(int[] nums, int target) {
    int left = 0, right = nums.length - 1;

    while(left < right) {
        int sum = nums[left] + nums[right];

        if(sum == target) {
            return new int[]{left, right};
        }

        if(target > sum) {
```

```
            //increase value by shifting rightward
            left++;
        } else {
            //decrease value by shifting leftward
            right--;
        }
    }

    return null;
}
```

2.2 Two Sum II - Return True / False

2.2.1 Description

You are given an array of n integers and a number k. Determine whether there is a pair of elements in the array that sums to exactly k.

2.2.2 Example

For example, given the array [1, 3, 7] and k = 8, the answer is "yes," but given k = 6 the answer is "no."

2.2.3 Solution

2.2.3.1 Walkthrough

Have a HashSet to iterate through the array while compute the difference (target - a[i]). Check if the diff exist. Return true if exists; false otherwise. HashSet essentially uses less space than HashTable.

2.2.3.2 Analysis

Each insertion and get operation from hashmap takes O(1) and there are n elements. The total time complexity costs O(n) and Auxiliary Space takes O(n)

2.2.3.3 Algorithm

2.2.4 Java Code

```
public boolean sumsToTarget(int[] arr, int target) {
    Set<Integer> values = new HashSet<Integer>();

    for (int i = 0; i < arr.length; i++) {
        int diff = target - arr[i];

        //if hash set contains the difference
        if (values.contains(diff)) {
            return true;
        } else {
            values.add(arr[i]);
        }
    }

    return false;
}
```

2.3 Two Sum IV - Input is a BST / LeetCode 653 / Easy

2.3.1 Description

Given a Binary Search Tree and a target number, return true if there exist two elements in the BST such that their sum is equal to the given target.

2.3.2 Example

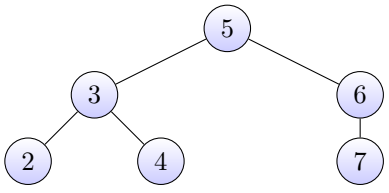


Figure 2.1: Some caption.

Target = 9, Output = True.

2.3.3 Solution

2.3.3.1 Walkthrough

Use a HashSet to store value of current node, that is (node.val). Then check if answer = (target - node.val) and return true if existed; otherwise, recursively call the same function for its left and right children.

2.3.3.2 Analysis

Time complexity is O(n) since every node is visited. Auxiliary Space is O(1)

2.3.3.3 Algorithm

recursion 1.1.5

2.3.4 Java Code

```
public boolean findTarget(TreeNode root, int target) {
    Set<Integer> set = new HashSet<>();
    return findTarget(root, target, set);
}

public boolean findTarget(TreeNode node, int target, Set<Integer> set) {
    if(node == null) {
        return false;
    } else if(set.contains(target - node.val)) {
        return true;
    } else {
        //recursively calling
        set.add(node.val);
        return findTarget(node.right, target, set) || findTarget(node.left, target, se
    }

}
```

2.4 TwoSum V / / }

2.4.1 Description

Given a sorted array S of n integers, are there elements a, b in S such that a + b = target? Find all unique pairs in the array which gives the sum of zero. Note: The solution set must not contain duplicate triplets.

2.4.2 Example

For example, given array S = [-2, -1, -1, 0, 1, 2], target = 0 A solution set is: [[-1, 1], [-2, 2]]

2.4.3 Solution

2.4.3.1 Walkthrough

While searching for the other two elements from both ends, with indices left incrementally and right decrmentally, find the sum = nums[left] + nums[right] == target. Since the array is sorted, we need to avoid duplicated entry by moving forward the index while searching.

2.4.3.2 Analysis

There are one loops, the time complexity is $O(n)$, and Auxiliary Space is $O(n)$ since one member of any pair is uniquely determined by the other member. If the numbers are not distinct, the Auxiliary Space as large as $O(\binom{n}{2}) = O(\frac{n!}{2!\cdot(n-2)!}) = O(\frac{n\cdot(n-1)}{2}) = O(n^2)$

2.4.3.3 Algorithm

2.4.4 Java Code

```
public List<List<Integer>> twoSum(int[] num, int target) {
    List<List<Integer>> result = new ArrayList<>();

    int left = 0, right = num.length - 1;

    while (left < right) {
        int sum = num[left] + num[right];
```

```
        if (sum == target) {
            result.add(Arrays.asList(num[left], num[right]));

            //avoid duplicated entry by moving forward the index
            while (left < right && num[left] == num[left + 1]) {
                left++;
            }
            while (left < right && num[right] == num[right - 1]) {
                right--;
            }

            left++;
            right--;
        } else if (sum < target) {
            left++;
        } else {
            // sum > target
            right--;
        }
    }

    return result;
}
```

2.5 3Sum / Leet Code 15 / Medium}

2.5.1 Description

Given an array S of n integers, are there elements a, b, c in S such that a + b + c = 0? Find all unique triplets in the array which gives the sum of zero. Note: The solution set must not contain duplicate triplets.

2.5.2 Example

For example, given array S = [-1, 0, 1, 2, -1, -4], A solution set is: [[-1, 0, 1], [-1, -1, 2]]

2.5.3 Solution

2.5.3.1 Walkthrough

Looping the array with index i, while searching for the other two elements from both ends, with indices left incrementally and right decrmentally, find the sum = nums[i] + nums[left] + nums[right] == 0. Since the array is sorted, we need to avoid duplicated entry by moving forward the index while searching.

2.5.3.2 Analysis

There are two nested loops, the time complexity is $O(n^2)$, and Auxiliary Space is is $O(n^2)$ since one member of any triplet is uniquely determined by the other two members. If the numbers are not distinct, the Auxiliary Space as large as $O(\binom{n}{3}) = O(\frac{n!}{3!\cdot(n-3)!}) = O(\frac{n\cdot(n-1)\cdot(n-2)}{3}) = O(n^3)$

2.5.3.3 Algorithm

2.5.4 Java Code

```
public List<List<Integer>> threeSum(int[] num) {
    Arrays.sort(num); //sort
    List<List<Integer>> result = new ArrayList<>();

    //last possible pair is [i=len - 3, left=len - 2, right=len - 1]
    for (int i = 0; i < num.length - 2; i++) {
        // Since the array is sorted, we need to avoid duplicated entry by moving forward the ind
        if (i == 0 || (i > 0 && num[i] != num[i-1])) {
            int left = i + 1, right = num.length-1;

            while (left < right) {
                int sum = num[i] + num[left] + num[right];

                if (sum == 0) {
                    result.add(Arrays.asList(num[i], num[left], num[right]));

                    //avoid duplicated entry by moving forward the index
                    while (left < right && num[left] == num[left + 1]) {
                        left++;
                    }
                    while (left < right && num[right] == num[right - 1]) {
                        right--;
                    }
```

```
                    left++;
                    right--;
                } else if (sum < 0) {
                    left++;
                } else {
                    // sum > 0
                    right--;
                }
            }
        }
    }

    return result;
}
```

2.6 3 Sum Closest / LeetCode 16 / Medium}

2.6.1 Description

Given an array S of n integers, find three integers in S such that the sum is closest to a given number, target. Return the sum of the three integers. You may assume that each input would have exactly one solution.

2.6.2 Example

For example, given array S = -1 2 1 -4, and target = 1. The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

2.6.3 Solution

2.6.3.1 Walkthrough

Looping the array with index i, while searching for the other two elements from both ends, with indices left incrementally and right decrmentally, find the sum = nums[i] + nums[left] + nums[right]. Return sum if existed; otherwise, keep for the closest tuple of (nums[i], nums[left], nums[right]) against target using Math.abs().

2.6.3.2 Analysis

There are two nested loops, the time complexity is $O(n^2)$, and Auxiliary Space is O(1) since it is returning the closest answer.

2.6.3.3 Algorithm

2.6.4 Java Code

```
public int threeSumClosest(int[] nums, int target) {
    Arrays.sort(num); //sort
    int minDelta = Integer.MAX_VALUE, result = 0;

    //last possible pair is [i=len - 3, left=len - 2, right=len - 1]
    for (int i = 0; i < num.length - 2; i++) {
        int left = i + 1, right = num.length - 1;

        while (left < right) {
            int sum = num[i] + num[left] + num[right];

            if (sum == target) {
                //find exact match
                return sum;
            } else if (sum < target) {
                left++;
            } else {
                // sum > target
                right--;
            }

            int delta = Math.abs(sum - target);
            if(delta < minDelta) {
                minDelta = delta;
                result = sum;
            }
        }
    }

    return result;
}
```

2.7 4Sum / Leet Code 18 / Medium}

2.7.1 Description

Given an array nums of n integers and an integer target, are there elements a, b, c, and d in nums such that a + b + c + d = target? Find all unique quadruplets in the array which gives the sum of target. The solution set must not contain duplicate quadruplets.

2.7.2 Example

Given array nums = [1, 0, -1, 0, -2, 2], and target = 0.

A solution set is: [[-1, 0, 0, 1], [-2, -1, 1, 2], [-2, 0, 0, 2]]

2.7.3 Solution

2.7.3.1 Walkthrough

The solution is similar to 3Sum to find the sum = nums[i] + nums[j] + nums[left] + nums[right] == target. The only difference is there is an extra loop as well as we need to avoid duplicated entry by moving forward the index during search.

2.7.3.2 Analysis

Overall, there are three nested loops, the time complexity is $O(n^3)$, and Auxiliary Space is $O(n^3)$ since one member of any quotriplet is uniquely determined by the other three member. If the numbers are not distinct, the Auxiliary Space as large as is $O(\binom{n}{4}) = O(n^4)$

2.7.3.3 Algorithm

2.7.4 Java Code

```
public List<List<Integer>> fourSum(int[] num, int target) {
    Arrays.sort(num); //sort
    List<List<Integer>> res = new LinkedList<>();

    //last possible pair is [j=len - 4, i=len-3, left=len - 2, right=len - 1]
    for (int j = 0; i < num.length - 3; i++) {
```

```
        // Since the array is sorted, we need to avoid duplicated entry by moving forward the ind
        if (j == 0 || (j > 0 && num[j] != num[j - 1])) {
            for (int i = j + 1; i < num.length - 2; i++) {

                // Since the array is sorted, we need to avoid duplicated entry by moving forward
                if (i == j + 1 || (i > 0 && num[i] != num[i - 1])) {
                    int left = i + 1, right = num.length-1;

                    while (left < right) {
                        int sum = num[i] + num[j] + num[left] + num[right];

                        if (sum == target) {
                            res.add(Arrays.asList(num[i], num[j], num[left], num[right]));

                            //avoid duplicated entry by moving forward the index
                            while (left < right && num[left] == num[left + 1]) {
                                left++;
                            }
                            while (left < right && num[right] == num[right - 1]) {
                                right--;
                            }
                            left++;
                            right--;
                        } else if (sum < target) {
                            left++;
                        } else {
                            // sum > 0
                            right--;
                        }
                    }
                }
            }
        }
    }

    return res;
}
```

2.8 4Sum II}

2.8.1 Description

Given four lists A, B, C, D of integer values, compute how many tuples (i, j, k, l) there are such that A[i] + B[j] + C[k] + D[l] is zero.

To make problem a bit easier, all A, B, C, D have same length of N where $0 \leq N \leq 500$. All integers are in the range of -2^{28} to $2^{28} - 1$ and the result is guaranteed to be at most $2^{31} - 1$.

2.8.2 Example

Input: A = [1, 2] B = [-2,-1] C = [-1, 2] D = [0, 2]

Output: 2

Explanation: The two tuples are: 1. (0, 0, 0, 1) − > A[0] + B[0] + C[0] + D[1] = 1 + (-2) + (-1) + 2 = 0 2. (1, 1, 0, 0) − > A[1] + B[1] + C[0] + D[0] = 2 + (-1) + (-1) + 0 = 0 ### Solution #### Walkthrough *Rewrite the equation as A[i] + B[j] = -(C[k] + D[l]). Create a Map<Integer, Integer> where 'key' is A[i] + B[j] and 'value' is the number of pairs with this sum. For each -(C[k] + D[l]), see if this desired sum is in our map. If so, add the map's 'value' to our total count.

2.8.2.1 Analysis

There are two nested loops, the time complexity is $O(n^2)$, and Auxiliary Space for Map is O(n)

2.8.2.2 Algorithm

2.8.3 Java Code

```
public int fourSumCount(int[] A, int[] B, int[] C, int[] D) {
    int n = A.length;

    if(n == 0) {
        return 0;
    }

    int[] sumOfAandB = new int[n * n];
    int result = 0;
```

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

    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            sumOfAandB[i*n + j] = A[i] + B[j];

            //record # of pairs for this sum of [A, B]
            int count = map.getOrDefault(sumOfAandB[i*n + j], 0) + 1;
            map.put(sumOfAandB[i*n + j], count);
        }
    }

    //A + B = - (C + D)
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            int sumOfCandD = - (C[i] + D[j]);

            if(map.containsKey(sumOfCandD)){
                result += map.get(sumOfCandD);
            }
        }
    }


    return result;
}
```

2.9 Max Gain / Firecode / Level 2}

2.9.1 Description

Given an array of integers, write a method - maxGain - that returns the maximum gain. Maximum Gain is defined as the maximum difference between 2 elements in a list such that the larger element appears after the smaller element. If no gain is possible, return 0.

2.9.2 Example

[0,50,10,100,30] returns 100

[100,40,20,10] returns 0

[0,100,0,100,0,100] returns 100

2.9.3 Solution

2.9.3.1 Walkthrough

By definition, **the larger element must always appear after the smaller element**, we could do this in one pass by finding the minimum element and the maximum gain (so far) by Math.max(min, a[i] - min)

2.9.3.2 Analysis

The time complexity is O(n) since every element is visited in the loop.

2.9.3.3 Algorithm

2.9.4 Java Code

```
public static int maxGain(int[] a) {
    int min = Integer.MAX_VALUE, gain = 0;

    for(int i = 0; i < a.length; i++) {
        min = Math.min(min, a[i]);
        gain = Math.max(gain, a[i] - min);
    }

    return gain;
}
```

2.10 Pascal's Triangle / Leet Code 118 / Easy}

2.10.1 Description

Given a non-negative integer numRows, generate the first numRows of Pascal's triangle. In Pascal's triangle, each number is the sum of the two numbers directly above it.

2.10.2 Example

2.10.3 Solution

2.10.3.1 Walkthrough

For level 1 and level 2, add number 1. For level 3 or above if it is the first or last element, insert 1, otherwise, insert the sum of last two elements in the level above : [i-1][j] + [i-1][j-1].

2.10.3.2 Analysis

There are two nested loops, the time complexity is $O(n^2)$\$, and Auxiliary Space is $O(n^2)$.

2.10.3.3 Algorithm

2.10.4 Java Code

```
public List<List<Integer>> generate(int numRows) {
    List<List<Integer>> triangle = new ArrayList<>();

    for(int i = 0; i < numRows; i++) {
        List<Integer> level = new ArrayList<>();

        for(int j = 0; j < (i+1); j++) {
            if(i == 0 || i == 1) {
                // for level 1 or level 2
                level.add(1);
            } else {
                // for level 3 or above
                if(j == 0 || j == i) {
                    level.add(1);
                } else {
                    int op1 = triangle.get(i-1).get(j-1);
                    int op2 = triangle.get(i-1).get(j);
                    level.add(op1 + op2);
                }
            }
        }

        triangle.add(level);
    }
```

```
    return triangle;
}
```

2.11 Search in Rotated Sorted Array / Leet Code 33 / Medium }

2.11.1 Description

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand. (i.e., [0,1,2,4,5,6,7] might become [4,5,6,7,0,1,2]).

You are given a target value to search. If found in the array return its index, otherwise return -1. You may assume no duplicate exists in the array. Your algorithm's runtime complexity must be in the order of O(log n).

2.11.2 Example

Input: nums = [4,5,6,7,0,1,2], target = 0 Output: 4

Input: nums = [4,5,6,7,0,1,2], target = 3 Output: -1 ### Solution #### Walkthrough The key point is to search the element in a divide-and-conquer manner. We need to repeated compare the mid element of index = (left + right) / 2 with target and keep shrinking the boundaries from left and right two ends according to the conditions. Return the mid index if found; otherwise, return -1.

2.11.2.1 Analysis

The time complexity is O(log n) since we only pick one from half of target elements each time.

2.11.2.2 Algorithm

dnc 1.1.10

2.11.3 Java Code

```
public int search(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
```

```
    while(left <= right) {
        int mid = (right - left) / 2 + left;

        if(nums[mid] == target) {
            //Found index
            return mid;
        } else if(nums[mid] >= nums[left]) {
            //left half of array
            if(target >= nums[left] && target < nums[mid]) {
                //<-- moving leftward
                right = mid -1;
            } else {
                //--> moving rightward
                left = mid + 1;
            }
        } else {
            //nums[mid] < nums[right
            //right half of array
            if(target > nums[mid] && target <= nums[right]) {
                //--> moving rightward
                left = mid + 1;
            } else {
                //<-- moving leftward
                right = mid - 1;
            }
        }
    }

    return -1;
}
```

2.12 Median of Two Sorted Arrays / Leet Code 4 / Hard}

2.12.1 Description

There are two sorted arrays nums1 and nums2 of size m and n respectively. Find the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)). You may assume nums1 and nums2 cannot be both empty.

2.12.2 Example

nums1 = [1, 3], nums2 = [2], The median is 2.0

nums1 = [1, 2], nums2 = [3, 4], The median is (2 + 3)/2 = 2.5

2.12.3 Solution

2.12.3.1 Walkthrough

Recursively find K^{th} element in two sorted array by comparing and discarding the $\frac{k}{2}$ smaller elements.

2.12.3.2 Analysis

For each round of recursive, it is eliminating $\frac{k}{2}$ element, so total time complexity is $O(log(k)) = O(log(m+n))$. Auxiliary Space is O(1).

2.12.3.3 Algorithm

recursion 1.1.5

2.12.4 Java Code

```
public double findMedianSortedArrays(int[] nums1, int[] nums2) {
    int len1 = nums1.length;
    int len2 = nums2.length;
    int total = len1 + len2;

    if ((total & 1) != 0) {
        // odd number length
        return findKth(nums1, 0, len1 - 1, nums2, 0, len2 - 1, total / 2 + 1);
    } else {
        // even number length
        return (findKth(nums1, 0, len1 - 1, nums2, 0, len2 - 1, total / 2) + findKth(nu
    }
}

private int findKth(int[] nums1, int start1, int end1, int[] nums2, int start2, int end
    int len1 = end1 - start1 + 1;
    int len2 = end2 - start2 + 1;
```

```
    /*
    * swap parameters for nums1 with nums2
    */
    if (len1 > len2) {
        return findKth(nums2, start2, end2, nums1, start1, end1, k);
    }

    if (len1 == 0) {
        return nums2[start2 + k - 1];
    }

    if (k == 1) {
        //return the smaller element of nums1[] and nums2[]
        return Math.min(nums1[start1], nums2[start2]);
    }

    //Calculate number of elements to discard for next recursive
    int endToDiscard1 = start1 + Math.min(len1, k / 2) - 1;
    int endToDiscard2 = start2 + Math.min(len2, k / 2) - 1;

    if (nums1[endToDiscard1] > nums2[endToDiscard2]) {
        //if nums2[] are smaller, discard, and start from following position recursively
        int newK = k - (endToDiscard2 - start2 + 1);
        return findKth(nums1, start1, end1, nums2, endToDiscard2 + 1, end2, newK);
    } else {
        //if nums1[] are smaller, discard, and start from following position recursively
        int newK = k - (endToDiscard1 - start1 + 1);
        return findKth(nums1, endToDiscard1 + 1, end1, nums2, start2, end2, newK);
    }
}
```

2.13 Retrieve List of Elements Appeared at k^{th} Time and in Insertion Order / /}

2.13.1 Description

Given an unsorted, possibly duplicated elments of array, return the list of element which appeared at kth time where k >= 1. The returned elements need to be stable as they were in insertion order.

2.13.2 Example

$[1_1, 2_1, 3_1, 4_1, 2_2, 1_2, 1_3, 3_2], k = 1. => [1_1, 2_1, 3_1, 4_1]$

$[1_1, 2_1, 3_1, 4_1, 2_2, 1_2, 1_3, 3_2], k = 2. => [2_2, 1_2, 3_2]$

$[1_1, 2_1, 3_1, 4_1, 2_2, 1_2, 1_3, 3_2], k = 3. => [1_3]$

2.13.3 Solution

2.13.3.1 Walkthrough

Have a map to record the integer and occurrence. Only save to the list when latest occurrence equals to k, so that we could maintain insertion order.

2.13.3.2 Analysis

Time complexity is O(n) since every element is visited, and Auxiliary Space is O(n).

2.13.3.3 Algorithm

2.13.4 Java Code

```
List<Integer> insertToKthElement(int[] array, int k) {
    List<Integer> result = new ArrayList<>();
    if(array == null || array.length == 0) {
        return result;
    }

    Map<Integer, Integer> map = new HashMap<>();
    for(int num : array) {
        int count = map.getOrDefault(num, 0);
        map.put(num, ++count);

        //latest occurrences equal to k
        if( count == k) {
            result.add(num);
        }
    }

    return result;
}
```

2.14 Permutations / LeetCode 46 / Medium}

2.14.1 Description

Given a collection of distinct integers, return all possible permutations.

2.14.2 Example

Input: [1,2,3]

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

2.14.3 Solution - Backtrack with Memoization}

2.14.3.1 Walkthrough

One way to enumerate all permuations is to recursively add elements into list (avoid adding duplicates) and removing the last element form the last. In order to save time to process the same element, we further save flag for each element to denote that if the element has been visited or not.

2.14.3.2 Analysis

Time complexity with memoization to skip some subproblems is $(n + n \cdot (n-1) + n \cdot (n-1) \cdot (n-2) + ... + n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 1) \cdot n \Rightarrow O(n!)$

2.14.3.3 Algorithm

backtrack 1.1.6, memoization 1.1.8

2.14.4 Java Code - Backtrack with Memoization}

```
List<List<Integer>> result= new ArrayList<>();

public List<List<Integer>> permute(int[] nums) {
    boolean[] visited = new boolean[nums.length];

    Arrays.sort(nums);
    backtrack(new ArrayList<>(), nums, visited);

    return result;
}

private void backtrack(List<Integer> list, int [] nums, boolean[] visited){
    if(list.size() == nums.length){
        //copy elements from the current list
        result.add(new ArrayList<>(list));
        return;
    }

    for(int i = 0; i < nums.length; i++){
        Integer element = nums[i];

        if(visited[i]) {
            continue;
        }

        //add a new element
        list.add(element);
        visited[i] = true;

        backtrack(list, nums, visited);

        //backtrack the last element
        list.remove(list.size()-1);
        visited[i] = false;
    }
}
```

2.15 Permutations II / LeetCode 47 / Medium}

2.15.1 Description

Given a collection of numbers that might contain duplicates, return all possible unique permutations.

2.15.2 Example

Input: [1,1,2]

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

2.15.3 Solution - Backtrack with Memoization}

2.15.3.1 Walkthrough

While we enumerate all possible enumerate with recursive nature, we need to maintain a visited flag for each element to ensure that same element (or same value) would not be processed again. Thus, we could define the following skipping conditions:

```
* The current[i] element has been visited.
* The current[i] element is the same (value) as the previously[i-1] visited element.
```

2.15.3.2 Analysis

Time complexity $O(n!)$ with memoization to skip some subproblems.

2.15.3.3 Algorithm

backtrack 1.1.6, memoization 1.1.8

2.15.4 Java Code - Backtrack with Memoization}

```
List<List<Integer>> result= new ArrayList<>();

public List<List<Integer>> permuteUnique(int[] nums) {
    boolean visited[] = new boolean[nums.length];

    //Sort the array
    Arrays.sort(nums);
    backtrack(new ArrayList<>(), nums, visited);
    return result;
}

private void backtrack(List<Integer> list, int [] nums, boolean[] visited){
    if(list.size() == nums.length){
        //copy elements from the current list
        result.add(new ArrayList<>(list));
        return;
    }

    for(int i = 0; i < nums.length; i++){
        if(visited[i] == true || (i > 0 && visited[i-1] == false && nums[i] == nums[i-
            /*
            * Skip the permutation if any of the condition satisifies:
            * 1) The current[i] element has been visited.
            * 2) The current[i] element is the same (value) as the previously[i-1] vis
            */
            continue;
        }

        Integer element = nums[i];

        //add a new element
        list.add(element);
        visited[i] = true;

        backtrack(list, nums, visited);

        //backtrack the last element
        list.remove(list.size()-1);
        visited[i] = false;
    }
}
```

2.16 Subsets / LeetCode 78 / Medium}

2.16.1 Description

Given a set of distinct integers, nums, return all possible subsets (the power set). Note: The solution set must not contain duplicate subsets.

2.16.2 Example

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

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

2.16.3 Solution

2.16.3.1 Walkthrough

The enumeration tree is as the following:

Enumerate all possible result by adding a new element that is greater than current element while traversing the array. Remember to backtrack.

2.16.3.2 Analysis

Time complexity between $O(2^n)$ as there are at most 2^n subsets.

2.16.3.3 Algorithm

backtrack 1.1.6, recursion 1.1.5

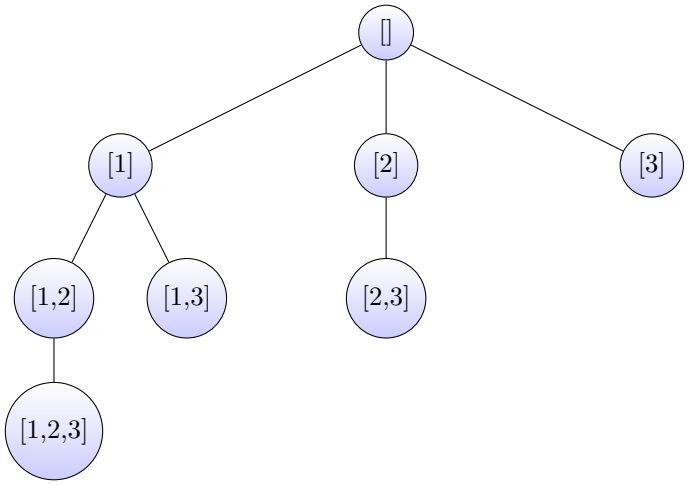


Figure 2.2: Some caption.

2.16.4 Java Code

```
List<List<Integer>> result = new ArrayList<>();

public List<List<Integer>> subsets(int[] nums) {
    if (nums.length == 0) {
        return result;
    }

    Arrays.sort(nums);

    backtrack(nums, 0, new ArrayList<>());

    return result;
}

private void backtrack(int[] nums, int index, List<Integer> list) {
    //copy elements from the current list
    result.add(new ArrayList<>(list));

    for(int i = index; i < nums.length; i++) {
        list.add(nums[i]);
        backtrack(nums, i + 1, list);

        //remove last element to backtrack
        list.remove(list.size() - 1);
```

```
    }
}
```

2.17 Subsets II / LeetCode 90 / Medium}

2.17.1 Description

Given a collection of integers that might contain duplicates, nums, return all possible subsets (the power set).

Note: The solution set must not contain duplicate subsets.

2.17.2 Example

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

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

2.17.3 Solution

2.17.3.1 Walkthrough

The enumeration tree is as the following:

Enumerate all possible result by adding a new element that is greater than current element while traversing the array. Skip if two consecutive elements are the same. Remember to backtrack.

2.17.3.2 Analysis

Time complexity between $O(2^n)$ as there are at most 2^n subsets.

2.17.3.3 Algorithm

backtrack 1.1.6, recursion 1.1.5

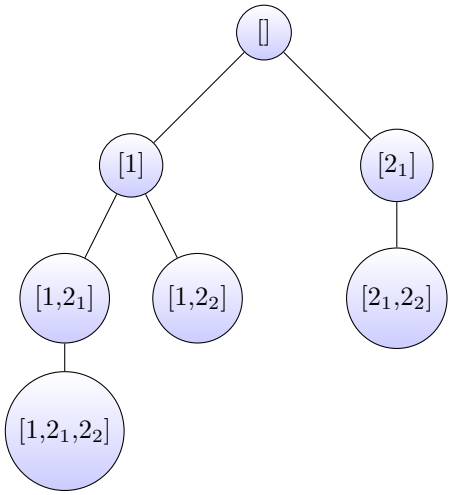


Figure 2.3: Some caption.

2.17.4 Java Code

```
List<List<Integer>> result = new ArrayList<>();

public List<List<Integer>> subsetsWithDup(int[] nums) {
    if (nums.length == 0) {
        return result;
    }

    Arrays.sort(nums);
    backtrack(nums, 0, new ArrayList<>());

    return result;
}

private void backtrack(int[] nums, int index, List<Integer> list) {
    //copy elements from the current list
    result.add(new ArrayList<>(list));

    for(int i = index; i < nums.length; i++) {
        if(i != index && nums[i] == nums[i - 1]) {
            continue;
        }

        list.add(nums[i]);
        backtrack(nums, i + 1, list);
```

```
        //remove last element to backtrack
        list.remove(list.size() - 1);
    }
}
```

2.18 Sort Array By Parity / LeetCode 906 / Easy}

2.18.1 Description

Given an array A of non-negative integers, return an array consisting of all the even elements of A, followed by all the odd elements of A.

You may return any answer array that satisfies this condition.

2.18.2 Example

Input: [3,1,2,4]. Output: [2,4,3,1]. The outputs [4,2,3,1], [2,4,1,3], and [4,2,1,3] would also be accepted.

2.18.3 Solution

2.18.3.1 Walkthrough

Have two indices, left and right. Shrink both indices (left, right) where they satisfy the condition. Swap those who do not and shrink both indices again.

2.18.3.2 Analysis

Complexity is $O(n)$ since each element is visited once.

2.18.3.3 Algorithm

2.18.4 Java Code

```
public int[] sortArrayByParity(int[] A) {
    if(A == null || A.length == 0) {
        return null;
    }
```

```
    int l = 0, r = A.length - 1;
    while(l < r) {
        while (A[l]%2 == 0 && l < r) {
            //do nothing, incremnt left index
            l++;
        }

        while (A[r]%2 == 1 && l < r) {
            //do nothing, decrement right index
            r--;
        }


        if( l < r ) {
            //odd #, swap
            int temp = A[l];
            A[l] = A[r];
            A[r] = temp;

            l++;
            r--;
        }
    }

    return A;
}
```

2.19 Merge Intervals / LeetCode 56 / Medium}

2.19.1 Description

Given a collection of intervals, merge all overlapping intervals.

2.19.2 Example

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

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

2.19.3 Solution

2.19.3.1 Walkthrough

Sort the list of intervals first. Use a stack to track the lastly pushed interval. If the current interval does not overlap with the top interval, push current interval to stack. If there is an overlap, we merge the current and previous interval.

2.19.3.2 Analysis

Complexity is : $O(n \cdot logn)$ since we sort the array first.

2.19.3.3 Algorithm

2.19.4 Java Code

```
static class Interval {
    int start;
    int end;

    public Interval(int l, int r) {
        start = l;
        end = r;
    }
}

public int[][] merge(int[][] intervals) {
    List<Interval> input = new ArrayList<>();
    for(int[] interval : intervals) {
        input.add(new Interval(interval[0], interval[1]));
    }

    if(intervals.length == 0) {
        return new int[0][0];
    }

    List<Interval> output = merge(input);
    int[][] result = new int[output.size()][2];

    for(int i = 0; i < output.size(); i++) {
        result[i][0] = output.get(i).start;
        result[i][1] = output.get(i).end;
    }
```

```
    return result;
}

private class IntervalComparator implements Comparator<Interval> {
    @Override
    public int compare(Interval a, Interval b) {
        return a.start - b.start;
    }
}

public List<Interval> merge(List<Interval> intervals) {
    //sort the list
    Collections.sort(intervals, new IntervalComparator());

    Stack<Interval> merged = new Stack<Interval>();
    merged.push(intervals.get(0));

    for (int i = 1; i < intervals.size(); i++) {
        Interval interval = intervals.get(i);
        Interval top = merged.peek();

        // if interval does not overlap with the previous, simply append it.
        if (top.end < interval.start) {
            merged.push(interval);
        }
        // if there is an overlap, we merge the current with the last interval
        // by comparing their end boundaries
        else {
            top.end = Math.max(top.end, interval.end);
        }
    }

    return merged;
}
```

2.20 Non-overlapping Intervals / LeetCode 435 / Medium}

2.20.1 Description

Given a collection of intervals, find the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.

2.20.2 Example

Input: [[1,2],[2,3],[3,4],[1,3]]. Output: 1 Explanation: [1,3] can be removed and the rest of intervals are non-overlapping.

Input: [[1,2],[1,2],[1,2]]. Output: 2 . Explanation: You need to remove two [1,2] to make the rest of intervals non-overlapping.

2.20.3 Solution

2.20.3.1 Walkthrough

First, sort the array and count non-overlapping interval - not overlapped with previous end. Finally, number of overlapping intervals would be n - count.

2.20.3.2 Analysis

Complexity is $O(n \cdot logn)$ because of sorting ahead.

2.20.3.3 Algorithm

2.20.4 Java Code

```
public int eraseOverlapIntervals(int[][] intervals) {
    if(intervals == null || intervals.length == 0) {
        return 0;
    }

    Arrays.sort(intervals, new Comparator<int[]>() {
        @Override
        public int compare(int[] i1, int[] i2) {
            if (i1[1] != i2[1]){
                //compare end time
                return i1[1] - i2[1];
            }else {
                //compare start time
                return i1[0] - i2[0];
            }
        }
    });

    //end for latest non-overlapped interval
    int end = intervals[0][1];
```

```
    int n = intervals.length;
    int count = 1;

    for (int i = 1; i < n; i++) {
        if (intervals[i][0] >= end) {
            //for any non-overlapped interval, update end & count
            end = intervals[i][1];
            count++;
        }
    }

    return n - count;
}
```

2.21 Interval List Intersections / LeetCode 986 / Medium}

2.21.1 Description

Given two lists of closed intervals, each list of intervals is pairwise disjoint and in sorted order. Return the intersection of these two interval lists.

2.21.2 Example

Input: A = [[0,2],[5,10],[13,23],[24,25]], B = [[1,5],[8,12],[15,24],[25,26]] Output: [[1,2],[5,5],[8,10],[15,23],[24,24],[25,25]] Reminder: The inputs and the desired output are lists of Interval objects, and not arrays or lists.

2.21.3 Solution

2.21.3.1 Walkthrough

Traverse the two lists of intervals. Merge two intervals if there is an intersection. Move the index of A if $A[i]$.end $< B[i]$.end since the interval is the smaller out of comparison.

2.21.3.2 Analysis

The runtime complexity is $O(n * m)$

2.21.3.3 Algorithm

2.21.4 Java Code

```
LeetCode 986 / Medium
private static class Interval {
    public int start;
    public int end;

    public Interval(int s, int e) {
        this.start = s;
        this.end = e;
    }

    public static int[][] toArray(List<Interval> list) {
        int[][] result = new int[list.size()][2];

        for(int i = 0; i < list.size(); i++) {
            Interval interval = list.get(i);

            result[i] = new int[] {interval.start, interval.end};
        }

        return result;
    }

    public static List<Interval> toList(int[][] array) {
        List<Interval> result = new ArrayList<>();

        for(int i = 0; i < array.length; i++) {
            Interval interval = new Interval(array[i][0], array[i][1]);

            result.add(interval);
        }

        return result;
    }

}
public int[][] intervalIntersection(int[][] A, int[][] B) {
    List<Interval> intervalsA = Interval.toList(A);
    List<Interval> intervalsB = Interval.toList(B);
    List<Interval> result = new ArrayList<>();

    int i = 0, j = 0;
```

```
    while( i < intervalsA.size() && j < intervalsB.size()) {
        int highStart = Math.max(intervalsA.get(i).start, intervalsB.get(j).start);
        int lowEnd = Math.min(intervalsA.get(i).end, intervalsB.get(j).end);

        //there is an intersection
        if( highStart <= lowEnd) {
            result.add( new Interval(highStart, lowEnd));
        }
        if( intervalsA.get(i).end < intervalsB.get(j).end) {
            //move index of A[]
            i++;
        } else {
            //move index of B[]
            j++;
        }
    }

    return Interval.toArray(result);
}
```

2.22 Insert Interval / LeetCode 57 / Hard}

2.22.1 Description

Given a set of non-overlapping intervals, insert a new interval into the intervals (merge if necessary).

You may assume that the intervals were initially sorted according to their start times.

2.22.2 Example

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

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].

2.22.3 Solution

2.22.3.1 Walkthrough

Leverage the merge() method to find out which intervals can be merged with new Inserted interval.

2.22.3.2 Analysis

Time complexity is $O(n \cdot log(n))$ because of sorting ahead.

2.22.3.3 Algorithm

2.22.4 Java Code

```
private static class Interval {
    public int start;
    public int end;

    public Interval(int s, int e) {
        this.start = s;
        this.end = e;
    }

    public static int[][] toArray(List<Interval> list) {
        int[][] result = new int[list.size()][2];

        for(int i = 0; i < list.size(); i++) {
            Interval interval = list.get(i);

            result[i] = new int[] {interval.start, interval.end};
        }

        return result;
    }

    public static List<Interval> toList(int[][] array) {
        List<Interval> result = new ArrayList<>();

        for(int i = 0; i < array.length; i++) {
            Interval interval = new Interval(array[i][0], array[i][1]);

            result.add(interval);
```

```
        }

        return result;
    }
}

public int[][] insert(int[][] existed, int[] target) {
    List<Interval> intervals = Interval.toList(existed);
    Interval newInterval = new Interval(target[0], target[1]);

    intervals.add(newInterval);

    List<Interval> merged = merge(intervals);

    return Interval.toArray(merged);
}

private class IntervalComparator implements Comparator<Interval> {
    @Override
    public int compare(Interval a, Interval b) {
        return a.start < b.start ? -1 : a.start == b.start ? 0 : 1;
    }
}

public List<Interval> merge(List<Interval> intervals) {
    //sort the list
    Collections.sort(intervals, new IntervalComparator());

    LinkedList<Interval> merged = new LinkedList<Interval>();
    for (Interval interval : intervals) {
        if (merged.isEmpty() || merged.getLast().end < interval.start) {
            // if the list of merged intervals is empty or if the current
            // interval does not overlap with the previous, simply append it.
            merged.add(interval);
        } else {
            // otherwise, there is overlap, so we merge the current and previous
            // intervals.
            merged.getLast().end = Math.max(merged.getLast().end, interval.end);
        }
    }
    return merged;
}
```

2.23 Find Common Characters / LeetCode 1002 / Easy}

2.23.1 Description

Given an array A of strings made only from lowercase letters, return a list of all characters that show up in all strings within the list (including duplicates). For example, if a character occurs 3 times in all strings but not 4 times, you need to include that character three times in the final answer. You may return the answer in any order.

2.23.2 Example

Input: [“bella”,“label”,“roller”]. Output: [“e”,“l”,“l”]

Input: [“cool”,“lock”,“cook”]. Output: [“c”,“o”]

2.23.3 Solution

2.23.3.1 Walkthrough

First retrieve the alphabet distribution for the first word. For each of following word, maintain the min number of duplicated alphabets. Lastly, take the alphabet which has more than 1 occurrences.

2.23.3.2 Analysis

Time complexity is O(n) where n is the number of words.

2.23.3.3 Algorithm

2.23.4 Java Code

```
public List<String> commonChars(String[] A) {
    int[] minFreq = new int[26];

    //init frequency map with the first string
    for(int i = 0; i < A[0].length(); i++) {
        char c = A[0].charAt(i);

        int index = c - 'a';
```

```
            int counter = minFreq[index];
            minFreq[index] = ++counter;
        }


        //Do the actions for the following words
        for(int i = 1; i < A.length; i++) {
            String str = A[i];
            int[] tempFreq = new int[26];

            //1. Store the frequency of alphabets
            for(int j = 0; j < str.length(); j++) {
                char c = str.charAt(j);

                int index = c - 'a';
                int counter = tempFreq[index];
                tempFreq[index] = ++counter;
            }

            //2. Iterate thru 2 arrays and get the min number of (duplicated) alphabets
            //   non-duplicated returns 0
            for(int j = 0; j < 26; j++) {
                minFreq[j] = Math.min(minFreq[j], tempFreq[j]);
            }
        }

        int numOfWords = A.length;
        List<String> result = new ArrayList<>();


        for(int i = 0; i < 26; i++) {
            int minCounter = minFreq[i];

            //Take the min number of duplicated alphabets (1 min)
            for(int j = 0; j < minCounter; j++) {
                char alphabet = (char) ('a' + i);
                result.add(String.valueOf(alphabet));
            }
        }

        return result;
    }
```

2.24 Top K Frequently Appeared Elements / Leet Code 347 / Medium}

2.24.1 Description

Given a non-empty array of integers, return the k most frequent elements. You may assume k is always valid, $1 k $ number of unique elements. Your algorithm's time complexity must be better than O($nlogn$), where n is the array's size.

2.24.2 Example

Input: nums = [1,1,1,2,2,3], k = 2 Output: [1,2] ### Solution #### Walkthrough

```
* We first gather the frequency of integer by count with a HashMap
* We sort the frequency map by comparing by value() in descending order and sort them in
$List<Entry<Number, Frequency>>$.
* We could traverse the top K entries from the list in a loop.
```

2.24.2.1 Analysis

Since we sort the frequency map, thus the time complexity is $O(n \cdot logn)$, and Auxiliary Space is $O(n)$ to store several maps and list.

2.24.2.2 Algorithm

2.24.3 Java Code

```java
public List<Integer> topKFrequent(int[] nums, int k) {
    Map<Integer, Integer> freq = new HashMap<>();

    for (int num : nums) {
        int count = freq.getOrDefault(num, 0);
        freq.put(num, ++count);
    }

    //Sort Map.Entry by comparing by value() in descending order
    List<Map.Entry<Integer, Integer>> sortedList = new ArrayList<>(freq.entrySet());
    sortedList.sort(Map.Entry.comparingByValue(Comparator.reverseOrder()));
```

```
    List<Integer> result = new ArrayList<>();
    int topN = 0;

    for (Map.Entry<Integer, Integer> entry : sortedList) {
        //Top K elements retrieved
        if(topN == k) {
            break;
        }

        result.add(entry.getKey());
        topN++;
    }

    return result;
}
```

2.25 Count iterations towards filling 1's in an array / / }

2.25.1 Description

Given an array of 0s and 1s, in how many iterations the whole array can be filled with 1s if in a single iteration immediate neighbors of 1s can be filled. If we cannot fill array with 1s, then print "-1"

2.25.2 Example

Input : arr[] = {1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1} Output : 1

Input : arr[] = {0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1} Output : 2

2.25.3 Solution

2.25.3.1 Walkthrough

For each sub-block of the array, consider the following 3 scenarios

```
* Case 1: There are no 1's. In this case, array cannot be filled with 1's. Thus, return -1;
* Locate the first 1 and evalute the following

      * Case 2: [1, 0, 0, ...., 0, 1] Another 1 follows and makes it a block of 0s with 1s on
```

to flip the 0s become $num_zeo/2$ if number is even otherwise $(num_zero + 1)/2$ * Case 3: [1, 0, 0, ..., 0] Another 1 cannot be found and makes it a single 1 at the end. Number of iteration needed equals to the number of 0s.

* Case 3: [0, 0, 0, ..., 1] Count the number of 0s until the first 1 is met.

Finally, we need to get the maximum iteration for each subproblems.

2.25.3.2 Analysis

Time Complexity : O(n) since every element is visited once.

2.25.3.3 Algorithm

2.25.4 Java Code

```
int countIterations(int arr[]) {
    boolean oneFound = false;
    int maxIteration = 0;

    int n = arr.length;
    int i = 0;

    // Start traversing the array
    while ( i < n ) {
        if (arr[i] == 1) {
            oneFound = true;
        }

        // Traverse and skip 1s until a 0 is met
        while (i < n && arr[i]==1) {
            i++;
        }

        // Count initial contiguous 0s until a 1 is met
        int inialCountZero = 0;
        while ( i < n && arr[i]==0) {
            inialCountZero++;
            i++;
        }

        // Condition for Case 1
        if (oneFound == false && i == n) {
            return -1;
```

```
        }

        // Condition to check if Case 2 satisfies:
        int countIteration;
        if (i < n && oneFound == true) {

            // If inialCountZero is even
            if ((inialCountZero & 1) == 0) {
                countIteration = inialCountZero / 2;
            } else {
                //odd
                countIteration = (inialCountZero + 1) / 2;
            }

            inialCountZero = 0;
        } else {
            // Case 3
            countIteration = inialCountZero;
            inialCountZero = 0;
        }

        maxIteration = Math.max(maxIteration, countIteration);
        //totalIteration += countIteration
    }

    return maxIteration;
}
```

2.26 Maximum Subarray / Leet Code 53 / Easy}

2.26.1 Description

Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

2.26.2 Example

For example, given the array [-2,1,-3,4,-1,2,1,-5,4], the contiguous subarray [4,-1,2,1] has the largest sum = 6.

2.26.3 Solution

2.26.3.1 Walkthrough

While traversing the array dp[i], add the current element dp[i] with the previous element dp[i] if and only if previous element $dp[i-1] > 0$. In the meantime, compute the max value out of current element dp[i].

2.26.3.2 Analysis

Time complexity is O(n) as every element is visited once.

2.26.3.3 Algorithm

dp 1.1.7

2.26.4 Java Code

```
public int maxSubArray(int[] nums) {
    if(nums.length == 0) {
        return 0;
    }
    int[] dp = new int[nums.length];
    int max = Integer.MIN_VALUE;

    //init dp[]
    for(int i = 0; i < nums.length; i++) {
        dp[i] = nums[i];
    }

    for(int i = 0; i < nums.length; i++) {
        //keep summing up with positive integer until a max is found
        if(i > 0 && dp[i-1] > 0) {
            dp[i] += dp[i - 1];
        }
        max = Math.max(dp[i], max);
    }

    return max;
}
```

2.27 Maximum Product Subarray / Leet Code 152 / Medium}

2.27.1 Description

Find the contiguous subarray within an array (containing at least one number) which has the largest product.

2.27.2 Example

For example, given the array [2,3,-2,4], the contiguous subarray [2,3] has the largest product = 6.

2.27.3 Solution

2.27.3.1 Walkthrough

Need to keep track the not only maximum but also minimum product since there could be negative number in array.

2.27.3.2 Analysis

Time complexity is O(n) as every element is visited once.

2.27.3.3 Algorithm

dp 1.1.7

2.27.4 Java Code

```
public int maxProduct(int[] nums) {
    if(nums.length == 0) {
        return 0;
    }

    int maxProduct = nums[0];

    int maxPrev = nums[0], minPrev = nums[0];

    int maxCurrent, minCurrent;
```

```
    for(int i = 1; i < nums.length; i++) {
        maxCurrent = Math.max(Math.max(maxPrev * nums[i], minPrev * nums[i]), nums[i]);
        minCurrent = Math.min(Math.min(maxPrev * nums[i], minPrev * nums[i]), nums[i]);

        //refresh max
        maxProduct = Math.max(maxProduct, maxCurrent);

        //refresh values
        maxPrev = maxCurrent;
        minPrev = minCurrent;
    }

    return maxProduct;
}
```

2.28 Longest Valid Parentheses / Leet Code 32 / Hard}

2.28.1 Description

Given a string containing just the characters '(' and ')', find the length of the longest valid (well-formed) parentheses substring.

2.28.2 Example

For "(()", the longest valid parentheses substring is "()", which has length = 2. Another example is ")()())", where the longest valid parentheses substring is "()()", which has length = 4.

2.28.3 Solution

2.28.3.1 Walkthrough

Find the indices of unmatched paranthesis, then locate the adjacent indices with largest difference which would be the longest valid paranthesis.

2.28.3.2 Analysis

Time complexity is O(n) where n is the number of alphabets.

2.28.3.3 Algorithm

dp 1.1.7

2.28.4 Java Code

```
public int longestValidParentheses(String s) {
    //Stack contains the indices of chars which cannot be matched.
    Stack<Integer> stack = new Stack<>();

    for(int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);

        if(c == '(') {
            stack.push(i);
        } else if(c == ')') {
            if(stack.isEmpty()) {
            stack.push(i);
            } else {
                int lastIndex = stack.peek();

                if(s.charAt(lastIndex) == '(') {
                    //pop out the valid pairs
                    stack.pop();
                } else {
                    //XXX: )))))(
                    stack.push(i);
                }
            }
        }
    }

    int longestLength = 0;
    if(stack.isEmpty()) {
        //string contains a well-formed paranthesis
        longestLength = s.length();
    } else {
        /*
        * Any discontinual adjacent indices represents a valid parenthesis.
        * Therefore, locate the adjacent indices with largest difference ==> the longe
        *
        * Example:
        * Input: (()()(((()((
        * Indices in stack:
```

```
        * 10<-9<-6<-5<-0
        *
        * Longest valid parenthesis: 5 - 0  - 1 = 4
        */
        int stopIndex = s.length(), startIndex = 0;

        while(!stack.isEmpty()) {
            startIndex = stack.pop();
            longestLength = Math.max(longestLength, stopIndex - startIndex - 1);
            stopIndex = startIndex;
        }

        longestLength = Math.max(longestLength, stopIndex);
    }

    return longestLength;
}
```

2.29 Longest Increasing Subsequence / Leet Code 300 / Medium}

2.29.1 Description

Given an unsorted array of integers, find the length of longest increasing subsequence.

2.29.2 Example

For example, Given [10, 9, 2, 5, 3, 7, 101, 18],

The longest increasing subsequence is [2, 3, 7, 101], therefore the length is 4. Note that there may be more than one LIS combination, it is only necessary for you to return the length.

Your algorithm should run in O(n2) complexity.

2.29.3 Solution- Brute Force}

2.29.3.1 Walkthrough

Create an array of length of nums to store length of subsequences. Create a nested loop where two indices, stopIdx from 0 to length -1 and from star-

tIdx from 0 to stopIdx - 1. First check if it is a increasing subsequence if (nums[stopIdx] < nums[startIdx]). Increment the current length of subsequence for at startIdx - dp[startIdx] + 1. However, it is possible that dp[stopIdx] is larger than dp[startIdx] + 1 for previously traversed subsequence. Thus, we need to take the maximum out of both cases. Finally, we evaluate the final length out of dp[] array.

2.29.3.2 Analysis

There are two loops, thus, the time complexity is $O(n^2)$

2.29.3.3 Algorithm

dp 1.1.7

2.29.4 Java Code - Brute Force}

```
public int lengthOfLIS(int[] nums) {
    int dp[] = new int[nums.length];

    if(nums.length == 0) {
        return 0;
    }

    //init
    Arrays.fill(dp, 1);

    for(int stopIdx = 0; stopIdx < nums.length; stopIdx++) {
        for(int startIdx = 0; startIdx < stopIdx; startIdx++) {
            //Valid Increasing Subsequence
            if(nums[startIdx] < nums[stopIdx]) {
                dp[stopIdx] = Math.max(dp[startIdx] + 1, dp[stopIdx]);
            }
        }
    }

    int maxLength = 0;
    for(int i = 0; i < dp.length; i++) {
        maxLength = Math.max(maxLength, dp[i]);
    }

    return maxLength;
}
```

2.29.5 Solution - Binary Search}

2.29.5.1 Walkthrough

2.29.5.2 Analysis

For each element, we search for the index using binary search which cost $O(log(n))$. The overall time complexity is $O(n \cdot log(n))$

2.29.5.3 Algorithm

2.29.6 Java Code - Binary Search}

```
public int lengthOfLIS(int[] nums) {
    int len = 1;
    for (int i = 0; i < nums.length; i++) {
        int k = Arrays.binarySearch(nums, 0, len, nums[i]);
        if (k < 0) {
            k = -(k + 1);
            if (k == len) {
                len++;
            }
            nums[k] = nums[i];
        }
    }
    return len;
}
```

2.30 Paint House / Leet Code 256 / Easy}

2.30.1 Description

There are a row of n houses, each house can be painted with one of the three colors: red, blue or green. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color. The cost of painting each house with a certain color is represented by a n x 3 cost matrix. For example, costs[0][0] is the cost of painting house 0 with color red; costs[1][2] is the cost of painting house 1 with color green, and so on… Find the minimum cost to paint all houses.

2.30.2 Example

2.30.3 Solution

2.30.3.1 Walkthrough

We need to find the cost of one house of having one color with minimum cost with another color. Last but not least, these values need to minimized.

2.30.3.2 Analysis

Time complexity is O(n) as every house is visited once.

2.30.3.3 Algorithm

dp 1.1.7

2.30.4 Java Code

```
public int minCost(int[][] costs) {
    int[] house = new int[3];
    int h0 = 0, h1 = 0, h2 = 0;

    for (int i = 0; i < costs.length; i++) {
        /*
        * house[0] = cost of current house of of r/g/b plus minimum cost of the other
        */
        house[0] = costs[i][0] + Math.min(h1, h2);
        house[1] = costs[i][1] + Math.min(h0, h2);
        house[2] = costs[i][2] + Math.min(h0, h1);
        h0 = house[0];
        h1 = house[1];
        h2 = house[2];
    }
    return Math.min(Math.min(h0, h1), h2);
}
```

2.31 Climbing Stairs / Leet Code 70 / Easy}

2.31.1 Description

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

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

Note: Given n will be a positive integer.

2.31.2 Example

Input: 2 , Output: 2 Explanation: There are two ways to climb to the top.

- 1 step + 1 step
- 2 steps \end{itemize}

Input: 3, Output: 3 Explanation: There are three ways to climb to the top.

- 1 step + 1 step + 1 step
- 1 step + 2 steps
- 2 steps + 1 step \end{itemize}

2.31.3 Solution

2.31.3.1 Walkthrough

- i = 3, total = 1 + 2 = 3
- i = 4, total = 3 + 2 = 5
- i = 5, total = 5 + 3 = 8
- i = 13, total = 8 + 5 = 13 \end{itemize}

2.31.3.2 Analysis

Time complexity is O(n) as each step is computed.

2.31.3.3 Algorithm

dp 1.1.7

2.31.4 Java Code

```
public int climbStairs(int n) {
    if(n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    if (n == 2) {
        return 2;
    }

    int n_minus_2 = 2;
    int n_minus_1 = 1;
    int total = 0;

    for(int i = 3; i <= n; i++) {
        total = n_minus_1 + n_minus_2;
        n_minus_1 = n_minus_2;
        n_minus_2 = total;
    }

    return total;
}
```

2.32 Find the Maximum Number of Repetitions / Firecode / Level 3}

2.32.1 Description

Given an Array of integers, write a method that will return the integer with the maximum number of repetitions. Your code is expected to run with O(n) time complexity and O(1) Auxiliary Space. The elements in the array are between 0 to size(array) - 1 and the array will not be empty.

2.32.2 Example

$f(\{3, 1, 2, 2, 3, 4, 4, 4\}) = 4$

2.32.3 Solution

2.32.3.1 Walkthrough

The key is to utilize the characteristic of the elements in the array being between 0 to size(array) - 1. We should increment a[a[i]] by k for each element. Retrieve the maximum element and return the index of the element. k is an increment factor as long as it is bigger than size of array.

2.32.3.2 Analysis

Time complexity is O(n) as every element is visited once.

2.32.3.3 Algorithm

dp 1.1.7

2.32.4 Java Code

```
public static int getMaxRepetition(int[] a) {
    //k to be the increment factor, larger >= size(a) to be significant
    int k = 100;

    // Iterate though input array, for every element
    // arr[i], increment a[a[i]] by k
    for (int i = 0; i< a.length; i++) {
        a[(a[i] \% k)] += k;
    }

    // Find index of the maximum repeating element
    int max = a[0], maxIdx = 0;
    for (int i = 1; i < a.length; i++) {
        if (a[i] > max) {
            max = a[i];
            maxIdx = i;
        }
    }

    // Return index of the maximum element
    return maxIdx;
}
```

2.33 House Robber / Leet Code 198 / Easy}

2.33.1 Description

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

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

2.33.2 Example

Input: [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.

Input: [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.

2.33.3 Solution

2.33.3.1 Walkthrough

Create an array of same length of nums. While looping through the array, compute the max profit of:

```
* Rob this house plus every two houses before : nums[i] + dp[i-2]
* Rob prev house: dp[i-1]
```

\end{itemize} Finally, return the final outcome, which is the dp[n-1]

2.33.3.2 Analysis

Time complexity is O(n) as every element is visited once.

2.33.3.3 Algorithm

dp 1.1.7

2.33.4 Java Code

```java
public int rob(int[] nums) {
    int n = nums.length;
    if(n == 0) {
        return 0;
    }

    int[] dp = new int[n];
    int max = 0;

    for(int i = 0; i < n; i++) {
        dp[i] = Math.max(nums[i] + (i >= 2 ? dp[i-2] : 0), (i >= 1 ? dp[i-1]: 0));
    }

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

2.34 Best Time to Buy and Sell Stock / Leet Code 121 / Easy}

2.34.1 Description

Say you have an array for which the ith element is the price of a given stock on day i.

If you were only permitted to complete at most one transaction (i.e., buy one and sell one share of the stock), design an algorithm to find the maximum profit.

Note that you cannot sell a stock before you buy one

2.34.2 Example

Input: [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. Not 7-1 = 6, as selling price needs to be larger than buying price.

2.34.3 Solution

2.34.3.1 Walkthrough

While computing (sell price - buy price), we need to keep track the maximum profit as well as the minimum minimum buying price.

2.34.3.2 Analysis

Time complexity is O(n) as every element is visited once.

2.34.3.3 Algorithm

dp 1.1.7

2.34.4 Java Code

```
public int maxProfit(int[] prices) {
    int maxProfit = 0, minPrice = Integer.MAX_VALUE;

    for(int price : prices) {
        // profit = selling price - buying price
        int profit = price - minPrice;

        maxProfit = Math.max(profit, maxProfit);
        minPrice = Math.min(minPrice, price);
    }

    return maxProfit;
}
```

2.35 Best Time to Buy and Sell Stock II / Leet Code 122 / Easy}

2.35.1 Description

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (i.e., buy one and sell one share of the stock multiple times).

Note: You may not engage in multiple transactions at the same time (i.e., you must sell the stock before you buy again).

2.35.2 Example

Input: [7,1,5,3,6,4], Output: 7 Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5-1 = 4. Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = 6-3 = 3.

2.35.3 Solution

2.35.3.1 Walkthrough

if current (selling)price greater than previous (buying)price, profit += (selling price - buying price).

2.35.3.2 Analysis

Time complexity is O(n) as every element is visited once.

2.35.3.3 Algorithm

dp 1.1.7

2.35.4 Java Code

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

2.36 Best Time to Buy and Sell Stock with Cooldown / Leet Code 309 / Medium}

2.36.1 Description

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times) with the following restrictions:

You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again). After you sell your stock, you cannot buy stock on next day. (ie, cooldown 1 day)

2.36.2 Example

Input: [1,2,3,0,2], Output: 3 Explanation: transactions = [buy, sell, cooldown, buy, sell]

2.36.3 Solution

2.36.3.1 Walkthrough

2.36.3.2 Analysis

Time complexity is O(n) as every element is visited once.

2.36.3.3 Algorithm

dp 1.1.7

2.36.4 Java Code

```
public int maxProfit(int[] prices) {
    int profit = 0, prevProfit = 0, diff = Integer.MIN_VALUE, prevDiff = 0, maxProfit =
    for (int price : prices) {
        prevDiff = diff;
        //diff = MAX (price[i - 1] - price[i], diff)
        diff = Math.max(prevProfit - price, prevDiff);
```

```
        prevProfit = profit;
        //profit = MAX(prevDiff + price[i], prevProfit)
        profit = Math.max(prevDiff + price, prevProfit);
        maxProfit = Math.max(maxProfit, profit);
    }

    return maxProfit;
}
```

2.37 Minimum Swaps To Make Sequences Increasing / LeetCode 801 / Medium}

2.37.1 Description

We have two integer sequences A and B of the same non-zero length.

We are allowed to swap elements A[i] and B[i]. Note that both elements are in the same index position in their respective sequences.

At the end of some number of swaps, A and B are both strictly increasing. (A sequence is strictly increasing if and only if A[0] < A[1] < A[2] < … < A[A.length - 1].)

Given A and B, return the minimum number of swaps to make both sequences strictly increasing. It is **guaranteed** that the given input always makes it possible.

2.37.2 Example

Input: A = [1,3,5,4], B = [1,2,3,7] Output: 1 Explanation: Swap A[3] and B[3]. Then the sequences are: A = [1, 3, 5, 7] and B = [1, 2, 3, 4] which are both strictly increasing.

2.37.3 Solution

2.37.3.1 Walkthrough

Have 2 variables to keep track states as the following:

```
* the minimum swaps to maintain sequence increasing if we swap at i
* the minimum swaps to maintain sequence increasing if we DO NOT swap at i
```

\end{itemize}

Since it is guaranteed that there is valid answer, then we could have the following scenario

```
*  Scenaior 1: Need to swap at [i] iff [i-1] swapped
    ```java
    //          | i - 1 | i     | Note
    // A        | 3     | 4     |
    // B        | 1     | 2     |
    // noSwap   | x     | x     | [i-1] NOT swapped, [i] NOT swapped
    // withSwap| y      | y + 1 | [i-1] swapped, [i] swapped.
    ```
* Scenario 2: Need to swap at [i] iff [i-1] NOT swapped
    ```java
    //          | i - 1 | i     | Note
    // A        | 3     | 2     |
    // B        | 1     | 4     |
    // noSwap   | x     | y     | [i-1] swapped => [i] NOT swapped
    // withSwap| y      | x + 1 | [i-1] NOT swapped => [i] swapped
    ```
* Scenario 3 : either swap or no swap
```

\end{itemize}

2.37.3.2 Analysis

Time complexity is O(n) as every element is visited once.

2.37.3.3 Algorithm

2.37.4 Java Code

```java
public int minSwap(int[] A, int[] B) {
    //For the ith element in A and B
    //withSwap means the minimum swaps to maintain IS if we swap at [i]
    //noSwap means the minimum swaps to maintain IS if we DO NOT swap at [i]

    //for [0] element, init noSwap = 1 and withSwap = 0
    int withSwap = 1, noSwap = 0;

    // System.out.println("i:  " + "Y" + " | " + "N");
    //assume A.length = B.length
    int length = A.length;
    for(int i = 1; i < length; i++) {
```

```
        if(A[i - 1] >= B[i] || B[i - 1] >= A[i]) {
            //Scenaior 1: Need to swap at [i] iff [i-1] swapped
            //          | i - 1 | i     | Note
            // A        | 3     | 4     |
            // B        | 1     | 2     |
            // noSwap   | x     | x     | [i-1] NOT swapped, [i] NOT swapped
            // withSwap | y     | y + 1 | [i-1] swapped, [i] swapped.
            //no-swap counter should remain
            withSwap++;
        } else if(A[i - 1] >= A[i] || B[i - 1] >= B[i]) {
            //Implicitly A[i - 1] < B[i] && B[i - 1] < A[i]
            //Scenario 2: Need to swap at [i] iff [i-1] NOT swapped
            //          | i - 1 | i     | Note
            // A        | 3     | 2     |
            // B        | 1     | 4     |
            // noSwap   | x     | y     | [i-1] swapped => [i] NOT swapped
            // withSwap | y     | x + 1 | [i-1] NOT swapped => [i] swapped
            int temp = withSwap;
            withSwap = noSwap + 1;
            noSwap = temp;
        } else {
            //Scenario 3 : either swap or no swap
            int min = Math.min(withSwap, noSwap);
            withSwap = min + 1;
            noSwap = min;
        }
        // System.out.println(i + ": " + withSwap + " | " + noSwap);
    }

    return Math.min(noSwap, withSwap);
}
```

2.38 Retrieve an Optimal Computation from an Array / / }

2.38.1 Description

Each piece has a positive integer (weight) that indicates how tasty it is. Since taste is subjective, there is also an expectancy factor. A piece will taste better if you eat it later: if the taste is m on the first day, it will be k * m on day number k. Your task is to design an efficient algorithm that computes an optimal chocolate eating strategy and tells you how much pleasure to expect.

2.38.2 Example

```
[1, 2, 3], multiplier = 1
```

Result $= 1 * 1 + 2 * 2 + 3 * 3 + 4 * 4 = 30$

2.38.3 Solution - Recursion}

2.38.3.1 Walkthrough

In the recursive startegy:

```
* In the base case, where array contains only one element, the function returns the correct
* When array contains more than 1 element, we have to start with either array[0] or array[n-
```

computes for each of these cases with recursion, and returns the maximum.

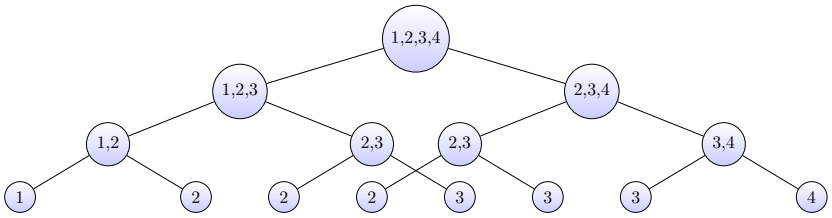


Figure 2.4: Some caption.

2.38.3.2 Analysis

The time complexity is exponential, since there are some overlapping subproblems shown in the recursion tree. $O(2^{log(n)})$

2.38.3.3 Algorithm

recursion 1.1.5

2.38.4 Java Code - Recursion}

```
int solution(int[] array) {
    return rec(array, 1);
}
```

```
int rec(int[] array, int multiplier) {
    int len = array.length;

    if(len == 1) {
        return array[0] * multiplier;
    }

    int left = array[0] * multiplier + rec(Arrays.copyOfRange(array, 1, len), multiplier + 1);
    int right = array[len - 1] * multiplier + rec(Arrays.copyOfRange(array, 0, len - 1), multipli

    return Math.max(left, right);
}
```

2.38.5 Solution - Recursion with Memoization}

2.38.5.1 Walkthrough

We can derive $multiplier = 1 + n - (j - i), 0 \leq i < j \leq n$,

The sum of product array[i:j] is either computed directly (the base case), or it can be computed in constant time from the already known sum of array[i+1:j] and array[i:j-1].

2.38.5.2 Analysis

If we use dynamic programming and memorize all of these subresults, we will get an algorithm with $O(n^2)$ time complexity.

2.38.5.3 Algorithm

recursion 1.1.5, memoization 1.1.8

2.38.6 Java Code - Recursion with Memoization}

```
int solution(int[] array) {
    int len = array.length;
    int[][] dp = new int[len + 1][len + 1];

    return memoization(array, 0, len, dp);
}

// 0 <= start < stop <= n
```

```
// start inclusive, stop exclusive
int memoization(int[] array, int start, int stop, int[][] dp) {
    int multiplier = 1 + array.length - (stop - start);

    if(dp[start][stop] > 0) {
        //visited subproblem
        return dp[start][stop];
    }

    if(stop - start == 1) {
        return array[start] * multiplier;
    }

    int left = multiplier * array[start] + memoization(array, start + 1, stop, dp);
    int right = multiplier * array[stop - 1] + memoization(array, start, stop - 1, dp)

    int result = Math.max(left, right);

    dp[start][stop] = result;

    return result;
}
```

2.38.7 Solution - Dynamic Programming with Tabulation}

2.38.7.1 Walkthrough

We can derive $multiplier = 1 + n - (j - i), 0 \leq i < j \leq n$,

As an alternative, we can use tabulation and start by filling up the memo table. Note that the order of computation matters: to compute the value memo[i][j], the values of memo[i+1][j] and memo[i][j-1] must first be known.

Here is the final view of table from the example:

```
0   4   11  20  30
0   0   8   18  29
0   0   0   12  25
0   0   0   0   16
0   0   0   0   0
```

2.38.7.2 Analysis

2.38.7.3 Algorithm

dp 1.1.7, tabulation 1.1.9

2.38.8 Java Code - Dynamic Programming with Tabulation}

```
int solution(int[] array) {
    int len = array.length;
    int[][] dp = new int[len + 1][len + 1];

    return tabulation(array, dp, len);
}

int tabulation(int[] array, int[][] dp, int len) {
    for(int i = 0; i < len; i++) {
        dp[i][i + 1] = len * array[i];
    }

    for (int i = len - 1; i >= 0; i--) {
        for(int j = i + 2; j <= len; j++) {
            int multiplier = 1 + len - (j - i);
            int left = multiplier * array[i] + dp[i + 1][j];
            int right = multiplier * array[j - 1] + dp[i][j - 1];
            int result = Math.max(left, right);
            dp[i][j] = result;
        }
    }

    return dp[0][len];
}
```

2.39 Shortest Word Distance / / }

2.39.1 Description

Given a list of words and two words word1 and word2, return the shortest distance between these two words in the list.

2.39.2 Example

words = [“practice”, “makes”, “perfect”, “coding”, “makes”]

word1 = “coding”, word2=“practice” result is 3

word1 = “makes”, word2=“coding” result is 1

2.39.3 Solution

2.39.3.1 Walkthrough

Record the indices for each word. Retrieve the minimum distance between two group of indices.

2.39.3.2 Analysis

Each word in the list is visited once, and each index in both list is visited once. Thus time complexity is O(n), where the Auxiliary Space is O(n) for storing indices.

2.39.3.3 Algorithm

2.39.4 Java Code

```
public int shortestDistance(String[] words, String word1, String word2) {
    // list of indices for word1 and word2 in the array respectively.
    List<Integer> list1 = new ArrayList<>();
    List<Integer> list2 = new ArrayList<>();

    for(int i = 0; i < words.length; i++) {
        if(words[i].equals(word1)) {
            list1.add(i);
        } else if(words[i].equals(word2)) {
            list2.add(i);
        }
    }

    int min = words.length;
    for(int i = 0, j = 0; i < list1.size() && j < list2.size();) {
        int index1 = list1.get(i);
        int index2 = list2.get(i);
```

```
        min = Math.min(min, Math.abs(index1 - index2));

        if(index1 < index2) {
            i++;
        } else if(index1 > index2) {
            j++;
        } else {
            // comparing the same indices
            return 0;
        }
    }

    return min;
}
```

Chapter 3

Matrix

3.1 Design a Tic-tac-toe / LeetCode 348 / Medium}

3.1.1 Description}

Design a Tic-tac-toe game that is played between two players on a n x n grid.

3.1.2 Example}

3.1.3 Solution}

3.1.3.1 Walkthrough}\

Check if the player win for the same row, column, forward and back diagonals.

3.1.3.2 Analysis}\

All entries in matrix will be visited, that is, O(rows * cols)

3.1.3.3 Algorithm}\

3.1.4 Java Code}

```
int[][] matrix;

public int move(int row, int col, int player) {
    matrix[row][col]=player;

    //check row
    boolean win = true;
    for(int i = 0; i < matrix.length; i++){
        if(matrix[row][i] != player){
            win = false;
            break;
        }
    }

    if(win) return player;

    //check column
    win = true;
    for(int i = 0; i < matrix.length; i++){
        if(matrix[i][col] != player){
            win = false;
            break;
        }
    }

    if(win) return player;

    //check back diagonal
    win = true;
    for(int i = 0; i < matrix.length; i++){
        if(matrix[i][i] != player){
            win = false;
            break;
        }
    }

    if(win) return player;

    //check forward diagonal
    win = true;
    for(int i = 0; i < matrix.length; i++){
        if(matrix[i][matrix.length-i-1] != player){
            win = false;
            break;
        }
```

```
    }

    if(win) return player;

    return 0;
}
```

3.2 Rotate Image / Leet Code 48 / Medium}

3.2.1 Description}

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.

3.2.2 Example}

Given input matrix, and rotate the input matrix in-place.

```
[1,2,3],          [7,4,1],
[4,5,6],  ==>     [8,5,2],
[7,8,9]           [9,6,3]
```

3.2.3 Solution}

3.2.3.1 Walkthrough}\

Take this matrix as example,

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

first swap the symmetric elements via top-left to bottom-right diagnal,

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

and swap columns via centeral vertical line.

```
[7,|4|,1],
[8,|5|,2],
```

```
[9,|6|,3]
```

3.2.3.2 Analysis}\

Time complexity is $O(n^2)$ as there are two loops

3.2.3.3 Algorithm}\

3.2.4 Java Code}

```java
public void rotate(int[][] matrix) {
    //Swap symmetric element via diagnal
    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;
        }
    }
    //Swap columns via centeral vertical line
    for (int i = 0, j = matrix.length - 1; i < matrix.length / 2; i++, j--) {
        for (int k = 0; k < matrix.length; k++) {
            int temp = matrix[k][i];
            matrix[k][i] = matrix[k][j];
            matrix[k][j] = temp;
        }
    }
}
```

3.3 Spiral Matrix / LeetCode 54 / Medium}

3.3.1 Description}

Given a matrix of m x n elements (m rows, n columns), return all elements of the matrix in spiral order.

3.3.2 Example}

Input:

```
1, 2, 3 ,
4, 5, 6 ,
7, 8, 9
```

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

Input:

```
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] ### Solution} #### Walkthrough}\
While walking through the circles, track current left, right, top, and bottom positions.

```
(r)
    0  1  2   (c)
 0  -  -  -
 1  |  >  |
 2  -  -  |
```

```
\*  Going rightward
\*  Going downward
\*  Going leftward
\*  Going upward
```

Be sure to check for duplicated column or row before going for each direction.

3.3.2.1 Analysis}\

Time complexity is O(rows * cols), since all elements are visited once

3.3.2.2 Algorithm}\

3.3.3 Java Code}

```
public List spiralOrder(int[][] matrix) {
    List result = new ArrayList<>();

    if(matrix.length == 0) {
        return result;
    }

    int rows = matrix.length;
    int cols = matrix[0].length;
```

```
    // position of left, right, top and buttom border
    int left = 0;
    int right = cols - 1;
    int top = 0;
    int bottom = rows - 1;

    while( result.size() < (cols * rows) ) {
        //traverse a circle

        // avoid duplicated row
        if(top > bottom) {
            break;
        }

        //going rightward
        for(int i = left; i <= right; i++){
            result.add(matrix[top][i]);
        }
        top++;

        // avoid duplicated column
        if(left > right) {
            break;
        }

        //going downward
        for(int i = top; i <= bottom; i++){
            result.add(matrix[i][right]);
        }
        right--;

        // avoid duplicated row
        if(top > bottom) {
            break;
        }

        // going leftward
        for(int i = right; i >= left; i--){
            result.add(matrix[bottom][i]);
        }
        bottom--;

        // avoid duplicated column
        if(left > right) {
            break;
```

```
        }

        // going upward
        for(int i = bottom; i >= top; i--){
            result.add(matrix[i][left]);
        }
        left++;
    }

    return result;
}
```

3.4 Search a 2D Matrix / LeetCode 74 / Medium}

3.4.1 Description}

Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:

```
\*  Integers in each row are sorted from left to right.
\* The first integer of each row is greater than the last integer of the previous row.
```

3.4.2 Example}

Input:

```
[1,   3,  5,  7],
[10, 11, 16, 20],
[23, 30, 34, 50]
```

target = 3 Output: true

Input:

```
[1,   3,  5,  7],
[10, 11, 16, 20],
[23, 30, 34, 50]
```

target = 13 Output: false ### Solution} #### Walkthrough}\ We can treat the 2D matrix as the one large array, and search the element using binary search. We only need to compute middle of row or column for each divide-and-conquer turn.

3.4.2.1 Analysis}\

The overall time complexity is O(log(rows * cols))

3.4.2.2 Algorithm}\

recursion 1.1.10

3.4.3 Java Code}

```
public boolean searchMatrix(int[][] matrix, int target) {
    if(matrix == null || matrix.length == 0 || matrix[0].length == 0) {
        return false;
    }

    int rows = matrix.length;
    int cols = matrix[0].length;

    int start = 0;
    int end = rows * cols - 1;

    while(start <= end) {
        int mid = (end - start) / 2 + start;
        int midRow = mid / cols;
        int midCol = mid % cols;

        if(matrix[midRow][midCol] == target) {
            return true;
        } else if(matrix[midRow][midCol] < target) {
            start = mid + 1;
        } else {
            //matrix[midRow][midCol] > target
            end = mid - 1;
        }
    }

    return false;
}
```

3.5 Search a 2D Matrix II / LeetCode 240 / Medium}

3.5.1 Description}

Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:

* Integers in each row are sorted in ascending from left to right.
* Integers in each column are sorted in ascending from top to bottom.

3.5.2 Example}

java [[1, 4, 7, 11, 15], [2, 5, 8, 12, 19], [3, 6, 9, 16, 22], [10, 13, 14, 17, 24], [18, 21, 23, 26, 30]], 5 returns true

3.5.3 Solution}

3.5.3.1 Walkthrough}\

Traverse the matrix by

* locate row when target is smaller than m[r][c]
* locate col when target is larger than m[r][c]

3.5.3.2 Analysis}\

A portion of cell is visited, O(m + n) - instead of every cell (m * n)

3.5.3.3 Algorithm}\

3.5.4 Java Code}

```java
public boolean searchMatrix(int[][] matrix, int target) {
    if(matrix == null || matrix.length == 0) {
        return false;
    }

    int rows = matrix.length-1;
    int cols = matrix[0].length-1;
```

```
    int r = rows;
    int c = 0;

    while(r >= 0 && c <= cols){
        if(target < matrix[r][c]) {
            r--;
        } else if(target > matrix[r][c]) {
            c++;
        } else {
            return true;
        }
    }

    return false;
}
```

3.6 Minimum Path Sum / LeetCode 64 / Medium}

3.6.1 Description}

Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right which minimizes the sum of all numbers along its path. You can only move either down or right at any point in time. ### Example} Input:

```
[1],[3],[1],
 1,  5, [1],
 4,  2, [1]
```

Output: 7, because the path $1- > 3- > 1- > 1- > 1$ minimizes the sum.

3.6.2 Solution - Recursive}

3.6.2.1 Walkthrough}\

Time Complexity is exponential.

See the above recursion tree, there are many nodes which apear more than once. Final cost matrix:

```
[1], [4], [5]
2,   7,   [6]
6,   8,   [7]
```

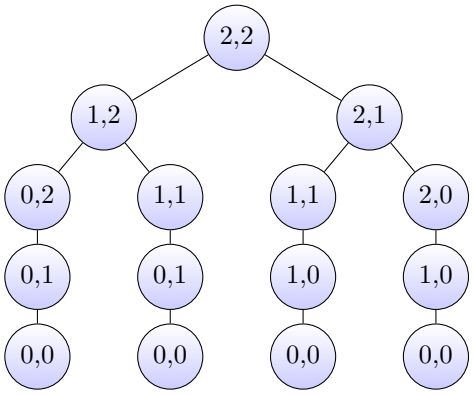


Figure 3.1: Some caption.

3.6.2.2 Analysis}\

There will be $2^{h+1} - 1 = 2^{log(m*n)+1} - 1$ subproblems, including the overlapping subproblems, which means the same subproblem has been computed again and gain. Thus, the time complexity is $O(2^{log(m*n)})$

3.6.2.3 Algorithm}\

recursive **??**

3.6.3 Java Code - Recursive}

```
public int minPathSum(int[][] grid) {
    if(grid == null || grid.length==0) {
        return 0;
    }

    return rec(0,0,grid);
}

public int rec(int i, int j, int[][] grid){
    int rows = grid.length;
    int cols = grid[0].length;


    if(i == (rows - 1) && j == (cols - 1)){
        //at bottom right
        return grid[i][j];
```

```
    } else if(i < (rows - 1) && j == (cols - 1)) {
        //at last column, moving downward
        return grid[i][j] + dfs(i+1, j, grid);
    } else if(i == (rows - 1) && j < (cols - 1)) {
        //at last row, moving rightward
        return grid[i][j] + dfs(i, j+1, grid);
    } else {
        //other place, get min of downward and rightward
        int r1 = grid[i][j] + rec(i+1, j, grid);
        int r2 = grid[i][j] + rec(i, j+1, grid);
        return Math.min(r1,r2);
    }
}
```

3.6.4 Solution - Dynamic Programming}

3.6.4.1 Walkthrough}\

```
\*  create a cost matrix of the same size
\*  init cost[0][0] to be grid[0][0]
\* init first row and first column, previous cell cost plus current cell cost
\* For each other cell, determine the minimum cost of previous top or left cell plus curren
```

3.6.4.2 Analysis}\

Complexity: O(n * m) since each cell is visited once

3.6.4.3 Algorithm}\

dp 1.1.7

3.6.5 Java Code - Dynamic Programming}

```
public int minPathSum(int[][] grid) {
    if(grid == null || grid.length == 0) {
        return 0;
    }

    int rows = grid.length;
    int cols = grid[0].length;
```

```
    int[][] dp = new int[rows][cols];
    dp[0][0] = grid[0][0];

    // initialize first row
    for(int c = 1; c < cols; c++){
        dp[0][c] = dp[0][c - 1] + grid[0][c];
    }

    // initialize first column
    for(int r = 1; r < rows; r++){
        dp[r][0] = dp[r - 1][0] + grid[r][0];
    }

    // fill up the dp table
    for(int i = 1; i < rows; i++){
        for(int j = 1; j < cols; j++){
            int minNeighbor = Math.min(dp[i - 1][j], dp[i][j - 1]);
            dp[i][j] = minNeighbor + grid[i][j];
        }
    }

    return dp[rows - 1][cols - 1];
}
```

3.7 Unique Paths / LeetCode 62 / Medium}

3.7.1 Description}

A robot is located at the top-left corner of a m x n grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid. How many possible unique paths are there?

3.7.2 Example}

Given a matrix of size 2 * 3:

```
[],[],[]
[],[],[]
```

There are 3 unique path:

```
\*  (0,0) - (1,0) - (1,1) - (1,2)
\*  (0,0) - (0,1) - (1,1) - (1,2)
\*  (0,0) - (0,1) - (0,2) - (1,2)
```

3.7.3 Solution - Recursion}

3.7.3.1 Walkthrough}\

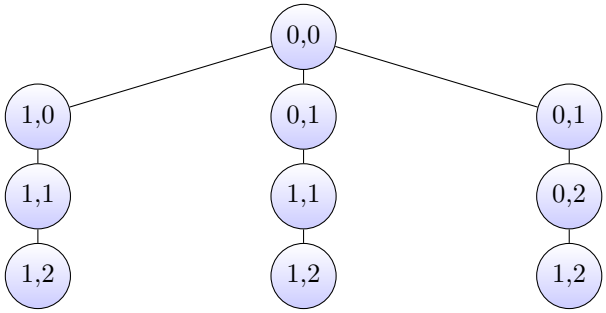


Figure 3.2: Some caption.

3.7.3.2 Analysis}\

Naive recursive strategy results in overlapping subproblems. The time complexity is $O(2^{log(m*n)})$

3.7.3.3 Algorithm}\

recursive **??**

3.7.4 Java Code - Recursion}

```java
public int uniquePaths(int m, int n) {
    return rec(0, 0, m, n);
}
int rec(int i, int j, int rows, int cols){

    if(i == (rows - 1) && j == (cols - 1)){
        //at bottom right
        return 1;
    } else if(i < (rows - 1) && j == (cols - 1)) {
```

```
        //at last column, moving downward
        return rec(i+1, j, rows, cols);
    } else if(i == (rows - 1) && j < (cols - 1)) {
        //at last row, moving rightward
        return rec(i, j+1, rows, cols);
    } else {
        //other place, get min of downward and rightward
        return rec(i+1, j, rows, cols) + rec(i, j+1, rows, cols);
    }
}
```

3.7.5 Solution - Dynamic Programming}

3.7.5.1 Walkthrough}\

Number of unique path to current cell equals to the sum of paths to the previous cells. For example, the matrix representing # of path to reach current cell. The weight at (1,2) equals to sum of (0,2) and (1,1)

```
1 1 1
1 2 3
```

3.7.5.2 Analysis}\

Time complexity is O(n * m) since each cell is visited once. A cache of size O(n * m) is used to store visited cell value.

3.7.5.3 Algorithm}\

dp 1.1.7

3.7.6 Java Code - Dynamic Programming}

```
public int uniquePaths(int m, int n) {
    if(m == 0 || n == 0) {
        return 0;
    }
    if(m == 1 || n == 1) {
        return 1;
    }

    int[][] dp = new int[m][n];
```

```
    //init first column
    for(int i = 0; i < m; i++){
        dp[i][0] = 1;
    }

    //init first row
    for(int j = 0; j < n; j++){
        dp[0][j] = 1;
    }

    //fill up the rest of table
    for(int i = 1; i < m; i++){
        for(int j = 1; j < n; j++){
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
        }
    }

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

3.8 Word Search / LeetCode 79 / Medium}

3.8.1 Description}

Given a 2D board and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

3.8.2 Example}

```
board =
[
['A','B','C','E'],
['S','F','C','S'],
['A','D','E','E']
]
```

Given word = "ABCCED", return true. Given word = "SEE", return true. Given word = "ABCB", return false.

3.8.3 Solution - DFS}

3.8.3.1 Walkthrough}\

Enumerate all possible starting position from the 2D array by using two nested loops. While traversing the character in the taget string, perform a search recursively on neighboring cells [(i, j + 1), (i, j - 1), (i + 1, j), (i - 1, j)]. If there is a hit in any four neighboring cells, return true; otherwise, return false.

3.8.3.2 Analysis}\

The enumeration cost is exponential, as there are multiple overlapping subproblems. Thus, the time complexity is $O(2^{log(n)})$

3.8.3.3 Algorithm}\

backtrack 1.1.6, dfs 1.1.1

3.8.4 Java Code - DFS}

```
private static final char VISITED_CHARACTER = '*';

public boolean exist(char[][] board, String word) {
    int numOfRows = board.length;
    int numOfCols = board[0].length;

    //Enumerate the all possible starting position from char[][]
    for (int i = 0; i < numOfRows; i++) {
        for (int j = 0; j < numOfCols; j++) {
            if (backtrack(board, i, j, word, 0)) {
                return true;
            }
        }
    }
    return false;
}

private boolean backtrack(char[][] board, int row, int col, String word, int index) {
    int numOfRows = board.length;
    int numOfCols = board[0].length;

    //every other character has matched
    if (index == word.length()) {
```

```
        return true;
    }

    //exceed the boundary
    if (row < 0 || row >= numOfRows || col < 0 || col >= numOfCols) {
        return false;
    }

    //if current char is not equal
    if (board[row][col] != word.charAt(index)) {
        return false;
    }

    // 'nullify' current position
    board[row][col] ^= VISITED_CHARACTER;

    //result of next move from four directions.
    boolean result = backtrack(board, row + 1, col, word, index + 1)
                || backtrack(board, row - 1, col, word, index + 1)
                || backtrack(board, row, col + 1, word, index + 1)
                || backtrack(board, row, col - 1, word, index + 1);

    //nullify back
    board[row][col] ^= VISITED_CHARACTER;

    return result;
}
```

3.9 Number of Islands / LeetCode 200 / Medium}

3.9.1 Description}

Given a 2d grid map of '1's (land) and '0's (water), count 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.

3.9.2 Example}

```
11110
11010
```

11000
00000

= 1

11000
11000
00100
00011

= 3 ### Solution - DFS} #### Walkthrough}\ Traverse all cell and once we find a land tile, fill the adjacent land recursively. Count each land tile we have encountered.

3.9.2.1 Analysis}\

Everyone cell in matrix is visited. Thus, is O(rows * cols)

3.9.2.2 Algorithm}\

dfs 1.1.1

3.9.3 Java Code - DFS}

```
public int numIslands(char[][] grid) {
    if(grid==null || grid.length==0||grid[0].length==0)
        return 0;

    int rows = grid.length;
    int cols = grid[0].length;

    int count = 0;
    for(int i = 0; i < rows; i++){
        for(int j = 0; j < cols; j++){
            if(grid[i][j] == '1'){
                count++;
                fill(grid, i, j);
            }
        }
    }


    return count;
}
```

```
public void fill(char[][] grid, int i, int j){
    int rows = grid.length;
    int cols = grid[0].length;

    if(i < 0 || i >= rows || j < 0 || j >= cols || grid[i][j] != '1')
        return;

    grid[i][j] = 'X';

    // recursively fill the adjacent land
    fill(grid, i - 1, j);
    fill(grid, i + 1, j);
    fill(grid, i, j - 1);
    fill(grid, i, j + 1);
}
```

3.10 Surrounded Regions / LeetCode 130 / Medium}

Given a 2D board containing 'X' and 'O' (the letter O), capture all regions surrounded by 'X'. A region is captured by flipping all 'O's into 'X's in that surrounded region.

Surrounded regions shouldn't be on the border, which means that any 'O' on the border of the board are not flipped to 'X'. Any 'O' that is not on the border and it is not connected to an 'O' on the border will be flipped to 'X'. Two cells are connected if they are adjacent cells connected horizontally or vertically.

3.10.1 Description}

3.10.2 Example}

```
X X X X
X O O X
X X O X
X O X X
```

to

```
X X X X
X X X X
X X X X
X O X X
```

3.10.3 Solution - DFS}

3.10.3.1 Walkthrough}\

First preserve the border tile (or tile connected to a border ultimately) with '#'.

```
X X X X
X O O X
X X O X
X # X X
```

Traverse all cell by changing the inland tile ('O') to 'X' and reset '#' back to 'O'.

```
X X X X
X X X X
X X X X
X O X X
```

3.10.3.2 Analysis}\

All cell is visited a few times, thus, O(rows * cols). #### Algorithm}\ dfs 1.1.1

3.10.4 Java Code - DFS}

```
public void solve(char[][] board) {
    if(board == null || board.length==0)
        return;

    int rows = board.length;
    int cols = board[0].length;

    //preserve O's on left & right boarder
    for(int i = 0;i < rows;i++){
        if(board[i][0] == 'O'){
            preserve(board, i, 0);
        }

        if(board[i][cols - 1] == 'O'){
            preserve(board, i, n - 1);
        }
    }

    //preserve O's on top & bottom boarder
```

```
    for(int j = 0; j < cols; j++){
        if(board[0][j] == 'O'){
            preserve(board, 0, j);
        }

        if(board[rows - 1][j] == 'O'){
            preserve(board, m - 1, j);
        }
    }

    //process the board
    for(int i = 0;i < rows;i++) {
        for(int j = 0; j < cols; j++) {
            if(board[i][j] == 'O') {
                board[i][j] = 'X';
            } else if(board[i][j] == '#') {
                // change the preserved border tile back to 'O'
                board[i][j] = 'O';
            }
        }
    }
}

public void preserve(char[][] board, int i, int j){
    int rows = board.length;
    int cols = board[0].length;

    if(i < 0 || i >= rows || j < 0 || j >= cols || board[i][j] != 'O')
        return;

    board[i][j] = '#';

    // recursively fill the adjacent land
    preserve(board, i - 1, j);
    preserve(board, i + 1, j);
    preserve(board, i, j - 1);
    preserve(board, i, j + 1);
}
```

3.11 Best Meeting Point / LeetCode 296 / Hard}

3.11.1 Description}

A group of two or more people wants to meet and minimize the total travel distance. You are given a 2D grid of values 0 or 1, where each 1 marks the home of someone in the group. The distance is calculated using Manhattan Distance, where distance(p1, p2) = $|p2.x - p1.x| + |p2.y - p1.y|$

3.11.2 Example}

For example, given three people living at (0,0), (0,4), and (2,2):

```
1 - 0 - [0] - 0 - 1
|   |     |     |   |
0 - 0 -   0 -  0 - 0
|   |     |     |   |
0 - 0 -   1 -  0 - 0
```

The point (0,2) is an ideal meeting point, as the total travel distance of 2+2+2=6 is minimal. So return 6. ### Solution} #### Walkthrough}\ The median points for row and col should be the optimal meeting point. We could compute the distance by substracting the coordinate of people by the median X and Y.

3.11.2.1 Analysis}\

Each cell in matrix will be visited at least once. Thus, O(rows * cols).

3.11.2.2 Algorithm}\

3.11.3 Java Code}

```
public int minTotalDistance(int[][] grid) {
    int rows = grid.length;
    int cols = grid[0].length;

    List<Integer> cols = new ArrayList<Integer>();
    List<Integer> rows = new ArrayList<Integer>();
    for(int i = 0; i < rows; i++){
        for(int j = 0; cols < n; j++){
            if(grid[i][j] == 1){
```

```
                cols.add(j);
                rows.add(i);
            }
        }
    }

    int dist = 0;

    int medianRow = rows.get(rows.size() / 2);
    for(Integer r: rows){
        dist += Math.abs(r - medianRow);
    }

    //sort col position so that we could get median col properly
    Collections.sort(cols);
    int medianCol = cols.get(cols.size() / 2);

    for(Integer c: cols){
        dist += Math.abs(c - medianCol);
    }

    return dist;
}
```

Chapter 4

Tree

4.0.1 Breadth First Traversal

The algorithm starts at the tree root, and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

4.0.1.1 Algorithm\

bfs 1.1.2

4.0.1.2 Typical Implementation - Java

```
public boolean breadthFirstTraversal(TreeNode root) {
    Queue<TreeNode> queue = new LinkedList();

    while (!queue.isEmpty()) {
        TreeNode node = queue.poll();

        if (node.left != null) {
            queue.offer(node.left);
        }

        if (node.right != null) {
            queue.offer(node.right);
        }
    }
```

```
    return false;
}
```

4.0.1.3 Complexity Analysis

- Time Complexity: Breadth-first search visits every vertex once and checks every edge in the graph once. Therefore, the runtime complexity is $O(|V|+|E|)$. In simplicity, O(n) since $|V| = n$
- Auxiliary Space: In Breadth First traversal, visited node in different level is stored in a queue one by one. Extra Space required is O(w) where w is maximum width of a level in binary tree.

4.0.2 Depth First Traversal

The algorithm starts at the root node and explores as far as possible along each branch.

4.0.2.1 Algorithm

dfs 1.1.1

4.0.2.2 Type of Depth First Traversal

- Depth First InOrder Traversal
- Depth First PreOrder Traversal
- Depth First PostOrder Traversal

4.0.2.3 Complexity Analysis

- Time Complexity: Depth-first search visits every vertex once and checks every edge in the graph once. Therefore, the runtime complexity is $O(|V|+|E|)$. In simplicity, O(n) since $|V| = n$
- Auxiliary Space: In Depth First Traversals, stack (or function call stack) stores all ancestors of a node. Extra Space required is O(h) where h is maximum height of binary tree.

4.0.3 Selection Strategy

- Extra Space can be one factor
- Depth First Traversals are typically recursive and recursive code requires function call overheads.
- The most important points is, BFS starts visiting nodes from root while DFS starts visiting nodes from leaves. So if our problem is to search something that is more likely to closer to root, we would prefer BFS. And if the target node is close to a leaf, we would prefer DFS
- Maximum Width of a Binary Tree at depth (or height) h can be 2h where h starts from 0. So the maximum number of nodes can be at the last level. And worst case occurs when Binary Tree is a perfect Binary Tree with numbers of node
- It is evident from above points that extra space required for Level order traversal is likely to be more when tree is more balanced and extra space for Depth First Traversal is likely to be more when tree is less balanced.

4.0.4 Binary Search Tree

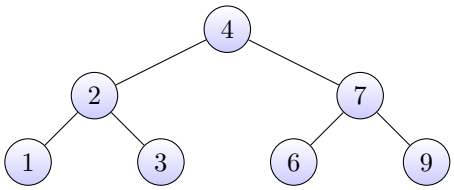


Figure 4.1: Some caption.

A BST is a binary tree where nodes are ordered in the following way:

- each node contains one key (also known as data)
- the keys in the left subtree are less then the key in its parent node
- the keys in the right subtree are greater the key in its parent node
- duplicate keys are not allowed.

4.0.4.1 Algorithm

bst 1.1.11

4.0.4.2 Complexity Analysis

The particular kind of binary tree is optimized in a way that only a PARTIAL nodes along a path will be visited during search, the general time complexity is reduced to O(h) where h = log(n).

Since a binary search tree with n nodes has a minimum of O(log(n)) levels, it takes at least O(log(n)) comparisons to find a particular node. However, in the edge cases where tree is in a list form, the worst time complexity is O(n)

4.1 Balanced Binary Tree / Leet Code 110 / Easy

4.1.1 Description

Given a binary tree, determine if it is height-balanced. For this problem, a height-balanced binary tree is defined as: a binary tree in which the depth of the two subtrees of every node never differ by more than 1.

4.1.2 Example

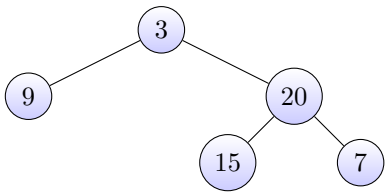


Figure 4.2: Some caption.

Return true.

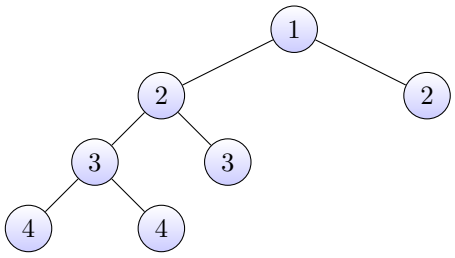


Figure 4.3: Some caption.

Return false.

4.1.3 Solution

4.1.3.1 Walkthrough\

Remember current height and recursively compute the height for the left and right subtree. In addition, evaluate if the absolute value of height of two subtrees is greater than 1. If yes, return false; true, otherwise.

4.1.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.1.3.3 Algorithm\

dfs 1.1.1

4.1.4 Java Code

```java
public boolean isBalanced(TreeNode root) {
    if (root == null) {
        return true;
    }

    // left height | right height > 1
    if (getHeight(root) == -1) {
        return false;
    }

    return true;
}

public int getHeight(TreeNode root) {
    if (root == null) {
        return 0;
    }

    int lHeight = getHeight(root.left);
    int rHeight = getHeight(root.right);

    if (lHeight == -1 || rHeight == -1) {
        return -1;
    }
```

```
    if (Math.abs(left - right) > 1) {
        return -1;
    }

    return Math.max(left, right) + 1;
}
```

4.2 Invert Binary Tree / Leet Code 226 Easy

4.2.1 Description

Invert a binary tree

4.2.2 Example

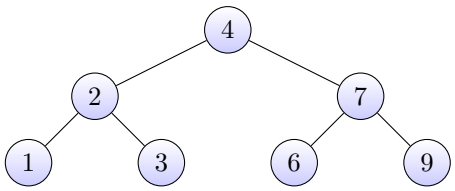


Figure 4.4: Some caption.

inverts

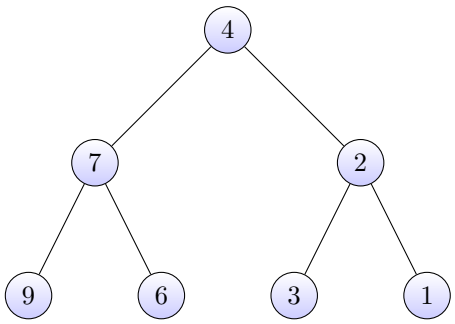


Figure 4.5: Some caption.

4.2.3 Solution - DFS

4.2.3.1 Walkthrough\

Swap the left and right subtree with recursive strategy.

4.2.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.2.3.3 Algorithm\

dfs 1.1.1

4.2.4 Java Code - DFS

```
public TreeNode invertTree(TreeNode root) {
    if (root == null) {
        return null;
    }

    //swap left and right subtrees
    TreeNode origRight = root.right;
    root.right = invertTree(root.left);
    root.left = invertTree(origRight);

    return root;
}
```

4.2.5 Solution - BFS

4.2.5.1 Walkthrough\

Swap the left and right subtree with iterative (BFS) strategy

4.2.5.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.2.5.3 Algorithm\

bfs 1.1.2

4.2.6 Java Code - BFS

```
public TreeNode invertTree(TreeNode root) {
    if (root == null) {
        return null;
    }

    Queue<TreeNode> queue = new LinkedList<TreeNode>();
    queue.offer(root);

    while(!queue.isEmpty()) {
        TreeNode node = queue.poll();

        //swap left and right subtrees
        TreeNode origRight = node.right;
        node.right = node.left;
        node.left = origRight;

        if (node.left != null) {
            queue.offer(node.left);
        }
        if (node.right != null) {
            queue.offer(node.right);
        }
    }
    return root;
}
```

4.3 Symmetric Tree / Leet Code 101 / Easy

4.3.1 Description

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

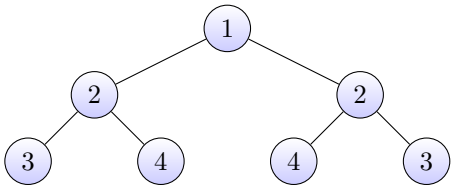


Figure 4.6: Some caption.

4.3.2 Example

is symmetric.

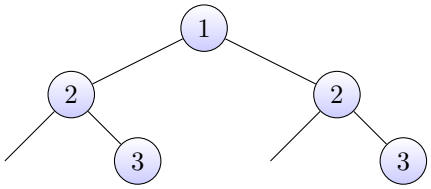


Figure 4.7: Some caption.

is NOT symmetric.

4.3.3 Solution

4.3.3.1 Walkthrough\

If they are symmetric, compare its value and recursively call its grand children node in symmetric manner:

- left.left vs right.right
- left.right vs right.left

4.3.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.3.3.3 Algorithm\

dfs 1.1.1

4.3.4 Java Code

```
public boolean isSymmetric(TreeNode root) {
    if (root == null) {
        return true;
    }
    return isSymmetric(root.left, root.right);
}
private boolean isSymmetric(TreeNode left, TreeNode right) {
    if (left == null && right == null) {
        return true;
    } else if (left == null || right == null) {
        return false;
    } else {
        return left.val == right.val && isSymmetric(left.left, right.right)
&& isSymmetric(left.right, right.left);
    }

}
```

4.4 Binary Tree Level Order Traversal / Leet Code 102 / Medium

4.4.1 Description

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

4.4.2 Example

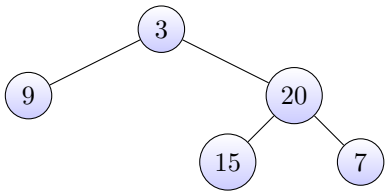


Figure 4.8: Some caption.

return its level order traversal as:

```
[
```

```
    [3],
    [9,20],
    [15,7]
]
```

4.4.3 Solution - BFS with Two Loops

4.4.3.1 Walkthrough\

Create a queue and push node of the same level in a two dimensional array list while traversing the tree with BFS algorithm.

4.4.3.2 Analysis\

Breadth first traversal has time complexity of O(n), as every node is visited and thus the space complexity is also O(n).

4.4.3.3 Algorithm\

bfs 1.1.2

4.4.4 Java Code - BFS with Two Loops

```java
public List<List<Integer>> levelOrder(TreeNode root) {
    Queue<TreeNode> queue = new LinkedList<>();
    List<List<Integer>> levels = new ArrayList<>();

    if(root == null) {
        return levels;
    }

    queue.offer(root);

    while(!queue.isEmpty()) {
        //number of element for this level
        int currentLevelSize = queue.size();

        List<Integer> level = new ArrayList<>();
        for(int i = 0; i < currentLevelSize; i++) {
            TreeNode node = queue.poll();
            level.add(node.val);
```

```
            // push all elements for next level
            if (node.left != null) {
                queue.offer(node.left);
            }

            if (node.right != null) {
                queue.offer(node.right);
            }
        }

        //add all elements for this level
        levels.add(level);
    }

    return levels;
}
```

4.4.5 Solution - BFS with One Loops

4.4.5.1 Walkthrough\

Create a queue and push node of the same level in a two dimensional array list while traversing the tree with BFS algorithm.

4.4.5.2 Analysis\

Breadth first traversal has time complexity of O(n), as every node is visited and thus the space complexity is also O(n).

4.4.5.3 Algorithm\

bfs 1.1.2

4.4.6 Java Code - BFS with One Loop

```
public List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> levels = new ArrayList<>();
    if (root == null) {
        return levels;
    }
    Queue<TreeNode> currentLevelQueue = new LinkedList<TreeNode>();
```

```
    Queue<TreeNode> nextLevelQueue = new LinkedList<TreeNode>();

    currentLevelQueue.offer(root);
    //init level to null
    List<Integer> level = null;

    while(!currentLevelQueue.isEmpty()) {
        TreeNode node = currentLevelQueue.poll();

        //starting a new level
        if (level == null) {
            level = new LinkedList<Integer>();
            levels.add(level);
        }
        level.add(node.val);

        //push children to next level queue
        if (node.left != null) {
            nextLevelQueue.offer(node.left);
        }
        if (node.right != null) {
            nextLevelQueue.offer(node.right);
        }

        //swap queue and next, where next collects all nodes for next level
        if (currentLevelQueue.isEmpty()) {
            Queue<TreeNode> temp = currentLevelQueue;
            currentLevelQueue = nextLevelQueue;
            nextLevelQueue = temp;
            //init level to null
            level = null;
        }
    }
    return levels;
}
```

4.4.7 Solution - DFS

4.4.7.1 Walkthrough\

We could traverse the tree recursively with **PreOrder** strategy, and additionally pass current level into recursive method.

4.4.7.2 Analysis\

DFS has time complexity of O(n) as every node is visited and thus Auxiliary Space is also O(n).

4.4.7.3 Algorithm\

dfs 1.1.1

4.4.8 Java Code - DFS

```
public List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> ans = new ArrayList<>();
    traverse(root, 0, ans);

    return ans;
}

private void traverse(TreeNode root, int level, List<List<Integer>> ans) {
    if(root == null){
        return;
    }

    if(ans.size()<=level){
        ans.add(new ArrayList<>());
    }

    ans.get(level).add(root.val);

    traverse(root.left, level + 1, ans);
    traverse(root.right, level + 1, ans);
}
```

4.5 Binary Tree Level Order Traversal II / Leet Code 107 / Medium

4.5.1 Description

Given a binary tree, return the bottom-up level order traversal of its nodes' values. (ie, from left to right, level by level from leaf to root).

4.5.2 Example

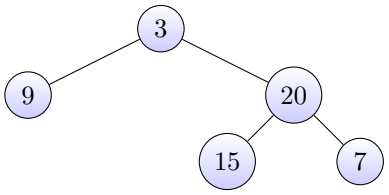


Figure 4.9: Some caption.

return its level order traversal as:

```
[
    [15,7],
    [9,20],
    [3]
]
```

4.5.3 Solution

4.5.3.1 Walkthrough\

Create a queue and push node of the same level in a two dimensional array list while traversing the tree with BFS algorithm. In addition, we only insert nodes at the first position at current level list.

4.5.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.5.3.3 Algorithm\

bfs 1.1.2

4.5.4 Java Code - Two Loops

```
public List<List<Integer>> levelOrderBottom(TreeNode root) {
    Queue<TreeNode> queue = new LinkedList<>();
    List<List<Integer>> levels = new ArrayList<>();

    if(root == null) {
```

```
        return levels;
    }

    queue.offer(root);

    while(!queue.isEmpty()) {
        //number of element for this level
        int currentLevelSize = queue.size();

        List<Integer> level = new ArrayList<>();
        for(int i = 0; i < currentLevelSize; i++) {
            TreeNode node = queue.poll();
            level.add(node.val);

            // push all elements for next level
            if (node.left != null) {
                queue.offer(node.left);
            }

            if (node.right != null) {
                queue.offer(node.right);
            }
        }

        // insert the head of the list
        levels.add(0, level);
    }

    return levels;
}
```

4.5.4.1 Walkthrough\

Create a queue and push node of the same level in a two dimensional array list while traversing the tree with BFS algorithm. In addition, we only insert nodes at the first position at current level list.

4.5.4.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.5.4.3 Algorithm\

bfs 1.1.2

4.5.5 Java Code - One Loop

```
public List<List<Integer>> levelOrderBottom(TreeNode root) {
    List<List<Integer>> levels = new ArrayList<>();
    if (root == null) {
        return levels;
    }

    Queue<TreeNode> currentLevelQueue = new LinkedList<TreeNode>();
    Queue<TreeNode> nextLevelQueue = new LinkedList<TreeNode>();

    currentLevelQueue.offer(root);
    //init level to null
    List<Integer> level = null;

    while (!currentLevelQueue.isEmpty()) {
        TreeNode node = currentLevelQueue.poll();

        //starting a new level
        if (level == null) {
            level = new LinkedList<Integer>();
            // insert to head
            levels.add(0, level);
        }
        level.add(node.val);

        //push children to next level queue
        if (node.left != null) {
            nextLevelQueue.offer(node.left);
        }
        if (node.right != null) {
            nextLevelQueue.offer(node.right);
        }

        //swap queue and next, where next collects all nodes for next level
        if (currentLevelQueue.isEmpty()) {
            Queue<TreeNode> temp = currentLevelQueue;
            currentLevelQueue = nextLevelQueue;
            nextLevelQueue = temp;
            //init level to null
```

```
            level = null;
        }
    }
    return levels;
}
```

4.6 Binary Tree Zigzag Level Order Traversal / Leet Code 103 / Medium

4.6.1 Description

Given a binary tree, return the zigzag level order traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).

4.6.2 Example

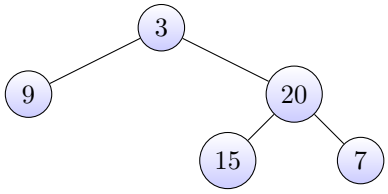


Figure 4.10: Some caption.

return its level order traversal as:

```
[
    [3],
    [20,9],
    [15,7]
]
```

4.6.3 Solution

4.6.3.1 Walkthrough\

Create a queue and push node of the same level in a two dimensional array list while traversing the tree with BFS algorithm. While pushing the value to the level, remember the current level and evaluate the zig-zag pattern. That is, insert nodes for certain level at the first position at current level list.

4.6.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.6.3.3 Algorithm\

bfs 1.1.2

4.6.4 Java Code - Two Loops

```
public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
    Queue<TreeNode> queue = new LinkedList<>();
    List<List<Integer>> levels = new ArrayList<>();

    if(root == null) {
        return levels;
    }

    queue.offer(root);
    int levelNum = 1;

    while(!queue.isEmpty()) {
        //number of element for this level
        int currentLevelSize = queue.size();
        List<Integer> level = new ArrayList<>();

        for(int i = 0; i < currentLevelSize; i++) {
            TreeNode node = queue.poll();

            if(levelNum % 2 != 0) {
                level.add(node.val);
            } else {
                level.add(0, node.val);
            }

            // push all elements for next level
            if (node.left != null) {
                queue.offer(node.left);
            }

            if (node.right != null) {
                queue.offer(node.right);
            }
```

```
        }
        levelNum++;

        levels.add(level);
    }

    return levels;
}
```

4.6.4.1 Walkthrough\

Create a queue and push node of the same level in a two dimensional array list while traversing the tree with BFS algorithm. While pushing the value to the level, remember the current level and evaluate the zig-zag pattern.

4.6.4.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.6.4.3 Algorithm\

bfs 1.1.2

4.6.5 Java Code - One Loop

```
public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
    List<List<Integer>> levels = new ArrayList<>();
    if (root == null) {
        return levels;
    }

    Queue<TreeNode> currentLevelQueue = new LinkedList<TreeNode>();
    Queue<TreeNode> nextLevelQueue = new LinkedList<TreeNode>();
    currentLevelQueue.offer(root);
    //init level = null
    List<Integer> level = null;
    int levelNum = 1;

    while (currentLevelQueue.isEmpty() == false) {
        TreeNode node = currentLevelQueue.poll();
```

```
        //starting a new level
        if (level == null) {
            level = new LinkedList<Integer>();
            levels.add(level);
        }

        if(levelNum % 2 != 0) {
            level.add(node.val);
        } else {
            level.add(0, node.val);
        }

        //push children to next level queue
        if (node.left != null) {
            nextLevelQueue.offer(node.left);
        }

        if (node.right != null) {
            nextLevelQueue.offer(node.right);
        }

        //swap queue and next, where next collects all nodes for next level
        if (currentLevelQueue.isEmpty()) {
            Queue<TreeNode> temp = currentLevelQueue;
            currentLevelQueue = nextLevelQueue;
            nextLevelQueue = temp;
            //init level = null
            level = null;
            levelNum++;
        }
    }

    return levels;
}
```

4.7 Count Complete Tree Nodes / Leet Code 222 / Medium

4.7.1 Description

Given a complete binary tree, count the number of nodes. In a complete binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. It can have between 1 and 2h nodes

inclusive at the last level h. In a full binary tree, it is completely filled at the last level, leaving the total node number of $2^{height} - 1$

4.7.2 Example

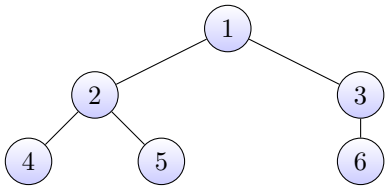


Figure 4.11: Some caption.

output 6

4.7.3 Solution

4.7.3.1 Walkthrough\

Recursively call at each node, and determine if it is a full binary tree or complete binary tree. if it is full binary tree (left height == right height), return $(2^h - 1)$, else return 1 + rec(node.left) + rec(node.right)

4.7.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.7.3.3 Algorithm\

dfs 1.1.1

4.7.4 Java Code

```
public int countNodes(TreeNode root) {
    if(root == null) {
        return 0;
    }
    int lHeight = 0, rHeight = 0;

    for (TreeNode node = root; node != null; node = node.left, lHeight++);
```

```
    for (TreeNode node = root; node != null; node = node.right, rHeight++);

    if (lHeight == rHeight) {
        /*
        * full binary tree
        * 2^leftNum - 1
        */
        return (1 << lHeight) - 1;
    } else {
        return 1 + countNodes(root.left) + countNodes(root.right);
    }
}
```

4.8 Distance of a node from the root / Firebase / Level 3

4.8.1 Description

Given the root of a Binary Tree and an integer that represents the data value of a TreeNode present in the tree, write a method - pathLengthFromRoot that returns the distance between the root and that node. You can assume that the given key exists in the tree. The distance is defined as the minimum number of nodes that must be traversed to reach the target node.

4.8.2 Example

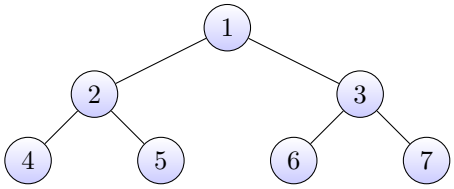


Figure 4.12: Some caption.

pathLengthFromRoot(root,5) = 3

pathLengthFromRoot(root,1) = 1

pathLengthFromRoot(root,3) = 2

4.8.3 Solution

4.8.3.1 Walkthrough\

Have a global variable of ArrayList to remember the path. Create a helper function to recursively traverse the tree. If the target does not lie either in the left or right subtree of the current node.Thus, remove current node's value from trace list.

4.8.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.8.3.3 Algorithm\

backtrack 1.1.6, dfs 1.1.1

4.8.4 Java Code

```
List<TreeNode> trace = new ArrayList<>();

int pathLengthFromRoot(TreeNode root, int target) {
    path.clear();

    boolean result = backtrack(root, target);

    if(result) {
        return trace.size();
    } else {
        return 0;
    }
}

boolean backtrack(TreeNode root, int target) {
    trace.add(root);

    if(root == null) {
        return false;
    }
    if(root.data == target) {
        //found the node
        return true;
    }
```

```
    if(backtrack(root.left, target) || backtrack(root.right, target) ) {
        return true;
    }

    //not in current path, backtrack
    trace.remove(path.size() - 1);
    return false;
}
```

4.9 Path Sum / Leet Code 112 / Easy

4.9.1 Description

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

4.9.2 Example

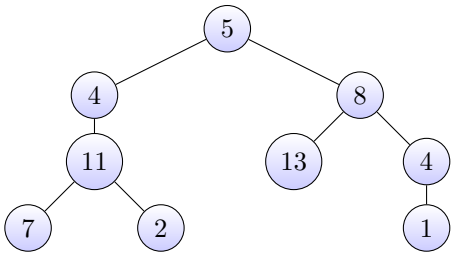


Figure 4.13: Some caption.

Given the below binary tree and sum = 22, return true, as there exist a root-to-leaf path $5- > 4- > 11- > 2$ which sum is 22.

4.9.3 Solution

4.9.3.1 Walkthrough\

Has a variable to remember accumulated sum and recursively call on its left and right subtree. **Return** the result if and only if the accumulated sum equals to target sum or accumulate result with logical OR operator also only when traversal reach to leaf node.

4.9.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.9.3.3 Algorithm\

dfs 1.1.1

4.9.4 Java Code

```
public boolean hasPathSum(TreeNode root, int sum) {
    return hasPathSum(root, 0, sum);
}

public boolean hasPathSum(TreeNode root, int current, int target) {
    if (root == null) {
        return false;
    }

    current += root.val;

    if (root.left == null && root.right == null) {
        return current == target;
    } else {
        return hasPathSum(root.left, current, target) || hasPathSum(root.right, current
    }
}
```

4.10 Path Sum II / Leet Code 113 / Medium

4.10.1 Description

Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum.

4.10.2 Example

Given the below binary tree and sum = 22, return:

```
[
    [5,4,11,2],
```

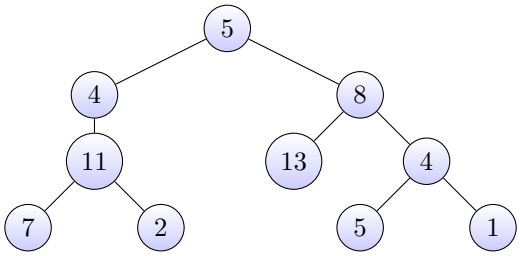


Figure 4.14: Some caption.

```
    [5,8,4,5]
]
```

4.10.3 Solution

4.10.3.1 Walkthrough\

Additional list is needed to keep track current path: adding current node on entering and removing (current) node on exiting. Remember and **return** the list if and only if current sum equals to target sum for leaf nodes. Need to remove the node list to back track also to help us locate more path in the tree.

4.10.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.10.3.3 Algorithm\

backtrack 1.1.6, dfs 1.1.1

4.10.4 Java Code

```
public List<List<Integer>> pathSum(TreeNode root, int sum) {
    List<List<Integer>> result = new LinkedList<>();
    List<Integer> trace = new LinkedList<>();
    backtrack(root, 0, sum, result, trace);
    return result;
}
private void backtrack(TreeNode root, int current, int target, List<List<Integer>> result, List<I
    if (root == null) {
```

```
            return;
        }
        trace.add(root.val);
        current += root.val;

        //leaf node
        if (root.left == null && root.right == null && current == target) {
            //to copy the valid list
            result.add(new LinkedList<>(trace));
            return;
        }

        //otherwise - intermediate nodes
        if (root.left != null) {
            pathSum(root.left, current, target, result, trace);

            //clean up last inserted left node to back track one step
            trace.remove(trace.size() - 1);
        }
        if (root.right != null) {
            pathSum(root.right, current, target, result, trace);

            //clean up last inserted right node to back track one step
            trace.remove(trace.size() - 1);
        }
    }
```

4.11 Path Sum III / Leet Code 437 / Easy

4.11.1 Description

You are given a binary tree in which each node contains an integer value.

Find the number of paths that sum to a given value.

The path does not need to start or end at the root or a leaf, but it must go downwards (traveling only from parent nodes to child nodes).

The tree has no more than 1,000 nodes and the values are in the range -1,000,000 to 1,000,000. ### Example

sum = 8, Return 3. The paths that sum to 8 are:

```
1. 5 -> 3
2. 5 -> 2 -> 1
3. -3 -> 11
```

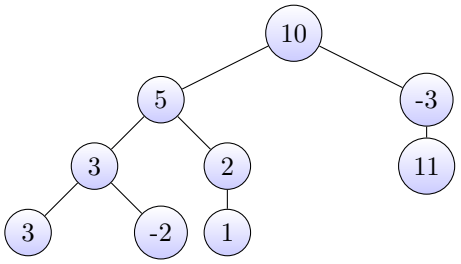


Figure 4.15: Some caption.

4.11.2 Solution

4.11.2.1 Walkthrough\

Continue the recursive on left and right subtrees and **do not return** when accumulated sum equals to the target sum. Use a length-1 array to keep track the count of path number in the recursive. We should consider both cases to include root and exlude root. For instance, if there is a tree [1,-2,1,-1] and taget sum is -1. There are four paths combined :

```
1. [1,-2]
2. [-2,1]
3. [-1]
4. [1,-2,1,-1]
```

In addition, we need to store accumulated count across recursive function calls. Thus, data type cannot be Immutable, since each arithmetic operation would create another object. Thus, we need to have a mutable variable that exists across each stack frame:

- Have a int[1] to store the accumulated count.
- Have a wrapper object to reset the value towards computation.
- Have a shared variable declared for this purpose.

4.11.2.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.11.2.3 Algorithm\

dfs 1.1.1

4.11.3 Java Code

```
int count = 0;
public int pathSum(TreeNode root, int sum) {
    helperSum(root, 0, sum);
    return count;
}
private void helperSum(TreeNode root, int current, int target) {
    if (root == null) {
        return;
    }
    // root included
    helper(root, current, target);
    // root excluded
    helperSum(root.left, current, target);
    helperSum(root.right, current, target);
}
private void helper(TreeNode root, int current, int target) {
    if (root == null) {
        return;
    }
    current += root.val;
    if (current == target) {
        count++;
        // do not return.
    }
    helper(root.left, current, target);
    helper(root.right, current, target);
}
```

4.12 Binary Tree Preorder Traversal / Leet Code 144 / Medium

4.12.1 Description

Given a binary tree, return the preorder traversal of its nodes' values.

4.12.2 Example

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

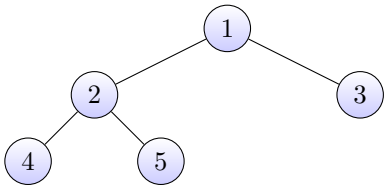


Figure 4.16: Some caption.

4.12.3 Solution - Iterative I

4.12.3.1 Walkthrough\

For iterative solution, use a stack to store the root node that will be used to retrieve root.right for later use. For initial loop condiition, be sure to include (root != null) to make sure loop starts since stack initially is empty.

4.12.3.2 Analysis\

Time complexity is O(n) as every node is visited once. #### Algorithm\

4.12.4 Java Code - Iterative I

```
public List<Integer> preorderTraversal(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    //stack is used to store the root node that will be used access root.right later
    Stack<TreeNode> stack = new Stack<>();

    while(!stack.isEmpty() || root != null) {
        if(root != null) {
            stack.push(root);
            result.add(root.val); // Add before going to children
            root = root.left;
        } else {
            root = stack.pop().right;
        }
    }
    return result;
}
```

4.12.5 Solution - Iterative II

4.12.5.1 Walkthrough\

To convert an inherently recursive procedures to iterative, we need an explicit stack, and do following while the stack is not empty:

- Pop an item from stack and process it.
- Push **right child** of popped item to stack
- Push **left child** of popped item to stack

Since stack is a LIFO, when the right child node is pushed before the left child node to a stack, the left child node would be processed before the right child node.

4.12.5.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.12.5.3 Algorithm\

4.12.6 Java Code - Iterative II

```
public List<Integer> preorderTraversal(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    //stack is used to store the root node that will be used access root.right later
    Stack<TreeNode> stack = new Stack<>();

    if (root == null) {
        return result;
    }
    stack.push(root);

    while (!stack.empty()) {
        Node node = nodeStack.pop();
        result.add(node.val);

        // Push the RIGHT child of the popped node to stack
        if (node.right != null) {
            stack.push(node.right);
        }
        // Push the LEFT child of the popped node to stack
        if (node.left != null) {
            stack.push(node.left);
```

```
        }
    }

    return result;
}
```

4.12.7 Solution - Recursive

4.12.7.1 Walkthrough\

For recursive implementation, do the following step

- To process data at current node
- Recursively invoke left child node.
- Recursively invoke right child node.

4.12.7.2 Analysis\

Time complexity is O(n) as every node is visited once. Auxiliary Space is O(1) if we do not consider the size of stacks for function calls, otherwise O(n).

4.12.7.3 Algorithm\

dfs 1.1.1

4.12.8 Java Code - Recursive

```
public List<Integer> preorderTraversal(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    traversalHelper(root, result);

    return result;
}

public void traversalHelper(TreeNode root, List<Integer> list) {
    if(root == null) {
        return;
    }

    list.add(root.val);
    traversalHelper(root.left, list);
```

```
    traversalHelper(root.right, list);
}
```

4.13 Binary Tree Inorder Traversal / Leet Code 94 / Medium

4.13.1 Description

Given a binary tree, return the inorder traversal of its nodes' values.

4.13.2 Example

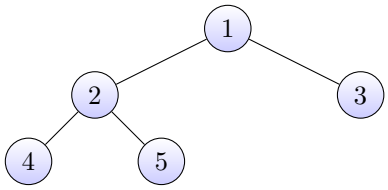


Figure 4.17: Some caption.

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

4.13.3 Solution - Iterative I

4.13.3.1 Walkthrough\

For iterative solution, use a stack to store the root node that will be used to retrieve root.right for later use. For initial loop condiition, be sure to include (root != null) to make sure loop starts since stack initially is empty.

4.13.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.13.3.3 Algorithm\

4.13.4 Java Code - Iterative I

```
public List<Integer> inorderTraversal(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    //stack is used to store the root node that will be used access root.right later
    Stack<TreeNode> stack = new Stack<>();

    while(!stack.isEmpty() || root != null) {
        if(root != null) {
            stack.push(root);
            root = root.left;
        } else {
            root = stack.pop();
            result.add(root.val);
            root = root.right;
        }
    }
    return result;
}
```

4.13.5 Solution - Inorder Recursive

4.13.5.1 Walkthrough\

For recursive implementation, do the following step

- Recursively invoke left child node.
- To process data at current node
- Recursively invoke right child node.

4.13.5.2 Analysis\

Time complexity is O(n) as every node is visited once. Auxiliary Space is O(1) if we do not consider the size of stacks for function calls, otherwise O(n).

4.13.5.3 Algorithm\

dfs 1.1.1

4.13.6 Java Code - Inorder Recursive

```
public List<Integer> inorderTraversal(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    traversalHelper(root, result);

    return result;
}

public void traversalHelper(TreeNode root, List<Integer> list) {
    if(root == null) {
        return;
    }

    traversalHelper(root.left, list);
    list.add(root.val);
    traversalHelper(root.right, list);
}
```

4.14 Binary Tree Postorder Traversal / Leet Code 145 / Hard

4.14.1 Description

Given a binary tree, return the postorder traversal of its nodes' values.

4.14.2 Example

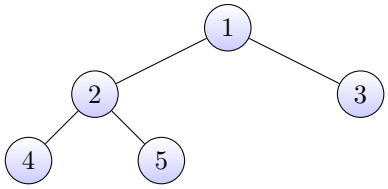


Figure 4.18: Some caption.

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

4.14.3 Solution - Iterative I

4.14.3.1 Walkthrough\

For iterative solution, use a stack to store the root node that will be used to retrieve root.right for later use. For initial loop condiition, be sure to include (root != null) to make sure loop starts since stack initially is empty.

4.14.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.14.3.3 Algorithm\

4.14.4 Java Code - Iterative I

```
public List<Integer> postorderTraversal(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    //stack is used to store the root node that will be used access root.right later
    Stack<TreeNode> stack = new Stack<>();

    while(!stack.isEmpty() || root != null) {
        if(root != null) {
            stack.push(root);

            //insert to head
            result.add(0, root.val);
            root = root.right;
        } else {
            root = stack.pop().left;
        }
    }
    return result;
}
```

4.14.5 Solution - Iterative with Two Stacks

4.14.5.1 Walkthrough\

The idea is to push reverse postorder traversal to a stack. Once we have the reversed postorder traversal in a stack, we can just pop all items one by one from the stack and process them; this order will be in postorder because of

the LIFO property of stacks. To get a reversed postorder elements in a stack - the second stack is used for this purpose, this sequence is very similar to the preorder traversal. The only difference is that the right child is visited before left child, and therefore the sequence is "root right left" instead of "root left right".

4.14.5.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.14.5.3 Algorithm\

4.14.6 Java Code - Iterative with Two Stacks

```
public List<Integer> postorderTraversal(TreeNode root) {
    List<Integer> result = new ArrayList<>();

    Stack<TreeNode> stack1 = new Stack<>();
    //stack2 is used to store the reversed postorder elements in a stack
    Stack<TreeNode> stack2 = new Stack<>();

    if (root == null) {
        return result;
    }
    stack.push(root);

    while (!stack1.isEmpty()) {
        // Pop an item from stack1 and push it to stack2
        TreeNode node = stack1.pop();
        stack2.push(node);

        // Push LEFT child of popped item to stack1
        if (node.left != null) {
            stack1.push(node.left);
        }
        // Push RIGHT child of popped item to stack1
        if (node.right != null) {
            stack1.push(node.right);
        }
    }

    //Traverse all reversed elements in second stack
    while (!stack2.isEmpty()) {
```

```
        TreeNode node = stack2.pop();
        result.add(node.val)
    }

    return result;
    }
```

4.14.7 Solution - Recursive

4.14.7.1 Walkthrough\

For recursive implementation, do the following step

- Recursively invoke left child node.
- Recursively invoke right child node.
- To process data at current node

4.14.7.2 Analysis\

Time complexity is O(n) as every node is visited once. Auxiliary Space is O(1) if we do not consider the size of stacks for function calls, otherwise O(n).

4.14.7.3 Algorithm\

dfs 1.1.1

4.14.8 Java Code - Recursive

```
public List<Integer> postorderTraversal(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    traversalHelper(root, result);

    return result;
}

public void traversalHelper(TreeNode root, List<Integer> list) {
    if(root == null) {
        return;
    }

    traversalHelper(root.left, list);
    traversalHelper(root.right, list);
```

```
    list.add(root.val);
}
```

4.15 Maximum Depth of Binary Tree / Leet Code 104 / Easy

4.15.1 Description

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

4.15.2 Example

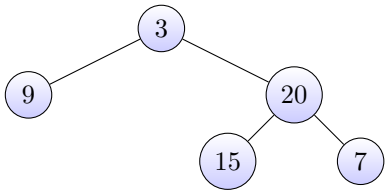


Figure 4.19: Some caption.

return its depth = 3.

4.15.3 Solution

4.15.3.1 Walkthrough\

Recursively for each level:

- Take the larger depth of left and right subtree.
- add one for current level

4.15.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.15.3.3 Algorithm\

recursive **??**

4.15.4 Java Code

```
public int maxDepth(TreeNode root) {
    if(root == null) {
        return 0;
    }

    int lHeight = 0, rHeight = 0;
    if(root.left != null) {
        lHeight = maxDepth(root.left);
    }

    if(root.right != null) {
        rHeight = maxDepth(root.right);
    }

    return Math.max(lHeight, rHeight) + 1;
}
```

4.16 Minimum Depth of Binary Tree / Leet Code 104 / Easy

4.16.1 Description

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

4.16.2 Example

return its minimum depth = 2.

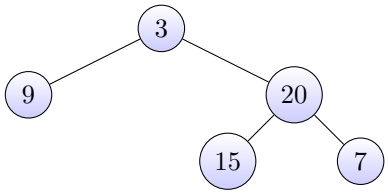


Figure 4.20: Some caption.

4.16.3 Solution - Recursive

4.16.3.1 Walkthrough\

If a node has a missing child (depth is **NOT** 0), then we need to drill down to the other path to the leaf node recursively. If both nodes exist, take the minimum value out of the left and right subtree.

4.16.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.16.3.3 Algorithm\

dfs 1.1.1

4.16.4 Java Code - Recursive

```
public int minDepth(TreeNode root) {
    if(root == null) {
        return 0;
    }

    if(root.left == null) {
        //drill down to leaf node of the other path
        return minDepth(root.right) + 1;
    } else if(root.right == null) {
        //drill down to leaf node of the other path
        return minDepth(root.left) + 1;
    } else {
        return Math.min(minDepth(root.left), minDepth(root.right)) + 1;
    }
}
```

4.16.5 Solution - Iterative BFS

4.16.5.1 Walkthrough\

Iterative: With BFS strategy, use two queues. One for current level and the other for next level.

- While traversing the currentLevel queue, add left and right children to the nextLevel queue.
- If currentLevel is empty, switch to nextLevel by currentLevel=nextLevel and nextLevel = new Queue(); Also, increment depth by 1

4.16.5.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.16.5.3 Algorithm\

bfs 1.1.2

4.16.6 Java Code - Iterative BFS

```
public int minDepth(TreeNode root) {
    if(root == null) {
        return 0;
    }

    Queue<TreeNode> currentLevel = new LinkedList<>();
    Queue<TreeNode> nextLevel = new LinkedList<>();

    currentLevel.offer(root);
    int depth = 1;

    while(!currentLevel.isEmpty()) {
        TreeNode node = currentLevel.poll();

        if(node.left == null && node.right == null) {
            return depth;
        }

        if(node.left != null) {
            nextLevel.offer(node.left);
        }
```

```
        if(node.right != null) {
            nextLevel.offer(node.right);
        }

        //go to children level by swapping queues, increase depth
        if(currentLevel.isEmpty()) {
            currentLevel = nextLevel;
            nextLevel = new LinkedList<>();
            depth++;
        }
    }

    return depth;
}
```

4.17 Count Univalue Subtrees / Leet Code 250 / Medium

4.17.1 Description

Given a binary tree, count the number of uni-value subtrees. A Uni-value subtree means all nodes of the subtree have the same value.

4.17.2 Example

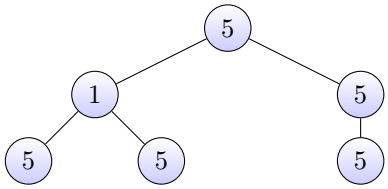


Figure 4.21: Some caption.

return 4.

4.17.3 Solution

4.17.3.1 Walkthrough\

There are **TWO** scenarios that returns true

- Recursively validate left node.val and right node.val == root.val
- A leaf node

In addition, we need to store accumulated count across recursive function calls. Thus, data type cannot be Immutable, since each arithmetic operation would create another object. Thus, we need to have a mutable variable that exists across each stack frame:

- Have a int[1] to store the accumulated count.
- Have a wrapper object to reset the value towards computation.
- Have a shared variable declared for this purpose.

4.17.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.17.3.3 Algorithm\

dfs 1.1.1

4.17.4 Java Code

```
int count = 0;
public int countUnivalSubtrees(TreeNode root) {
    helper(root, count);
    return count;
}
boolean helper(TreeNode root) {
    if (root == null) {
        return true;
    }
    boolean left = helper(root.left);
    boolean right = helper(root.right);
    if (left && right) {
        //validate left subtree
        if (root.left != null && root.left.val != root.val) {
            return false;
        }
        // validate right subtree
        if (root.right != null && root.right.val != root.val) {
            return false;
        }
        count++;
        return true;
```

```
    } else {
        return false;
    }
}
```

4.18 Validate Binary Search Tree / Leet Code 98 / Medium

4.18.1 Description

Given a binary tree, determine if it is a valid binary search tree (BST). Assume a 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.

4.18.2 Example

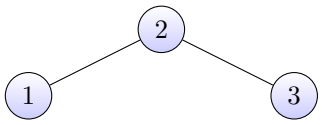


Figure 4.22: Some caption.

Output: true

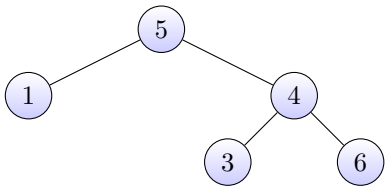


Figure 4.23: Some caption.

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

4.18.3 Solution - Recursive

4.18.3.1 Walkthrough\

For recursive, define a numeric lower and upper bounds for each validation - initially, (long) (Integer.MIN_VALUE - 1) and (long) (Integer.MAX_VALUE + 1). Recursively validate the node boundary by replacing with left and right nodes.

4.18.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.18.3.3 Algorithm\

dfs 1.1.1

4.18.4 Java Code - Recursive

```
public boolean isValidBST(TreeNode root) {
    return isValidBST(root, (long) Integer.MIN_VALUE - 1, (long) Integer.MAX_VALUE + 1);
}

public boolean isValidBST(TreeNode root, long lowerBound, long upperBound) {
    if(root == null) {
        return true;
    } else {
        boolean result = root.val > lowerBound && root.val < upperBound;
        return result && isValidBST(root.left, lowerBound, root.val) &&
            isValidBST(root.right, root.val, upperBound);
    }
}
```

4.18.5 Solution - Iterative BFS

4.18.5.1 Walkthrough\

For iterative, we have a wrapper to hold upper and lower bounds for each nodes. Use BFS to iteratively traverse the tree and add left or right nodes with proper boundaries accordingly.

4.18.5.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.18.5.3 Algorithm\

bfs 1.1.2

4.18.6 Java Code - Iterative BFS

```
//Decorator pattern
private static class BoundedTreeNode extends TreeNode {
    int upper;

    int lower;

    TreeNode proxy;

    BoundedTreeNode(TreeNode node, int max, int min) {
        this.proxy = node;
        this.upper = max;
        this.lower = min;
    }
}


public static boolean validateBSTItr(TreeNode root) {
    if(root == null) {
        return true;
    }

    Queue<BoundedTreeNode> queue = new LinkedList<>();

    queue.offer(new BoundedTreeNode(root, Integer.MAX_VALUE, Integer.MIN_VALUE));

    while(!queue.isEmpty()) {
        BoundedTreeNode node = queue.poll();


        if(node.proxy.data > node.upper || node.proxy.data < node.lower) {
            // out of boundary
            return false;
        }
```

```
        if(node.proxy.left != null) {
            queue.offer(new BoundedTreeNode(node.proxy.left, node.proxy.data, node.lower));
        }

        if(node.proxy.right != null) {
            queue.offer(new BoundedTreeNode(node.proxy.right, node.upper, node.proxy.data));
        }
    }

    return true;
}
```

4.19 Binary Tree Upside Down / Leet Code 156 / Medium

4.19.1 Description

Given a binary tree where all the right nodes are either leaf nodes with a sibling (a left node that shares the same parent node) or empty, flip it upside down and turn it into a tree where the original right nodes turned into left leaf nodes. Return the new root.

4.19.2 Example

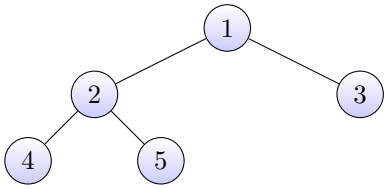


Figure 4.24: Some caption.

return the root of the binary tree

4.19.3 Solution

4.19.3.1 Walkthrough\

The newRoot of a flipped tree will be root.left that is

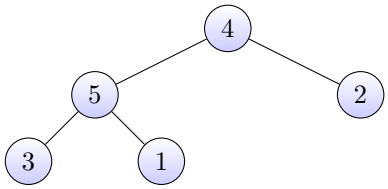


Figure 4.25: Some caption.

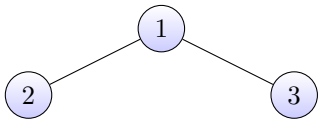


Figure 4.26: Some caption.

flips to

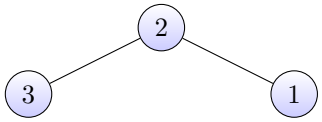


Figure 4.27: Some caption.

Additionally, we need to properly place the children of root.left.

4.19.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.19.3.3 Algorithm\

dfs 1.1.1

4.19.4 Java Code

```
public TreeNode upsideDownBinaryTree(TreeNode root) {
    if(root == null || root.left == null) {
        return root;
    }

    TreeNode newRoot = upsideDownBinaryTree(root.left);
```

```
    //children of root.left
    root.left.left = root.right;
    root.left.right = root;

    root.left = null;
    root.right = null;
    return newRoot;
}
```

4.20 Inorder Successor in BST / Leet Code 285 / Medium

4.20.1 Description

Given a binary search tree and a node in it, find the in-order successor of that node in the BST. Note: If the given node has no in-order successor in the tree, return null.

4.20.2 Example

4.20.3 Solution

4.20.3.1 Walkthrough\

4.20.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.20.3.3 Algorithm\

bfs 1.1.2

4.20.4 Java Code

```
public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {
    Stack<TreeNode> stack = new Stack<TreeNode>();
    TreeNode node = null, prev = null;
    while (!stack.isEmpty() || node != null) {
        if (node != null) {
```

```
            stack.push(node);
            node = node.left;
        } else {
            node = stack.pop();
            if (prev == p) {
                return node;
            }
            prev = node;
            node = node.right;
        }
    }
    return null;
}
```

4.21 Binary Tree Longest Consecutive Sequence / Leet Code 298 / Medium

4.21.1 Description

Given a binary tree, find the length of the longest consecutive sequence path. The path refers to any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The longest consecutive path need to be from parent to child (cannot be the reverse).

4.21.2 Example

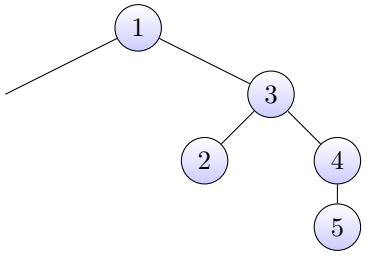


Figure 4.28: Some caption.

Longest consecutive sequence path is 3-4-5, so return 3.

Longest consecutive sequence path is 2-3,not3-2-1, so return 2.

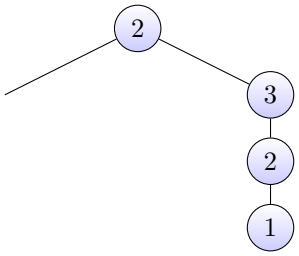


Figure 4.29: Some caption.

4.21.3 Solution

4.21.3.1 Walkthrough\

For each recursive call,

- if root.val == target, length++
- otherwise, reset length = 1

Retrieve for the maximum length and recursively call on left and right subtrees with increasing sequence (root.val + 1).

If we want longest increaing sequence, change to (root.val >= target) to verify increasing sequence.

4.21.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.21.3.3 Algorithm\

dfs 1.1.1

4.21.4 Java Code

```
private int longest;

public int longestConsecutive(TreeNode root) {
    if (root == null) {
        return 0;
    }

    longestConsecutive(root, 0, root.val);
```

```
    return longest;
}
public void longestConsecutive(TreeNode root, int length, int target) {
    if (root == null) {
        return;
    } else if (root.val == target) {
        length++;
    } else {
        length = 1;
    }

    longest = Math.max(longest, length);

    //root.val + 1 for increasing sequencef
    longestConsecutive(root.left, length, root.val + 1);
    longestConsecutive(root.right, length, root.val + 1);
}
```

4.22 Find Leaves of Binary Tree / Leet Code 366 / Medium

4.22.1 Description

Given a binary tree, collect a tree's nodes as if you were doing this: Collect and remove all leaves, repeat until the tree is empty.

4.22.2 Example

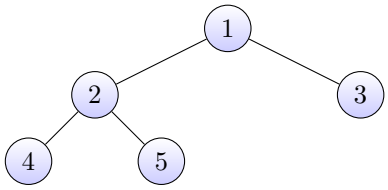


Figure 4.30: Some caption.

Returns [4, 5, 3], [2], [1].

4.22.3 Solution

4.22.3.1 Walkthrough\

For each node, compute maximum level (leaf node is -1) out of recursive call to left and right subtree + 1. If level increases, add a new list and add the node.val into the list – list.add(node.val)

4.22.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.22.3.3 Algorithm\

dfs 1.1.1

4.22.4 Java Code

```
public List<List<Integer>> findLeaves(TreeNode root) {
    List<List<Integer>> result = new LinkedList<List<Integer>>();
    helper(root, result);

    return result;
}
private int helper(TreeNode root, List<List<Integer>> result) {
    if (root == null) {
        return -1;
    }

    int level = Math.max(helper(root.left, result), helper(root.right, result)) + 1;

    //add a new level if level increases
    if (result.size() <= level) {
        result.add(new LinkedList<Integer>());
    }

    //add root.val
    result.get(level).add(root.val);

    return level;
}
```

4.23 Diameter of Binary Tree / Leet Code 543 / Easy

4.23.1 Description

Given a binary tree, you need to compute the length of the diameter of the tree. The diameter of a binary tree is the length of the longest path between any two nodes in a tree. This path may or may not pass through the root.

4.23.2 Example

Given a binary tree

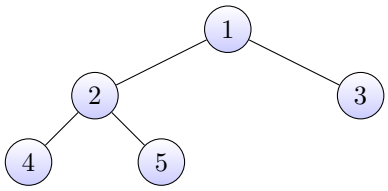


Figure 4.31: Some caption.

Return 3, which is the length of the path [4,2,1,3] or [5,2,1,3]. Note: The length of path between two nodes is represented by the number of edges between them.

4.23.3 Solution

4.23.3.1 Walkthrough\

Have a help function to compute the left and right height as well as diameter (left height + right height). Find the maximum diameter among root, left and right.

4.23.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.23.3.3 Algorithm\

dfs 1.1.1

4.23.4 Java Code

```
public int diameterOfBinaryTree(TreeNode root) {
    int[] result = diameterAndHeight(root);
    return result[0];
}

public int[] diameterAndHeight(TreeNode root) {
    int heightDiameter[] = { 0, 0 };          // initialize the diameter and height

    if (root != null) {
        int[] leftResult = diameterAndHeight(root.left);
        int[] rightResult = diameterAndHeight(root.right);
        int height = Math.max(leftResult[1], rightResult[1]) + 1;
        int leftDiameter = leftResult[0];
        int rightDiameter = rightResult[0];
        int rootDiameter = leftResult[1] + rightResult[1];
        int finalDiameter = Math.max(rootDiameter, Math.max(leftDiameter, rightDiameter));
        heightDiameter[0] = finalDiameter;
        heightDiameter[1] = height;
    }
    return heightDiameter;
}
```

4.24 Binary Tree Serialization / Firecode / Level 3

4.24.1 Description

In Computer Science, serialization is the process of converting objects or data structures into a sequence (or series) of characters that can be stored easily in a file / database table or transmitted across a network. Serialized objects need to be de-serialized to create a semantically identical clone of the original object, before being used in programs. You're given the root node of a binary tree - TreeNode root in the method serializeTree. This method should serialize the binary tree and output a String str, which is then used as an input parameter for the method restoreTree. restoreTree should create a Binary Tree that is structurally identical to the one you serialized and return the root node of the tree. Your task is to fill in the logic for these 2 methods. Don't worry about passing the serialized String to restoreTree - that will be done automatically when you run your code. Feel free to use any notation you prefer when serializing the binary tree. The choice of traversal algorithm is also open - but try and limit the time complexity of both methods to O(n).

4.24.2 Example

4.24.3 Solution

4.24.3.1 Walkthrough\

Your serialized String will be used to restore the tree. Be sure to use the same format and notation in restoreTree that you use to serialize in serializeTree. For example, the following binary tree

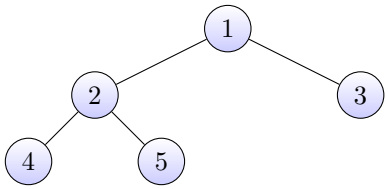


Figure 4.32: Some caption.

would be serialize into string "1 2 4 # # 5 # # 3 # #".

4.24.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.24.3.3 Algorithm\

dfs 1.1.1

4.24.4 Java Code

```
public String serializeTree(TreeNode root){
    StringBuilder sb = new StringBuilder();

    if (root == null) {
        sb.append("# ");
    } else {
        sb.append(root.data + " ");
        sb.append(serializeTree(root.left));
        sb.append(serializeTree(root.right));
    }

    return sb.toString();
```

```
}

public TreeNode restoreTree(String str){
    if (str == null || str.length() == 0) {
        return null;
    }

    StringTokenizer tokenizer = new StringTokenizer(str, " ");
    return deserialize(tokenizer);
}

private TreeNode deserialize(StringTokenizer tokenizer){
    if (!tokenizer.hasMoreTokens()) {
        return null;
    }
    String val = tokenizer.nextToken();

    if (val.equals("#")) {
        return null;
    }

    TreeNode root = new TreeNode(Integer.parseInt(val));
    root.left = deserialize(tokenizer);
    root.right = deserialize(tokenizer);

    return root;
}
```

4.25 Fill in the Ancestors of the Node in a Binary Tree / Firebase / Level 3

4.25.1 Description

Given a binary tree's root node, an empty ArrayList and an integer nodeData, write a method that finds a target node - N with data = nodeData and populates the ArrayList with the data of the ancestor nodes of N - added from the bottom - up

4.25.1.1 Algorithm\

dfs 1.1.1, backtrack 1.1.6

4.25.2 Example

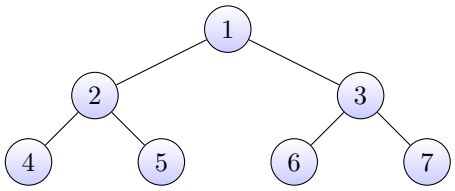


Figure 4.33: Some caption.

Node: 5 = [2, 1]

4.25.3 Solution

4.25.3.1 Walkthrough\

We use an arrayList to keep the current path of TreeNode visited, also TreeNode to be removed for backtracking purposes. We use recurisve call and terminate on leave nodes as well as target node is found. The recursive function returns a boolean value when the target node / value is found on the subtree path.

4.25.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.25.3.3 Algorithm\

dfs 1.1.1, backtrack 1.1.6

4.25.4 Java Code

```
public ArrayList<Integer> ancestorsList = new ArrayList<Integer>();

public boolean printAncestors(TreeNode root, int nodeData) {
    if(root == null) {
        return false;
    }

    if(root.data == nodeData) {
        return true;
    } else {
```

```
        ancestorsList.add(0, root.data);

        boolean isRightSubtree = printAncestors(root.left, nodeData) || printAncestors(root.right

        if(!isRightSubtree) {
            //remove the backtrack nodes
            ancestorsList.remove(0);
        }

        return isRightSubtree;
    }
}
```

4.26 Find the k^{th} Largest Node in a BST / Firecode / Level 3

4.26.1 Description

Given a Binary Search Tree and an integer k, implement a method to find and return its k^{th} largest node

4.26.2 Example

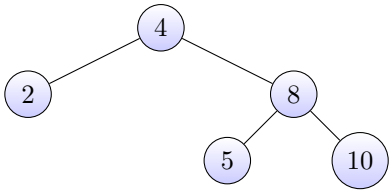


Figure 4.34: Some caption.

In the above scenario, if k = 1, then the output is 10 i.e. k = 1, represents the largest element of the tree, k = 2, represents the second largest element and so on.

4.26.3 Solution

4.26.3.1 Walkthrough\

First we compute the size of right subtree, and evaluate the following possibilities

- if k == (sizeOfRightSubtree + 1), return this node
- if k < sizeOfRightSubtree, the target must reside in the right subtree. Drill down to right subtree
- if k > sizeOfRightSubtree, the target must reside in the left subtree. Drill down to the left subtree and change target position to (k - sizeOfRightSubtree - 1)

4.26.3.2 Analysis\

Time complexity is O(log(n)) as a certain path of nodes will be visited since this is a BST.

4.26.3.3 Algorithm\

dfs 1.1.1, bst 1.1.11

4.26.4 Java Code

```
public TreeNode findKthLargest(TreeNode root, int k) {
    if(root == null) {
        return null;
    }

    int rightSubtreeSize = size(root.right);

    if( (rightSubtreeSize + 1) == k) {
        return root;
    } else if( rightSubtreeSize > k ) {
        //drill down to the right subtree
        return findKthLargest(root.right, k);
    } else {
        //drill down to the left subtree
        return findKthLargest(root.left, k - rightSubtreeSize - 1);
    }
}
```

4.27 Convert Sorted Array to Binary Search Tree / Leet Code 108 / Easy

4.27.1 Description

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of every node never differ by more than 1

4.27.2 Example

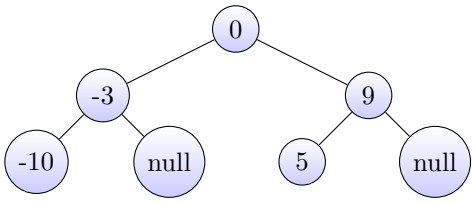


Figure 4.35: Some caption.

4.27.3 Solution

4.27.3.1 Walkthrough\

To create a tree of minimal height, we need to mathch the number of nodes in the left subtree to the number of the nodes in the right subtree as much as possible. This means that we want the root to be the middle of the **sorted** array, since this would mean that half the elements would be less than the root and half wood be greater than it.

4.27.3.2 Analysis\

The overal time complexity is O(n) as every node is visited once.

4.27.3.3 Algorithm\

dfs 1.1.1

4.27.4 Java Code

```
public TreeNode sortedArrayToBST(int[] nums) {
    return buildHeightBalancedBST(nums, 0, nums.length -1);
}

public TreeNode buildHeightBalancedBST(int[] nums, int start, int end) {
    if(start > end) {
        return null;
    }

    int mid = (start + end) / 2;
    TreeNode node = new TreeNode(nums[mid]);
    node.left = buildHeightBalancedBST(nums, start, mid - 1);
    node.right = buildHeightBalancedBST(nums, mid + 1, end);

    return node;
}
```

4.28 Populating Next Right Pointers in Each Node / Leet Code 116 / Medium

4.28.1 Description

You are given a perfect binary tree where all leaves are on the same level, and every parent has two children. Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL. Initially, all next pointers are set to NULL.

4.28.2 Example

4.28.3 Solution

4.28.3.1 Walkthrough\

4.28.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.28.3.3 Algorithm\

dfs 1.1.1

4.28.4 Java Code - PreOrder Recursive

```
public Node connect(Node root) {
    if(root == null){
        return root;
    }

    if(root.left == null) {
        return root;
    }

    // populate left child.next => right child
    root.left.next=root.right;

    // previously populated, i.e. root(2)
    if(root.next != null){
        //root.next = 3 which connects to another branch
        root.right.next=root.next.left;
    }

    connect(root.left);
    connect(root.right);

    return root;
}
```

4.29 Construct Binary Tree from Preorder and Inorder Traversal / Leet Code 105 / Medium

4.29.1 Description

Given preorder and inorder traversal of a tree, construct the binary tree. You may assume that duplicates do not exist in the tree.

4.29.2 Example

preorder = [3,9,20,15,7] inorder = [9,3,15,20,7]

Return the following binary tree:

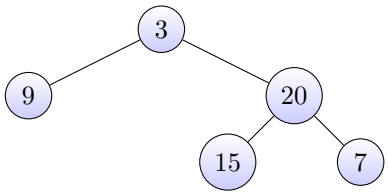


Figure 4.36: Some caption.

4.29.3 Solution

4.29.3.1 Walkthrough\

PreOrder array is used to construct data value of a node, whereas InOrder array is used to acknowledge the left and right boundary of a node. We could use a Map to store the InOrder array associated with its index (position in array) value and biuld tree according to the following conditions:

- if startIdx > endIdx Return null. (Terminal)
- if startIdx == endIdx The node is here. (Terminal)
- if startIdx > endIdx There should be more nodes in this branch.

4.29.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

4.29.3.3 Algorithm\

dfs 1.1.1

4.29.4 Java Code - PreOrder Recursive

```
int preorderIndex = 0;

public TreeNode buildTree(int[] preorder, int[] inorder) {
    if (preorder.length == 0) {
```

```
        return null;
    }

    Map<Integer, Integer> inorderIndex = new HashMap<>();
    for (int i = 0; i < inorder.length; i++) {
        inorderIndex.put(inorder[i], i);
    }

    return buildTree(preorder, inorderIndex, 0, inorder.length - 1);
}

private TreeNode buildTree(int[] preorder, Map<Integer, Integer> inorderIndex, int startIdx, int
    if (startIdx > endIdx) {
        return null;
    }

    int val = preorder[preorderIndex++];
    TreeNode node = new TreeNode(val);

    if (startIdx == endIdx) {
        return node;
    }

    int inIndex = inorderIndex.get(val);
    node.left = buildTree(preorder, inorderIndex, startIdx, inIndex - 1);
    node.right = buildTree(preorder, inorderIndex, inIndex + 1, endIdx);

    return node;
}
```

Chapter 5

Graph

Imagine a graph with as the following:

```
1 -- 0 -- 4
|
5 -- 6
|  / | \
| /  |   \
2 -- 3 -- 7
```

5.0.0.1 Typical Implementation - Adjacency List

An adjacency list is created with an array of lists. Size of the array is equal to the number of vertices. Each entry represents the list of vertices adjacent to the i^{th} vertex. Each node maintains a list of all its adjacent edges, then, for each node, you could discover all its neighbors by traversing its adjacency list just once in linear time.

```
| 0 | -- 1 -- 4
| 1 | -- 0 -- 5
| 2 | -- 3 -- 5 -- 6
| 3 | -- 2 -- 6 -- 7
| 4 | -- 0
| 5 | -- 1 -- 2 -- 6
| 6 | -- 2 -- 3 -- 5 -- 7
| 7 | -- 3 -- 6
```

5.0.0.2 Typical Implementation - Adjacency Matrix

The adjacency information can also transform into matrix form with the convention followed here (for undirected graphs) is that each edge adds 1 to the appropriate cell in the matrix, and each loop adds 2. For each node, you have to traverse an entire row of length V in the matrix to discover all its outgoing edges.

```
0, 1, 0, 0, 1, 0, 0, 0
1, 0, 0, 0, 0, 1, 0, 0
0, 0, 0, 1, 0, 1, 1, 0
0, 0, 1, 0, 0, 0, 1, 1
1, 0, 0, 0, 0, 0, 0, 0
0, 1, 1, 0, 0, 0, 1, 0
0, 0, 1, 1, 0, 1, 0, 1
0, 0, 0, 1, 0, 0, 1, 0
```

In addition, a "visited" array that we'll use to keep track of which vertices have been visited.

5.0.0.3 Typical Implementation - Graph based

```
class Graph  {
    private int numOfVertices;   //No. of vertices
    private List<Integer> adj[]; //Adjacency List
    private boolean[] visited;   //visited mark for each vertex

    // Constructor
    public Graph(int num) {
        numOfVertices = num;
        adj = new LinkedList[v];
        for (int i=0; i<numOfVertices; ++i) {
            adj[i] = new LinkedList();
        }
    }
    ...
}
```

There is an alternative implementation to represent per node with a list of adjacent nodes.

5.0.0.4 Typical Implementation - per Node based

```
class Node {
    public int val;
    public List<Node> neighbors;
    private boolean visited;     //visited mark for the vertex

    public Node(int _val, List<Node> _neighbors) {
        val = _val;
        neighbors = _neighbors;
    }
    ...
}
```

5.0.1 Depth First Traversal

A depth first traversal without labeling the visited node usually ends up in exponential time complexity. There will be many subproblems being reevaluated (which were visited in past). Thus, the time complexity is $O(2^{h+1}-1)$ (as a full binary tree)

If your graph is implemented as an adjacency matrix, then, for each node, you have to traverse an entire row of length V in the matrix to discover all its outgoing edges. So, the complexity of DFS is $O(|V| * |V|) = O(|V|^2)$.

If your graph is implemented using adjacency lists, wherein each node maintains a list of all its adjacent edges, then, for each node, you could discover all its neighbors by traversing its adjacency list just once in linear time. For a directed graph, the sum of the sizes of the adjacency lists of all the nodes is E (total number of edges). So, the complexity of DFS is O(V) + O(E) = O(V + E). For an undirected graph, each edge will appear twice. Once in the adjacency list of either end of the edge. So, the overall complexity will be $O(|V|) + O(2 \cdot |E|) = O(|V + E|)$.

5.0.1.1 Algorithm\

dfs 1.1.1

5.0.1.2 Typical Implementation - Java

```
void helper(int node, boolean visited[]) {
    // Mark the current node as visited and print it
    visited[node] = true;

    // Recur for all the vertices adjacent to this vertex
```

```
    for(Integer adjNode: adj[node]) {
        if (!visited[adjNode]) {
            helper(adjNode, visited);
        }
    }
}

// The function to do DFS traversal. It uses recursive DFSUtil()
void dfs(int root) {
    // Mark all the vertices as not visited(set as
    // false by default in java)
    boolean visited[] = new boolean[numOfVertices];

    helper(root, visited);
}
```

5.0.2 Bread First Traversal

For every single vertex in the graph, we will end up looking at its neighboring nodes only once (directed graph) or twice (undirected graph). The time complexity for both a directed and undirected graph is the sum of the vertices and their edges as represented by the graph in its adjacency list representation, or $O(|V| + |E|)$

The power of using breadth-first search to traverse through a graph is that it can easily tell us the shortest way to get from one node to another.

5.0.2.1 Algorithm\

bfs 1.1.2

5.0.2.2 Typical Implementation - Java

```
class Graph  {
    private int numOfVertices;   // No. of vertices
    private LinkedList<Integer> adj[]; //Adjacency Lists

    ...

    void traversal(Integer root)  {
        // Mark all the vertices as not visited(By default
        // set as false)
        boolean visited[] = new boolean[numOfVertices];
```

```
        // Create a queue for BFS
        Queue<Integer> queue = new LinkedList<Integer>();

        // Mark the current node as visited and enqueue it
        visited[root] = true;
        queue.offer(root);

        while (queue.size() != 0) {
            // Dequeue a vertex from queue and print it
            Integer node = queue.poll();

            // Get all adjacent vertices of the dequeued vertex s
            // If a adjacent has not been visited, then mark it
            // visited and enqueue it
            for(Integer adjNode : adj[node]) {
                if (!visited[adjNode] && !queue.contain(adjNode)) {
                    queue.offer(adjNode);
                }
            }
        }
    }
}
```

5.0.3 Topological Sorting

A traversal algorithm for **Directed Acyclic Graph (DAG)** is a linear ordering of vertices such that for every directed edge $[u, v]$, vertex u comes before v in the ordering.

5.0.3.1 By DFS

In DFS implementation of Topological Sort we focused on sink vertices, i.e, vertices with zero out-going edges, and then at last had to reverse the order in which we got the **sink vertices** (which we did by using a stack, which is a Last In First Out data structure). A DAG has to have at least one sink vertex which is the vertex which has no outbound edges. In DFS we print the nodes as we see them, which means when we print a node, it has just been discovered but not yet processed, which means it is in the Visiting state. So DFS gives the order in which the nodes enter the Visiting state and not the Visited state. For topological sorting we need to have the order in which the nodes are completely processed, i.e, the order in which the nodes are marked as Visited. Because when a node is marked Visited then all of its child node have already been processed,

so they would be towards the right of the child nodes in the topological sort, as it should be.

For example, in the given graph, the vertex '5' should be printed before vertex '0', but unlike DFS, the vertex '4' should also be printed before vertex '0'.

```
5 -> 0 <- 4
|         |
v         v
2 -> 3 -> 1
```

In topological sorting, we use a temporary stack. We don't print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of stack. The final sequence is [5, 4, 2, 3, 1, 0]

5.0.3.2 Algorithm\

```
topological \\ref{topological}, dfs \\ref{dfs}
```

5.0.3.3 Typical Implementation - Java Code

```
void helper(Integer v, Stack stack) {
    visited[v] = true;

    // Recur for all the vertices adjacent to this
    // vertex
    Iterator<Integer> it = adj[v].iterator();
    while (it.hasNext()) {
        Integer adjNode = it.next();

        if (!visited[adjNode])  {
            helper(adjNode, visited, stack);
        }
    }

    // Push current vertex to stack which stores result
    stack.push(new Integer(v));
}

List<Integer> topologicalSort(Graph graph) {
    Stack<Integer> stack = new Stack<>();
    List<Integer> result = new ArrayList<>();

    // Call the recursive helper function to store
```

```
    // Topological Sort starting from all vertices
    // one by one
    for (int i = 0; i < numOfVertices; i++) {
        if (visited[i] == false)  {
            helper(i, visited, stack);
        }
    }

    //Now the stack contains the topological sorting of the graph
    for (Integer vertex : stack) {
        result.add(stack.pop());
    }

    return result
}
```

5.0.3.4 By BFS

In BFS implementation of the Topological sort we do the opposite: We look for for edges with no inbound edges (**source vertex**). And consequently in BFS implementation we don't have to reverse the order in which we get the vertices, since we get the vertices in order of the topological ordering. We use First-In-First-Out data structure Queue in this implementation. We just search for the vertices with zero indegrees and put them in the queue till we have processed all the vertices of the graph. Polling vertices from the queue one by one give the topological sort of the graph. Lastly check if the result set does not contain all the vertices that means there is at least one cycle in the graph. That is because the indegree of those vertices participating in the loop could not be made 0 by decrementing.

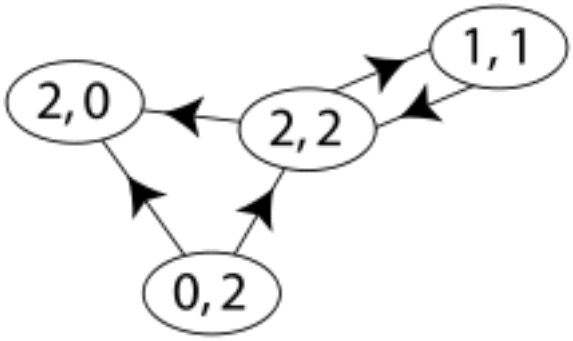


Figure 5.1: in-degree, out-degree

5.0.3.5 Algorithm\

topological 1.1.3, bfs 1.1.2

5.0.3.6 Typical Implementation - Java Code

```
public List topological_sort_bfs(int[][] graph) {
    List<Integer> result = new ArrayList<>();
    int[] indegree = new int[numOfVertices];

    // compute the indegree of all the vertices in the graph
    // For edge (0,1), indegree[1]++;
    for (int i = 0; i < numOfVertices; i++) {
        for(Integer vertex : adj[i]) {
            indegree[vertex]++;
        }
    }

    Queue<Integer> queue = new LinkedList<>();

    // initialize the queue with all the vertices with no inbound edges
    for (Integer vertex = 0; index < numOfVertices; index++) {
        if (indegree[vertex] == 0) {
            queue.offer(vertex);
        }
    }

    while (!queue.isEmpty()) {
        Integer vertex = queue.poll();
        result.add(vertex);

        // now disconnect vertex1 from the graph
        // and decrease the indegree of the other end of the edge by 1
        for (Integer adjVertex : adj[i]) {
            indegree[adjVertex]--;
            if (indegree[adjVertex] == 0) {
                queue.offer(adjVertex);
            }
        }
    }

    // check if the graph had a cycle / loop
    if (result.size() != numOfVertices) {
        return new ArrayList<Integer>();
```

```
    }
    return result;
}
```

5.1 Graph Serialization / /

5.1.1 Description

Given the a node to a Graph, represent that Graph as a String in such a way that it may be translated back to a Graph using the generated String A Node contains an integer value and all of the node's neighbors. ### Example N.A.

5.1.2 Solution

5.1.2.1 Walkthrough\

We use BFS to traverse the graph with a visited Set to avoid running into cycle, and add child node only on unvisited adjacent nodes. For serialization, we simply place current val as first element and adjacent nodes val at the same line for the following elements, i.e. current, adj1, adj2, adj3, ..etc. Therefore, there will be two occurrence for each node-pair (edges) in the serialized content. For deserialization, it is easier by first splitting the lines and splitting by the dilimter ',' and finally reassemble the graph data structure.

5.1.2.2 Analysis\

The time complexity for both a directed and undirected graph is the sum of the vertices and their edges as represented by the graph in its adjacency list representation, or $O(|V| + |E|)$

```
Graph:
1
Serialized Content:
1
```

```
Graph:
1 - 2
|   |
3 - 5
|
4
Serialized Content:
```

```
1, 2, 3
2, 1, 5
3, 1, 5, 4
4, 3
5, 2, 3
```

5.1.2.3 Algorithm\

bfs 1.1.2

5.1.3 Java Code

```
class Graph {
    private int numOfVertices; // No. of vertices
    private LinkedList<Integer> adj[]; //Adjacency Lists

    // Constructor
    Graph(int num) {
        numOfVertices = num;
        adj = new LinkedList[numOfVertices];
        for (int i = 0; i < numOfVertices; ++i) {
            adj[i] = new LinkedList();
        }
    }

    // Function to add an edge into the graph from v1 to v2
    public void addEdge(int v1, int v2) {
        adj[v1].add(v2);
    }

    public String serialize(Integer rootId) {
        StringBuilder stringBuilder = new StringBuilder();

        if(adj.length <= 0) {
            return "";
        }

        boolean visited[] = new boolean[numOfVertices];
        Queue<Integer> queue = new LinkedList<Integer>();

        // Mark the current node as visited and enqueue it
        queue.offer(rootId);
```

```
        while (queue.size() != 0) {
            Integer nodeId = queue.poll();
            visited[nodeId]=true;

            stringBuilder.append(serializeNode(nodeId));

            // Get all adjacent vertices of the dequeued vertex s
            // If a adjacent has not been visited, then mark it
            // visited and enqueue it
            for(Integer adjNodeId : adj[nodeId]) {
                if (!visited[adjNodeId] && !queue.contains(adjNodeId)) {
                    queue.offer(adjNodeId);
                }
            }
        }

        return stringBuilder.toString();
    }

    private String serializeNode(int nodeId) {
        StringBuilder sb = new StringBuilder();

        //serialize node and adjacency nodes
        sb.append(nodeId);
        for(Integer adjNodeId: adj[nodeId]) {
            sb.append(",");
            sb.append(adjNodeId);
        }
        sb.append("\r\n");

        return sb.toString();
    }

    public Graph deserialize(String input) {
        String[] lines = input.split("\r\n");
        Graph graph = new Graph(lines.length);

        for(String line: lines) {
            String[] nodes = line.split(",");

            if(nodes.length > 1) {
                int srcNodeId = Integer.parseInt(nodes[0]);

                //there are adjacent nodes
                for(int j = 1; j < nodes.length; j++) {
```

```
                    Integer dstNodeId = Integer.valueOf(nodes[j]);
                    graph.addEdge(srcNodeId, dstNodeId);
                }
            }
        }

        return graph;
    }
}
```

5.2 Distance of nearest cell having 1 in a binary matrix

5.2.1 Description

Given a binary matrix of N x M, containing at least a value 1. The task is to find the distance of nearest 1 in the matrix for each cell. The distance is calculated as $|i_1 - i_2| + |j_1 - j_2|$, where i_1, j_1 are the row number and column number of the current cell and i_2, j_2 are the row number and column number of the nearest cell having value 1.

5.2.2 Example

Input:

```
0, 0, 0, 1
0, 0, 1, 1
0, 1, 1, 0
```

Output:

```
3, 2, 1, 0
2, 1, 0, 0
1, 0, 0, 1
```

5.2.3 Solution

5.2.3.1 Walkthrough\

Consider each cell as a node and each boundary between any two adjacent cells be an edge. Number each cell from 1 to N*M. Now, push all the node whose

corresponding cell value is 1 in the matrix in the queue. Apply BFS using this queue to find the minimum distance of the adjacent node.

5.2.3.2 Analysis\

`* Create a graph with adjacency list for all nodes from 1 to rows * cols. Time complexity is O(rows * c`
`* Create an empty queue and 1 dimensional array of size (rows * cols) for visited[] and distance (init`

Time complexity is O(rows * cols) * For all nodes in the original matrix being 1, insert the associated node id into the queue and set the distance to 0. * Perform a BFS traversal of graph using above created queue. In BFS, we first explore immediate adjacent of all 1's, then adjacent of adjacent and determine the minimum distance. Time complexity is O(rows * cols) since all elements will be visited once.

Thus, the overall time complexity is O(rows * cols)

5.2.3.3 Algorithm\

bfs 1.1.2

5.2.4 Java Code

```
class Graph {
    private int numOfVertices; // No. of vertices
    private LinkedList<Integer> adj[]; //Adjacency Lists

    // Constructor
    Graph(int[][] matrix) {
        int rows = matrix.length;
        int cols = matrix[0].length;

        if(matrix.length == 0) {
            return;
        }
        numOfVertices = rows * cols;
        adj = new LinkedList[numOfVertices];

        for (int i = 0; i < numOfVertices; ++i) {
            adj[i] = new LinkedList();
        }
    }

    // Function to add an edge into the graph from v1 to v2
```

```
    public void addEdge(int src, int dst) {
        adj[src].add(dst);
    }

    public void fillAdjacency(int[][] matrix) {
        int rows = matrix.length;
        int cols = matrix[0].length;

        for(int i = 0; i < rows; i++) {
            for(int j = 0; j < cols; j++) {
                int srcNodeId = this.computeNodeId(i, j, cols);
                int dstNodeId = -1;

                //adding left or right neighbor nodes, if there are any (cols > 1)
                if(cols > 1) {
                    if (j == 0) {
                        dstNodeId = this.computeNodeId(i, j + 1, cols);
                        this.addEdge(srcNodeId, dstNodeId);
                    } else if (j == cols - 1) {
                        dstNodeId = this.computeNodeId(i, j - 1, cols);
                        this.addEdge(srcNodeId, dstNodeId);
                    } else {
                        dstNodeId = this.computeNodeId(i, j - 1, cols);
                        this.addEdge(srcNodeId, dstNodeId);
                        dstNodeId = this.computeNodeId(i, j + 1, cols);
                        this.addEdge(srcNodeId, dstNodeId);
                    }
                }

                //adding above or below neighbor nodes, if there are any (rows > 1)
                if(rows > 1) {
                    if (i == 0) {
                        dstNodeId = this.computeNodeId(i + 1, j, cols);
                        this.addEdge(srcNodeId, dstNodeId);
                    } else if (i == rows - 1) {
                        dstNodeId = this.computeNodeId(i - 1, j, cols);
                        this.addEdge(srcNodeId, dstNodeId);
                    } else {
                        dstNodeId = this.computeNodeId(i - 1, j, cols);
                        this.addEdge(srcNodeId, dstNodeId);
                        dstNodeId = this.computeNodeId(i + 1, j, cols);
                        this.addEdge(srcNodeId, dstNodeId);
                    }
                }
            }
```

```
        }
    }

    private int computeNodeId(int i, int j, int cols) {
        // Example
        // [0][0] = 0 * 2 + 0 = 0
        // [0][1] = 0 * 2 + 1 = 1
        // [1][0] = 1 * 2 + 0 = 2
        // [1][1] = 1 * 2 + 1 = 3
        return i * cols + j;
    }


    private void init(Queue<Integer> queue, int[][] matrix, boolean[] visited, int[] dist) {
        //Traverse all matrix elements. If it is a 1, then insert associated node id into queue
        if(matrix.length == 0) {
            return;
        }

        int rows = matrix.length;
        int cols = matrix[0].length;

        for(int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                int nodeId = computeNodeId(i, j, cols);

                if(matrix[i][j] == 1) {
                    queue.offer(nodeId);
                    visited[nodeId] = true;
                } else {
                    dist[nodeId] = Integer.MAX_VALUE;
                }
            }
        }
    }

    public int[] bfs(int[][] matrix) {
        int[] dist = new int[numOfVertices];
        boolean visited[] = new boolean[numOfVertices];

        Queue<Integer> queue = new LinkedList<Integer>();

        // Mark the current node as visited and enqueue it
        init(queue, matrix, visited, dist);
```

```
        while (queue.size() != 0) {
            Integer nodeId = queue.poll();

            // Get all adjacent vertices of the dequeued vertex s
            // If a adjacent has not been visited, then mark it
            // visited and enqueue it
            for (Integer adjNodeId : adj[nodeId]) {
                if (!visited[adjNodeId] ) {
                    //find minimum value among adjaNodeId and its neighbors + 1
                    dist[adjNodeId] = Math.min(dist[adjNodeId], dist[nodeId] + 1);

                    queue.offer(adjNodeId);
                    visited[adjNodeId] = true;
                }
            }
        }

        return dist;
    }
}
```

5.3 Clone Graph / LeetCode 133 / Medium

5.3.1 Description

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 val (int) and a list (List[Node]) of its neighbors.

5.3.2 Example

5.3.3 Solution - DFS

5.3.3.1 Walkthrough\

Have a map as cache to store the mapping between original and new node. Recursively walk through the each adjacent node and assign into the adjacency list if existed in cache; otherwise, recursively clone the node.

5.3.3.2 Analysis\

Since each edge will appear twice. Once in the adjacency list of either end of the edge. So, the overall complexity will be $O(|V|) + O(2 \cdot |E|) = O(|V + E|)$

5.3.3.3 Algorithm\

dfs 1.1.1

5.3.4 Java Code

```
//use this map to store original and cloned node.
//also act as visited[]
Map<Node, Node> map = new HashMap<>();

public Node cloneGraph(Node node) {
    map.put(node, new Node(node.val, new ArrayList<Node>()));

    for(Node neighbor: node.neighbors){
        if(map.containsKey(neighbor)) {
            //if neighbor is cloned
            Node cloneNode = map.get(node);
            Node cloneNeighbor = map.get(neighbor);
            cloneNode.neighbors.add(cloneNeighbor);
        } else {
            //recursively clone and add the neighbor
            Node cloneNode = map.get(node);
            cloneNode.neighbors.add(cloneGraph(neighbor));
        }
    }

    return map.get(node);
}
```

5.3.5 Solution - BFS

5.3.5.1 Walkthrough\

We use BFS to traverse the graph with a map cache to store the original and cloned node. This cache can only be used as visited[].

5.3.5.2 Analysis\

For every single vertex in the graph, we will end up visiting its neighboring nodes only twice. The time complexity is the sum of the vertices and their edges as represented by the graph in its adjacency list representation: $O(|V| + |E|)$

5.3.5.3 Algorithm\

bfs 1.1.2

5.3.6 Java Code

```
public Node cloneGraph(Node root) {
    //use this map to store original and cloned node
    //also act as visited[]
    Map<Node, Node> map = new HashMap<>();
    Queue<Node> queue = new ArrayDeque<>();

    queue.offer(root);
    map.put(root, new Node(root.val, new ArrayList<>()));

    while (!queue.isEmpty()) {
        Node node = queue.poll();

        for (Node neighbor : node.neighbors) {
            // neighbor node is not yet cloned
            if (!map.containsKey(neighbor)) {
                map.put(neighbor, new Node(neighbor.val, new ArrayList<>()));
                queue.offer(neighbor);
            }

            //assign the mapping righ tafter
            Node cloneNode = map.get(node);
            Node cloneNeighbor = map.get(neighbor);
            cloneNode.neighbors.add(cloneNeighbor);
        }
    }

    return map.get(root);
}
```

5.4 Course Schedule I / LeetCode 207 / Medium

5.4.1 Description

There are a total of n courses you have to take, labeled from 0 to n-1.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: [0,1]

Given the total number of courses and a list of prerequisite pairs, is it possible for you to finish all courses?

5.4.2 Example

5.4.2.1 Example 1

Input: 2, [[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.

5.4.2.2 Example 2

Input: 2, [[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.

5.4.3 Solution - Topological Sort via BFS

5.4.3.1 Walkthrough\

This problem can be converted to finding if a graph contains a cycle using topological sort

5.4.3.2 Analysis\

A BFS would cost $O(|V| + |E|)$

5.4.3.3 Algorithm\

topological 1.1.3, bfs 1.1.2

5.4.4 Java Code

```java
public boolean canFinish(int numCourses, int[][] prerequisites) {
    if(numCourses == 0 || prerequisites.length == 0){
        return true;
    }

    // counter for number of prerequisites
    int[] numPrereq = new int[numCourses];
    for(int i = 0; i < prerequisites.length; i++){
        numPrereq[prerequisites[i][0]]++;
    }

    //store courses that have no prerequisites
    Queue<Integer> queue = new LinkedList<Integer>();
        for(int i = 0; i < numCourses; i++){
        if(numPrereq[i] == 0){
            queue.offer(i);
        }
    }

    // store the topological sorting order
    List<Integer> order = new ArrayList<>();

    while(!queue.isEmpty()) {
        int vertex = queue.poll();
        order.add(vertex);

        for(int i = 0; i < prerequisites.length; i++){
            // if a course's prerequisite can be satisfied by a course in queue
            if(prerequisites[i][1] == vertex) {
                numPrereq[prerequisites[i][0]]--;

                if(numPrereq[prerequisites[i][0]] == 0) {
                    queue.offer(prerequisites[i][0]);
                }
            }
        }
    }

    return order.size() == numCourses;
}
```

5.4.5 Solution - Topological Sort via DFS

5.4.5.1 Walkthrough\

5.4.5.2 Analysis\

A DFS would cost $O(|V| + |E|)$

5.4.5.3 Algorithm\

topological 1.1.3, dfs 1.1.1

5.4.6 Java Code

```java
public boolean canFinish(int numCourses, int[][] prerequisites) {
    if(numCourses == 0 || prerequisites.length == 0){
        return true;
    }

    //track state of visited courses
    // -1: VISITING
    //  1: VISITED
    int[] visited = new int[numCourses];

    // use the map to store what courses depend on a course
    // can transform into an adjcency list
    Map<Integer, List<Integer>> adjMap = new HashMap<>();
    for(int[] edge: prerequisites){
        if(adjMap.containsKey(edge[1])) {
            adjMap.get(edge[1]).add(edge[0]);
        } else {
            List<Integer> list = new ArrayList<Integer>();
            list.add(edge[0]);
            adjMap.put(edge[1], list);
        }
    }

    for(int i = 0; i < numCourses; i++) {
        if(!dfs(adjMap, visited, i)) {
            return false;
        }
    }
```

```
    return true;
}

private boolean dfs(Map<Integer, List<Integer>> adjMap, int[] visited, Integer vertex)
    if(visited[vertex] == -1) {
        return false;
    }
    if(visited[vertex] == 1) {
        return true;
    }

    visited[vertex] = -1;

    if(adjMap.containsKey(vertex)){
        for(Integer adjVertex: adjMap.get(vertex)){
            if(!dfs(adjMap, visited, adjVertex)) {
                return false;
            }
        }
    }

    visited[vertex] = 1;

    return true;
}
```

5.5 Course Schedule II / LeetCode 210 / Medium

5.5.1 Description

There are a total of n courses you have to take, labeled from 0 to n-1.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: [0,1]. Given the total number of courses and a list of prerequisite pairs, return the ordering of courses you should take to finish all courses. There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all courses, return an empty array.

5.5.2 Example

5.5.2.1 Example 1

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

5.5.2.2 Example 2

Input: 4, [[1,0],[2,0],[3,1],[3,2]] Output: [0,1,2,3] or [0,2,1,3]

5.5.3 Solution - Topological Sort via BFS

5.5.3.1 Walkthrough\

This problem can be converted to finding if a graph contains a cycle using topological sort

5.5.3.2 Analysis\

A BFS would cost $O(|V| + |E|)$

5.5.3.3 Algorithm\

topological 1.1.3, bfs 1.1.2

5.5.4 Java Code

```
public int[] findOrder(int numCourses, int[][] prerequisites) {
    //if there is no prerequisites, return a sequence of courses
    if(prerequisites.length == 0){
        int[] res = new int[numCourses];

        for(int i = 0; i < numCourses; i++){
            res[i] = i;
        }

        return res;
    }

    // counter for number of prerequisites
    int[] numPrereq = new int[numCourses];
```

```
    for(int i = 0; i < prerequisites.length; i++){
        numPrereq[prerequisites[i][0]]++;
    }

    //store courses that have no prerequisites
    Queue<Integer> queue = new LinkedList<Integer>();
    for(int i = 0; i < numCourses; i++){
        if(numPrereq[i] == 0){
            queue.offer(i);
        }
    }

    // store the topological sorting order
    List<Integer> order = new ArrayList<>();

    while(!queue.isEmpty()) {
        int vertex = queue.poll();
        order.add(vertex);

        for(int i = 0; i < prerequisites.length; i++){
            // if a course's prerequisite can be satisfied by a course in queue
            if(prerequisites[i][1] == vertex) {
                numPrereq[prerequisites[i][0]]--;

                if(numPrereq[prerequisites[i][0]] == 0) {
                    queue.offer(prerequisites[i][0]);
                }
            }
        }
    }

    int numOfPreReq = order.size();
    if(numOfPreReq == numCourses) {
        return toArray(order);
    } else {
        return new int[0];
    }
}

private int[] toArray(List<Integer> list) {
    int[] result = new int[list.size()];

    for(int i = 0; i < list.size(); i++) {
        int item = list.get(i);
        result[i] = item;
```

```
    }

    return result;
}
```

5.6 Graph Validate Tree / LeetCode 261 / Medium

5.6.1 Description

Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), check if these edges form a valid tree.

5.6.2 Example

5.6.3 Solution - Topological Sorting via DFS

5.6.3.1 Walkthrough\

This problem can be converted to finding the cycle from a graph using Topological Sorting

5.6.3.2 Analysis\

A DFS would cost $O(|V| + |E|)$

5.6.3.3 Algorithm\

topological 1.1.3, dfs 1.1.1

5.6.4 Java Code - Topological Sorting via DFS

```
public boolean validTree(int n, int[][] edges) {
    HashMap<Integer, ArrayList<Integer>> map = new HashMap<Integer, ArrayList<Integer>>();
    for(int i=0; i<n; i++){
        ArrayList<Integer> list = new ArrayList<Integer>();
        map.put(i, list);
    }
```

```
    for(int[] edge: edges){
        map.get(edge[0]).add(edge[1]);
        map.get(edge[1]).add(edge[0]);
    }

    boolean[] visited = new boolean[n];

    if(!helper(0, -1, map, visited))
        return false;

    for(boolean b: visited){
        if(!b)
            return false;
    }

    return true;
}

private boolean helper(int curr, int parent,
    HashMap<Integer, ArrayList<Integer>> map, boolean[] visited){
    if(visited[curr])
        return false;

    visited[curr] = true;

    for(int i: map.get(curr)){
        if(i!=parent && !helper(i, curr, map, visited)){
            return false;
        }
    }

    return true;
}
```

5.6.5 Solution - Topological Sorting via BFS

5.6.5.1 Walkthrough\

This problem can be converted to finding the cycle from a graph using Topological Sorting

5.6.5.2 Analysis\

A BFS cost $O(|V| + |E|)$

5.6.5.3 Algorithm\

topological 1.1.3, bfs 1.1.2

5.6.6 Java Code - Topological Sorting via BFS

```
public boolean validTree(int n, int[][] edges) {
    ArrayList<ArrayList<Integer>> list = new ArrayList<>();
        for(int i=0; i<n; i++){
        list.add(new ArrayList<>());
    }

    //build the graph
    for(int[] edge: edges){
        int a = edge[0];
        int b = edge[1];

        list.get(a).add(b);
        list.get(b).add(a);
    }

    //use queue to traverse the graph
    HashSet<Integer> visited = new HashSet<>();
    LinkedList<Integer> q = new LinkedList<>();
    q.offer(0);

    while(!q.isEmpty()){
        int head = q.poll();

        if(visited.contains(head)){
            return false;
        }

        visited.add(head);

        ArrayList<Integer> vList = list.get(head);
        for(int v: vList){
            if(!visited.contains(v)){
                q.offer(v);
```

```
            }
        }
    }

    if(visited.size()<n){
        return false;
    }

    return true;
}
```

Chapter 6

Linked List

6.1 Add Two Numbers / Leet Code 2 / Medium

6.1.1 Description

You are given two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

6.1.2 Example

Input:

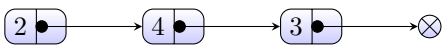


Figure 6.1: Some caption.

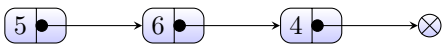


Figure 6.2: Some caption.

-

Output:

Explanation: 342 + 465 = 807.

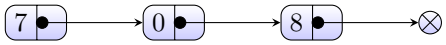


Figure 6.3: Some caption.

6.1.3 Solution

6.1.3.1 Walkthrough\

First add two number digit by digit until one number run out of digit. Copy the other number for the remaining of digits.

6.1.3.2 Analysis\

Time complexity is O(n + m) as every node is visited once.

6.1.3.3 Algorithm\

6.1.4 Java Code

```
public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
    ListNode currentN1 = l1;
    ListNode currentN2 = l2;
    ListNode currentDigit = new ListNode(0);
    ListNode resultHead = currentDigit;

    int digitCarry = 0;
    int digitSum = 0;

    while(currentN1 != null && currentN2 != null) {
        digitSum = currentN1.val + currentN2.val + digitCarry;
        digitCarry = digitSum / 10;
        digitSum %= 10;

        currentDigit.next = new ListNode(digitSum);
        currentDigit = currentDigit.next;

        currentN1 = currentN1.next;
        currentN2 = currentN2.next;
    }

    //Copy the remaining of number 1
    while(currentN1 != null) {
```

```
        digitSum = currentN1.val + digitCarry;
        digitCarry = digitSum / 10;
        digitSum %= 10;

        currentDigit.next = new ListNode(digitSum);
        currentDigit = currentDigit.next;

        currentN1 = currentN1.next;
    }

    //Copy the remaining of number 2
    while(currentN2 != null) {
        digitSum = currentN2.val + digitCarry;
        digitCarry = digitSum / 10;
        digitSum %= 10;

        currentDigit.next = new ListNode(digitSum);
        currentDigit = currentDigit.next;

        currentN2 = currentN2.next;
    }

    if(digitCarry > 0) {
        currentDigit.next = new ListNode(digitCarry);
        currentDigit = currentDigit.next;
    }

    return resultHead.next;
}
```

6.2 Reverse Linked List / Leet Code 206 / Easy

6.2.1 Description

Reverse a singly linked list.

6.2.2 Example

Input:

Output:

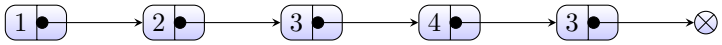


Figure 6.4: Some caption.

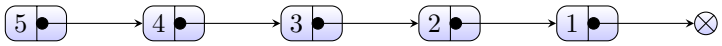


Figure 6.5: Some caption.

6.2.3 Solution - Recursive

6.2.3.1 Walkthrough\

For recursive solution, have a pointer remember original next and swap its link with the original head. Call the method recursively.

6.2.3.2 Analysis\

Time complexity is O(n) since every node is visited.

6.2.3.3 Algorithm\

recursive **??**

6.2.4 Java Code - Recursive

```
public ListNode reverseList(ListNode node) {
    if(node == null) {
        return null;
    } else if(node.next == null) {
        return node;
    } else {
        ListNode origNext = node.next;
        ListNode newNode = reverseList(origNext);
        //origNext -> node
        origNode.next = node;
        //delete link
        node.next = null;
        return newNode;
    }
}
```

6.2.5 Solution - Iterative

6.2.5.1 Walkthrough\

For iterative solution, have a helper pointer to remember the previous position and reverse the links between current and headPrev nodes while iterating the list.

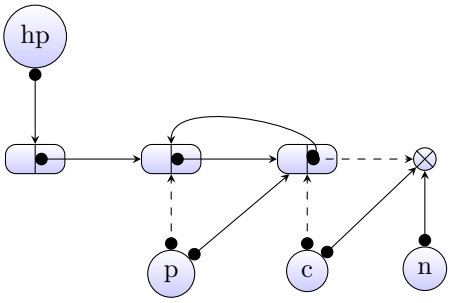


Figure 6.6: Some caption.

Finally, reset where begin.next points to head of the list after reverse - tail of the original list, and return the head of the reversed list.

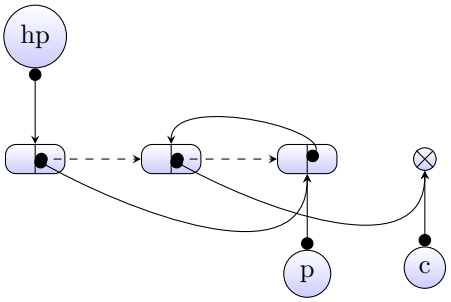


Figure 6.7: Some caption.

6.2.5.2 Analysis\

Time complexity is O(n) since every node is visited.

6.2.5.3 Algorithm\

6.2.6 Java Code - Iterative

```
public ListNode reverseList(ListNode head) {
    if(head == null || head.next == null) {
        return head;
    }

    ListNode headPrev = new ListNode(0);
    headPrev.next = head;

    return reverse(headPrev);
}

public ListNode reverse(ListNode begin) {
    ListNode prev = begin.next;
    ListNode current = prev.next;

    //reverse links between current and prev
    while (current != null) {
        ListNode next = current.next;
        /*
        * swap links between current and prev
        */
        current.next = prev;
        prev = current;
        current = next;
    }

    begin.next.next = current;
    begin.next = prev;

    return begin.next;
}
```

6.3 Reverse Linked List II / Leet Code 92 / Medium

6.3.1 Description

Reverse a linked list from position m to n. Do it in one-pass

6.3.2 Example

Input:

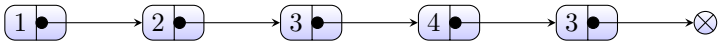


Figure 6.8: Some caption.

, m = 2, n = 4

Output:

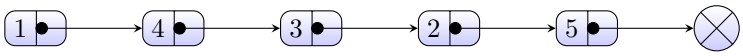


Figure 6.9: Some caption.

6.3.3 Solution

6.3.3.1 Walkthrough\

Have a helper pointer to to move to (M-1) position and reverse the links between current and prev nodes while iterating the list for (N-M) nodes.

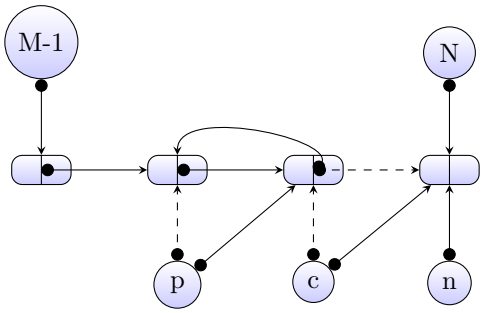


Figure 6.10: Some caption.

Finally, reset where begin.next points to head of the list after reverse - tail of the original list, and return the tail of the reversed list.

6.3.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

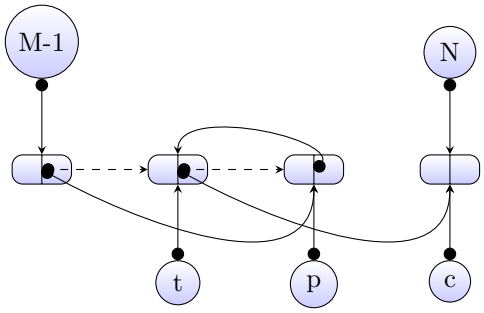


Figure 6.11: Some caption.

6.3.3.3 Algorithm\

6.3.4 Java Code

```
public ListNode reverseBetween(ListNode head, int m, int n) {
    if(head == null) {
        return head;
    }

    //headHelper.next is original head
    ListNode headPrev = new ListNode(0);
    headPrev.next = head;

    ListNode mMinus1 = headPrev;
    //move to starting position
    for (int i = 0; i < m - 1; i++) {
        mMinus1 = mMinus1.next;
    }

    //reverse node between (m, n)
    reverseList(mMinus1, n-m);

    return headPrev.next;
}

public ListNode reverseList(ListNode begin, int length) {
    ListNode prev = begin.next;
    ListNode current = prev.next;

    //reverse links between current and prev
    int i = 0;
```

```
    while (i < length) {
        ListNode next = current.next;
        /*
        * swap links between current and prev
        */
        current.next = prev;
        prev = current;
        current = next;
        i++;
    }

    // tail of reversed is head of original list
    ListNode tail = begin.next;

    begin.next.next = current;
    begin.next = prev;

    return tail;
}
```

6.4 Reverse Nodes in k-Group / LeetCode 25 / Hard

6.4.1 Description

Given a linked list, reverse the nodes of a linked list k at a time and return its modified list. If the number of nodes is not a multiple of k then left-out nodes in the end should remain as it is. You may not alter the values in the nodes, only nodes itself may be changed.

6.4.2 Example

Input:

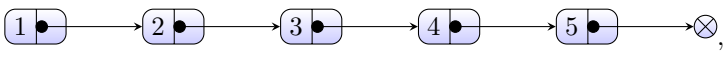


Figure 6.12: Some caption.

k = 2 becomes

k=3 becomes

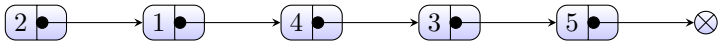


Figure 6.13: Some caption.

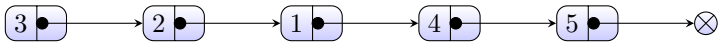


Figure 6.14: Some caption.

6.4.3 Solution

6.4.3.1 Walkthrough\

Have a index to remember when to start reversing and when to stop and reverse the links between current and prev nodes while iterating the list until the stop position is met.

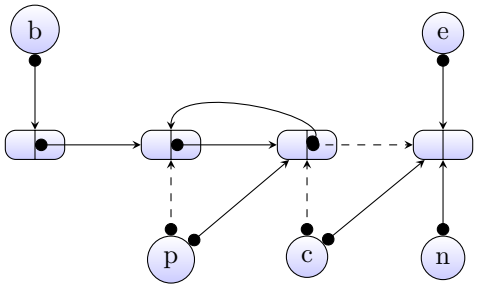


Figure 6.15: Some caption.

Finally, reset where begin.next points to head of the list after reverse - tail of the original list, and return the tail of the reversed list.

6.4.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

6.4.3.3 Algorithm\

6.4.4 Java Code

```
public ListNode reverseKGroup(ListNode head, int k) {
    if(head==null || k==1) {
        return head;
```

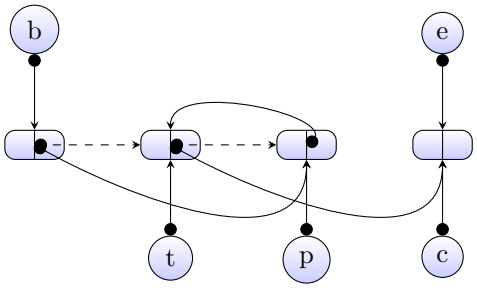


Figure 6.16: Some caption.

```
    }

    ListNode headPrev = new ListNode(0);
    headPrev.next = head;
    ListNode prev = headPrev;
    int i = 0;

    ListNode current = head;
    while(current!=null){
        i++;
        if((i % k) == 0){
            ListNode revTail = reverseList(prev, current.next);
            prev = revTail;
            current = revTail.next;
        }else{
            current = current.next;
        }
    }

    return headPrev.next;
}


public ListNode reverseList(ListNode begin, ListNode end) {
    ListNode prev = begin.next;
    ListNode current = prev.next;

    //reverse links between current and prev
    while (current != end) {
        ListNode next = current.next;
        /*
        * swap links between current and prev
```

```
        */
        current.next = prev;
        prev = current;
        current = next;
    }

    // tail of reversed is head of original list
    ListNode tail = begin.next;

    begin.next.next = end;
    begin.next = prev;

    return tail;
}
```

6.5 Linked List Cycle / Leet Code 141 / Easy

6.5.1 Description

Given a linked list, determine if it has a cycle in it.

To represent a cycle in the given linked list, we use an integer pos which represents the position (0-indexed) in the linked list where tail connects to. If pos is -1, then there is no cycle in the linked list.

6.5.2 Example

Input:

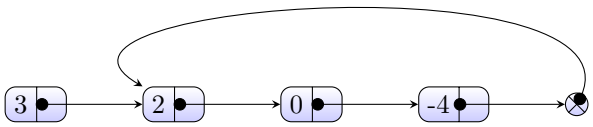


Figure 6.17: Some caption.

Output: true. Explanation: There is a cycle in the linked list, where tail connects to the second node.

Input:

Output: true. Explanation: There is a cycle in the linked list, where tail connects to the first node.

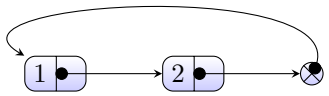


Figure 6.18: Some caption.

Input:

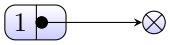


Figure 6.19: Some caption.

Output: false. Explanation: There is no cycle in the linked list.

6.5.3 Solution

6.5.3.1 Walkthrough\

Having two pointers, one running twice as fast as the other(p1 = p1.next; p2 = p2.next.next). If there is a cycle, eventually (p1 == p2) representing a node in the cycle. A better traversal algorithm would make p1 at the middle of the cycled list whereas p2 at the tail of the cycled list.

6.5.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

6.5.3.3 Algorithm\

6.5.4 Java Code

```
public boolean hasCycle(ListNode head) {
    if(head == null) {
        return false;
    }

    ListNode fast = head, slow = head;

    while(fast != null && fast.next != null) {
        fast = fast.next.next;
        slow = slow.next;
```

```
        if(fast == slow) {
            return true;
        }
    }

    return false;
}
```

6.6 Linked List Cycle II / Leet Code 142 / Medium

6.6.1 Description

Given a linked list, return the node where the cycle begins. If there is no cycle, return null.

To represent a cycle in the given linked list, we use an integer pos which represents the position (0-indexed) in the linked list where tail connects to. If pos is -1, then there is no cycle in the linked list.

Note: Do not modify the linked list.

6.6.2 Example

Input:

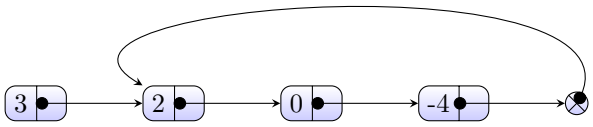


Figure 6.20: Some caption.

Output: tail connects to node index 1. Explanation: There is a cycle in the linked list, where tail connects to the second node.

Input:

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

Input:

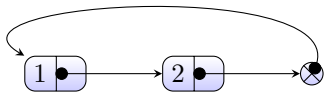


Figure 6.21: Some caption.

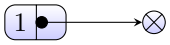


Figure 6.22: Some caption.

Output: null (no cycle). Explanation: There is no cycle in the linked list.

6.6.3 Solution

6.6.3.1 Walkthrough\

Having two pointers, one talks twice as fast as the other. If there is a cycle (p1 == p2), p1 would be at the middle of the cycled list whereas p2 at the tail of the cycled list. In other words, *# of steps taken from header to p1 equals to # of steps taken from p1 to p2.*

6.6.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

6.6.3.3 Algorithm\

6.6.4 Java Code

```
public ListNode detectCycle(ListNode head) {
    if(head == null || head.next == null) {
        return null;
    }

    ListNode fast = head, slow = head;
    while(fast != null && fast.next != null) {
        fast = fast.next.next;
        slow = slow.next;

        if(fast == slow) {
            break;
```

```
        }
    }

    if(fast == slow) {
        ListNode cycleHead = head;
        /*
         * slow is at middle of CYCLED list
         * Initially, cycleHead is at header
         * cycleHead --> slow == slow --> fast
         */
        while(fast != cycleHead) {
            fast = fast.next;
            cycleHead = cycleHead.next;
        }

        return cycleHead;
    } else {//no cycle
        return null;
    }
}
```

6.7 Odd Even Linked List / Leet Code 328 / Medium

6.7.1 Description

Given a singly linked list, group all odd nodes together followed by the even nodes. Please note here we are talking about the node number and not the value in the nodes. You should try to do it in place.

6.7.2 Example

Input:

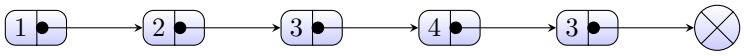


Figure 6.23: Some caption.

Output:

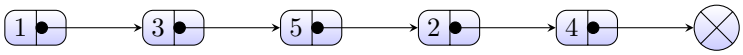


Figure 6.24: Some caption.

6.7.3 Solution

6.7.3.1 Walkthrough\

Have two helper pointers to construct to list: one for odd nodes and one for even nodes. Assign those nodes accordingly while traversing the list. Finally, connect odd nodes list to even nodes list.

6.7.3.2 Analysis\

The program should run in O(1) Auxiliary Space and O(n) time complexity.

6.7.3.3 Algorithm\

6.7.4 Java Code

```
public ListNode oddEvenList(ListNode head) {
    if(head == null || head.next == null || head.next.next == null ) {
        return head;
    }

    ListNode oddHeadPrev = new ListNode(0), currentOdd = oddHeadPrev;
    ListNode evnHeadPrev = new ListNode(0), currentEvn = evnHeadPrev;
    ListNode current = head;

    int index = 1;

    while(current != null) {
        // Same as index % 2 == 1
        if( (index & 1) > 0 ) {
            currentOdd.next = current;
            currentOdd = currentOdd.next;
        } else {
            currentEvn.next = current;
            currentEvn = currentEvn.next;
        }

        index++;
```

```
            current = current.next;
        }

        //connect odd nodes list with even nodes list
        currentOdd.next = evnHeadPrev.next;

        //connect event nodes to null
        currentEvn.next = null;

        return oddHeadPrev.next;
    }
```

6.8 Merge Two Sorted Lists / Leet Code 21 / Easy

6.8.1 Description

Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists.

6.8.2 Example

6.8.3 Solution

6.8.3.1 Walkthrough\

This is the same as merge() to sort a list in Leet Code 148. First we will create a new ListNode and a current pointer. While traversing the node, compre the nodes at each list and nodes with smaller value will attach to the final sorted list. Exit traversal when either list hits the end. If there are nodes remained in either list, connect to that remaining nodes of the list.

6.8.3.2 Analysis\

The cost for each merge is O(n) as every node is visited once in the list. There are two lists, thus the overall time complexity remains O(n).

6.8.3.3 Algorithm\

6.8.4 Java Code

```
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    ListNode headPrev = new ListNode(0), current = headPrev;

    while(l1 != null && l2 != null) {
        if(l1.val < l2.val) {
            //l1 < l2
            current.next = l1;
            l1 = l1.next;
        } else {
            current.next = l2;
            l2 = l2.next;
        }
        current = current.next;
    }

    // connect to the rest of the remaining nodes
    if(l1 != null) {
        current.next = l1;
    } else {
        // l2 = remaining of list or null
        current.next = l2;
    }

    return headPrev.next;
}
```

6.9 Merge k Sorted Lists / Leet Code 23 / Hard

6.9.1 Description

Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

6.9.2 Example

6.9.3 Solution - Merge lists one by one

6.9.3.1 Walkthrough\

One solution is to leverage the previous problem of merging 2 list.

6.9.3.2 Analysis\

We will need to merge two lists for (k-1) times, where each merge() will cost O(n) in time complexity and O(1) in Auxiliary Space. The total cost for merge is $\sum_{i=1}^{k-1}(i * \frac{n}{k} + \frac{n}{k}) = O(k \cdot n)$ in time complexity, where n is the total node number in k list. and O(1) in Auxiliary Space. However, this would not be an efficient solution if k is large.

6.9.3.3 Algorithm\

6.9.4 Java Code - Merge lists one by one

```
public ListNode mergeKLists(ListNode[] lists) {
    if(lists.length == 0) {
        return null;
    }

    ListNode newHead = lists[0];

    //merge k-1 times
    for(int i = 1; i < lists.length; i++) {
        newHead = merge(newHead, lists[i]);
    }

    return newHead;
}

public ListNode merge(ListNode l1, ListNode l2) {
    ListNode headPrev = new ListNode(0), current = headPrev;

    while(l1 != null && l2 != null) {
        if(l1.val < l2.val) {
            //l1 < l2
            current.next = l1;
            l1 = l1.next;
```

```
        } else {
            current.next = l2;
            l2 = l2.next;
        }
        current = current.next;
    }

    // connect to the rest of the remaining nodes
    if(l1 != null) {
        current.next = l1;
    } else {
        current.next = l2;
    }

    return headPrev.next;
}
```

6.9.5 Solution - Min Heap

6.9.5.1 Walkthrough\

We could compare every leading elements in k lists and retrieve the node with the smallest value using Min Heap.

6.9.5.2 Analysis\

Since the original k lists were sorted, therefore, we could retrieve all $n \cdot k$ elements in sorted order by retrieve the min element from PriorityQueue (min heap). The comparison cost is O(log k) for every pop and insertion to the priority queue since there are at most k elements (from k list). Additionally, there are a total number of n nodes in all lists. Thus, the overall time complexity is $O(\log k \cdot n)$ and Auxiliary Space for the final list is O(n), where n is the total node number in all k lists.

6.9.5.3 Algorithm\

bfs 1.1.2, heap 1.1.15

6.9.6 Java Code - Min Heap

```
public ListNode mergeKLists(ListNode[] lists) {
    if(lists.length == 0) {
        return null;
    }

    ListNode head = null, current = null;

    PriorityQueue<ListNode> queue = new PriorityQueue<>(new Comparator<ListNode>() {
        @Override
        public int compare(ListNode node1, ListNode node2) {
            return node1.val - node2.val;
        }
    });

    //insert all heads into the priority queue
    for(ListNode headOfList : lists) {
        if(headOfList != null) {
            queue.offer(headOfList);
        }
    }


    while(!queue.isEmpty()) {
        ListNode node = queue.poll();

        //add next node into priority queue
        if(node.next != null) {
            queue.offer(node.next);
        }

        // construct the list with the min element from queue
        if(head == null) {
            head = node;
            current = node;
        } else {
            current.next = node;
            current = current.next;
        }
    }

    return head;
}
```

6.10 Palindrome Linked List / Leet Code 234 / Easy

6.10.1 Description

Given a singly linked list, determine if it is a palindrome.

6.10.2 Example

Input:

Figure 6.25: Some caption.

return false

Output:

Figure 6.26: Some caption.

return true

6.10.3 Solution

6.10.3.1 Walkthrough\

Locate the mid node, and use that to reverse the 2nd half of the list. We will have two sublists: [o, ..., m] and [r, ..., m].

Event number of list:

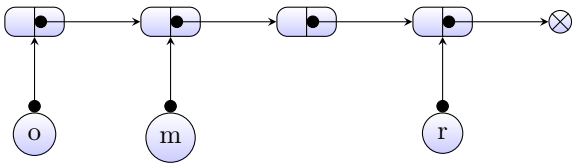


Figure 6.27: Some caption.

Odd number of list:

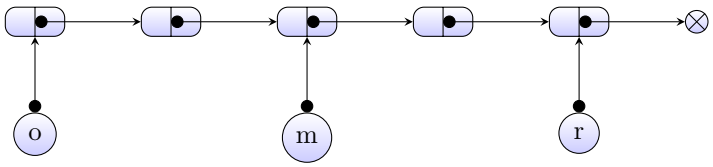


Figure 6.28: Some caption.

Traverse the two sublists, beginning at head of original and head of reversed sublists respectively. If any node is not equal, then it is NOT palindrome.

6.10.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

6.10.3.3 Algorithm\

6.10.4 Java Code

```
public boolean isPalindrome(ListNode head) {
    if(head == null || head.next == null) {
        return true;
    }

    ListNode mid = head, tail = head;

    //locating mid and tail nodes
    while(tail.next != null && tail.next.next != null) {
        mid = mid.next;
        tail = tail.next.next;
    }

    // retrieve the head of reversed 2nd half sublist
    ListNode reversed2ndHalfHead = reverse(mid);

    ListNode currentOrg = head, currentRev = reversed2ndHalfHead;

    while(currentOrg != null && currentRev != null) {
        //if any in original 1st half does not match reversed 2nd half
        if(currentOrg.val != currentRev.val) {
            return false;
        }
```

```
        currentOrg = currentOrg.next;
        currentRev = currentRev.next;
    }

    return true;
}

public ListNode reverse(ListNode head) {
    ListNode headPrev = new ListNode(0);
    ListNode current = head;

    //reverse links between current and prev
    while (current != null) {
        ListNode next = current.next;

        /*
        * swap links between current and prev
        */
        current.next = headPrev.next;
        headPrev.next = current;
        current = next;
    }

    return headPrev.next;
}
```

6.11 Intersection of Two Linked Lists / Leet Code 160 / Easy

6.11.1 Description

Write a program to find the node at which the intersection of two singly linked lists begins.

```
*  If the two linked lists have no intersection at all, return null.
* The linked lists must retain their original structure after the function returns.
* You may assume there are no cycles anywhere in the entire linked structure.
* Your code should preferably run in O(n) time and use only O(1) memory.
*  The intersected nodes must continue to the end of both lists.
* If two nodes intersect, two node pointers refer to the same ListNode object in memory.
```

6.11.2 Example

Input:

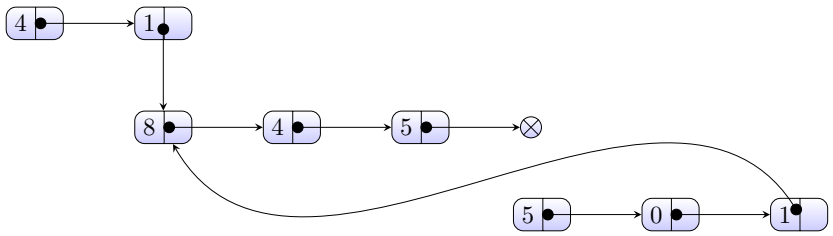


Figure 6.29: Some caption.

Output: Reference of the node with value = 8 Input Explanation: The intersected node's value is 8 (note that this must not be 0 if the two lists intersect). From the head of A, it reads as [4,1,8,4,5]. From the head of B, it reads as [5,0,1,8,4,5]. There are 2 nodes before the intersected node in A; There are 3 nodes before the intersected node in B.

6.11.3 Solution

6.11.3.1 Walkthrough\

First we compute the length of both list and we get rid of the leading extra node of the longer list. For the # of remaining nodes of the longer list should equal to the # of shorter list. We traverse both list, and determine when (p1 == p2) - instead of (p1.val == p2.val)

6.11.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

6.11.3.3 Algorithm\

6.11.4 Java Code

```
public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
    int lenA = 0, lenB = 0;

    //compute length of both lists
    for(ListNode current = headA; current != null; current = current.next, lenA++);
    for(ListNode current = headB; current != null; current = current.next, lenB++);

    //both conditions need to be the same
    ListNode currentLonger = lenA > lenB ? headA : headB;
    ListNode currentShorter = lenA > lenB ? headB : headA;

    //Move currentLonger to the position where remaining of nodes equal to len(currentShorter)
    for(int i = 0; i < Math.abs(lenA - lenB); i++, currentLonger = currentLonger.next);

    while(currentLonger != null && currentShorter != null) {
        //If the node intersect, nodeA == nodeB
        if(currentLonger == currentShorter) {
            //return current node
            return currentLonger;
        }

        currentLonger = currentLonger.next;
        currentShorter = currentShorter.next;
    }

    //no match
    return null;
}
```

6.12 Remove Nth Node From End of List / Leet Code 19 / Medium

6.12.1 Description

Given a linked list, remove the n-th node from the end of list and return its head.

6.12.2 Example

Input:

, and n = 2.

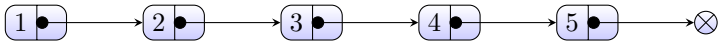


Figure 6.30: Some caption.

Output:

Figure 6.31: Some caption.

6.12.3 Solution - TWO Pass

6.12.3.1 Walkthrough\

The simpliest solution is to do solve this problem in two passes. First round is to find out the length of the list, whileas the second iteration is to iterate to the node posiion $length - n + 1$ or prev node position $length - n$, begining from a dummy node headPrev. Finally, we need to check and move the head node appropriately if we are deleting it.

6.12.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

6.12.3.3 Algorithm\

6.12.4 Java Code - TWO Pass

```
public ListNode removeNthFromEnd(ListNode head, int n) {
    if(head == null) {
        return null;
    }

    //Quick check to see if we are deleting the only node in the list
    if (head.next == null && n <= 1){
        return null;
    }

    int length = 0;

    //1st pass: to get length of the list
```

```
    for(ListNode current = head; current != null; current = current.next) {
        length++;
    }

    // removal position is over list length
    if(n > length) {
        return head;
    }

    //move to the position before the target
    int position = length - n;

    //keep track of current traversed # of nodes
    int count = 0;

    ListNode headPrev = new ListNode(0);
    headPrev.next = head;
    ListNode removedPrev = headPrev;

    //2nd pass: move to the one before target
    while( count < position) {
        removedPrev = removedPrev.next;
        count++;
    }

    removedPrev.next = removedPrev.next.next;

    //if removing head, renew head
    if(removedPrev == headPrev) {
        head = removedPrev.next;
    }


    return head;
}
```

6.12.5 Solution - ONE Pass

6.12.5.1 Walkthrough\

The simpliest solution is to do solve this problem in two passes. First round is to find out the length of the list, whileas the second iteration is to iterate to the node posiion $length - n + 1$ or prev node position $length - n$, begining from a dummy node headPrev. Finally, we need to check and move the head node

appropriately if we are deleting it.

6.12.5.2 Analysis\

Time complexity is O(n) as every node is visited once.

6.12.6 Java Code - ONE Pass

```java
public ListNode removeNthFromEnd(ListNode head, int n) {
    //Quick check to see if we are deleting the only node in the list
    if (head.next == null && n == 1){
        return null;
    }

    //keep track of current traversed # of nodes
    int count = 0;

    //Add a node to the begining of our linked list to make deletions easier at the to
    ListNode headPrev = new ListNode(0);
    headPrev.next = head;

    ListNode current = headPrev;
    ListNode removedPrev = headPrev;

    while(current != null) {
        count++;

        if( (count - n) > 1 ) {
            removedPrev = removedPrev.next;
        }

        if(current.next == null) {
            //the last node, removedPrev is also at the right position

            if(removedPrev.next != null && removedPrev.next.next == null) {
                //to remove tail node
                removedPrev.next = null;
            } else {
                // to remove non-tail node (including head)
                removedPrev.next = removedPrev.next.next;
            }

            break;
```

```
        }

        current = current.next;
    }

    return headPrev.next;
}
```

6.13 Insert a Node at the Nth Position in Doubly Linked List / Firebase / Level 3

6.13.1 Description

In doubly linked list, implement a method to insert a node at specified position and return the list's head. Do nothing if insertion position is outside the bounds of the list.

6.13.2 Example

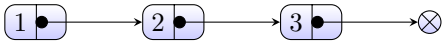


Figure 6.32: Some caption.

given the list, insert a new node at index 2 with value 4.

Result:

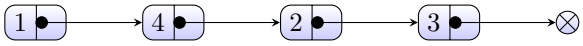
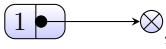


Figure 6.33: Some caption.

If the index exceed the length, return the list unchanged:



, given the list, insert a new node at index 3 with value 4.

Figure 6.34: Some caption.

6.13.3 Solution

6.13.3.1 Walkthrough\

The simpliest solution is to do solve this problem in two passes. First round is to find out the length of the list and check if the request position is over the lenght limit. The second iteration is to traverse to the previous insertion position, begining from a dummy node headPrev. Finally, we need to check if we are inserting a head / a tail or an intermediate node.

6.13.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

6.13.3.3 Algorithm\

6.13.4 Java Code

```
public DoublyLinkedNode insertAtPos(DoublyLinkedNode head, int data, int pos) {
    DoublyLinkedNode newNode = new DoublyLinkedNode(data);

    //Quick check if we are inserting the only node in the list
    if(head == null) {
        if(pos == 1) {
            return newNode;
        } else {
            return head;
        }
    }


    int len = 1;

    //1st pass: to get length of the list
    for(DoublyLinkedNode current = head; current != null; current = current.next) {
        len++;
    }

    //pos is over list length
    if(pos > len) {
        return head;
    }
```

```
    DoublyLinkedNode headPrev = new DoublyLinkedNode(0);
    headPrev.next = head;
    head.prev = headPrev;
    DoublyLinkedNode insertPrev = headPrev;


    //2nd pass: move to before the insertion position
    for(int i = 0; i < (pos - 1); i++) {
        insertPrev = insertPrev.next;
    }

    if(insertPrev == headPrev) {
        // insert head
        head.prev = newNode;
        newNode.next = head;

        return newNode;
    } else if(insertPrev.next == null) {
        // insert tail
        insertPrev.next = newNode;
        newNode.prev = insertPrev;

        return head;
    } else {
        // insert intermediate
        DoublyLinkedNode insertNext = insertPrev.next;

        insertPrev.next = newNode;
        newNode.prev = insertPrev;
        newNode.next = insertNext;
        insertNext.prev = newNode;

        return head;
    }
}
```

6.14 Reorder List / Leet Code 143 / Medium

6.14.1 Description

Given a singly linked list L:

, reorder it to

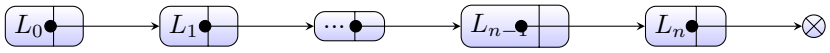


Figure 6.35: Some caption.

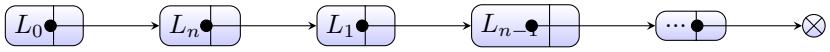


Figure 6.36: Some caption.

You may not modify the values in the list's nodes, only nodes itself may be changed.

6.14.2 Example

Given

Figure 6.37: Some caption.

, reorder it to

6.14.3 Solution

6.14.3.1 Walkthrough\

We could do the following:

```
*  Locate the middle and tail nodes
*  Reverse the nodes in the 2nd half othe list \textbf[in-place]
* Change links between nodes in 1st and 2nd half of list, one by one.
```

6.14.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

6.14.3.3 Algorithm\

6.14.4 Java Code

Figure 6.38: Some caption.

```java
public void reorderList(ListNode head) {
    if (head == null || head.next == null) {
        return;
    }

    //locating middle and tail nodes
    ListNode middle = head;
    ListNode tail = head;
    while (tail.next != null && tail.next.next != null) {
        middle = middle.next;
        tail = tail.next.next;
    }


    /*
    * reverse nodes for the 2nd half of the list
    * L0 > L1 > ... > Ln > L_{n-1} > ...
    */
    ListNode middlePrev = middle;
    ListNode prev = middle.next;

    /*
    * reverse links between prev, current and current.next by FIXING middlePrev
    * 7 > 8 > 9
    * 8 > 7 > 9
    */
    while (prev.next != null) {
        ListNode current = prev.next;
        prev.next = current.next;
        current.next = middlePrev.next;
        middlePrev.next = current;
    }

    //change links between nodes in 1st and 2nd half, one by one.
    ListNode current1 = head;
    ListNode current2 = middlePrev.next;
    while (current1 != middlePrev) {
        middlePrev.next = current2.next;
        current2.next = current1.next;
        current1.next = current2;
```

```
        current1 = current2.next;
        current2 = middlePrev.next;
    }
}
```

6.15 Remove Duplicates from Sorted List / Leet Code 83 / Easy

6.15.1 Description

Given a sorted linked list, delete all duplicates such that each element appear only once.

6.15.2 Example

Input

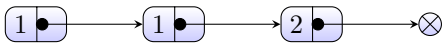


Figure 6.39: Some caption.

Output

Figure 6.40: Some caption.

6.15.3 Solution - non sorted list

6.15.3.1 Walkthrough\

If this is a non sorted list, we need to keep a record (HashSet) and a prev node. Remove the duplicate node if found - set.add() returns false

6.15.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

6.15.3.3 Algorithm\

6.15.4 Java Code - non sorted list

```
public ListNode removeDuplicates(ListNode head) {
    ListNode prev = new ListNode(0);
    prev.next = head;
    ListNode current = head;

    Set<Integer> set = new HashSet<>();

    while(current != null) {
        int value = current.data;

        if(!set.add(value)) {
            //duplicated entry
            prev.next = prev.next.next;
        } else {
            prev = prev.next;
        }
        current = current.next;
    }

    return head;
}
```

6.15.5 Solution - sorted list

6.15.5.1 Walkthrough\

If this is a sorted list, we could utilize the fact that if current.data == next.data. Remove the next node.

6.15.5.2 Analysis\

Time complexity is O(n) as every node is visited once.

6.15.5.3 Algorithm\

6.15.6 Java Code - sorted list

```
public ListNode deleteDuplicates(ListNode head) {
    ListNode current = head;

    while(current != null) {
        if(current.next != null && current.val == current.next.val) {
            //remove the current node
            current.next = current.next.next;
        } else {
            current = current.next;
        }
    }

    return head;
}
```

6.16 Copy List with Random Pointer / LeetCode 138 / Medium

6.16.1 Description

A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null. Return a deep copy of the list.

The Linked List is represented in the input/output as a list of n nodes. Each node is represented as a pair of [val, random_index] where:

```
*  val: an integer representing Node.val
* random\_index: the index of the node (range from 0 to n-1) where random pointer points to
```

does not point to any node.

6.16.2 Example

Deep copy of the following array:

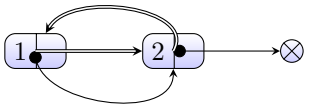


Figure 6.41: Some caption.

6.16.3 Solution

6.16.3.1 Walkthrough

We could not copy random index (⇒) in 1st pass, since the desginated node might not be created yet. Thus, we need to perform in 3 pass:

```
* copy every node, i.e., duplicate every node and next index ($\rightarrow$), and insert it to the lis
```

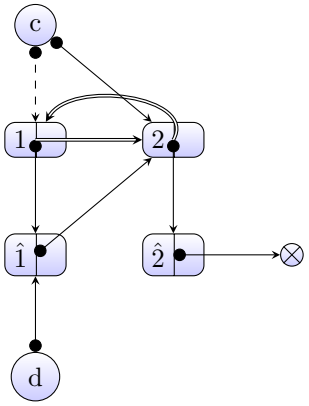


Figure 6.42: Some caption.

- copy random pointers for all newly created nodes
  - break the list to two
    ; and finally,

6.16.3.2 Analysis\

Time complexity is O(n) as every node is visited once.

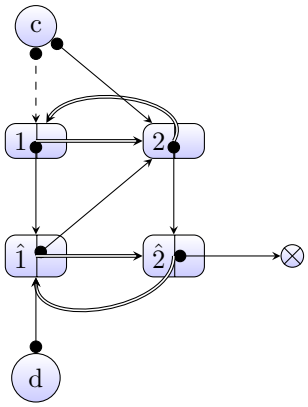


Figure 6.43: Some caption.

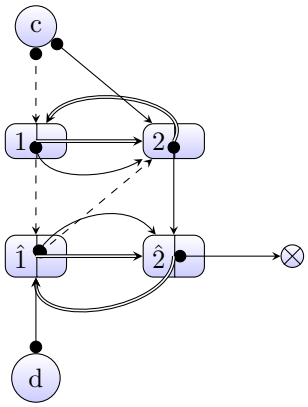


Figure 6.44: Some caption.

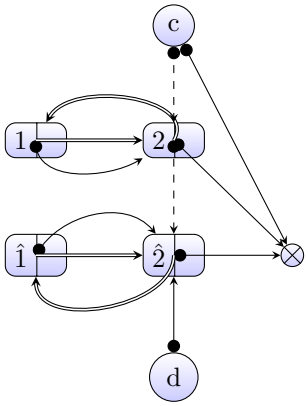


Figure 6.45: Some caption.

6.16.3.3 Algorithm\

6.16.4 Java Code

```java
public Node copyRandomList(Node head) {
    if (head == null) {
        return null;
    }

    // copy every node and insert to list
    Node current = head;
    while (current != null) {
        Node dup = new Node(current.val);
        dup.next = current.next;
        current.next = dup;
        current = dup.next;
    }

    // copy random pointer for each new node
    current = head;
    while (current != null) {
        Node dup = current.next;
        if (current.random != null) {
            dup.random = current.random.next;
        }
        current = dup.next;
    }

    // break list to two
    current = head;
    Node newHead = head.next;
    while (current != null) {
        Node dup = current.next;
        current.next = dup.next;
        if (dup.next != null) {
            dup.next = dup.next.next;
        }
        current = current.next;
    }

    return newHead;
}
```