Technical Interview

Michael Fong

2020-04-04

Contents

1 Introduction **5**
1.1 Glossary 5

2 Arrays **9**
2.1 Two Sums I / LeetCode 1 / Easy 10
2.2 Two Sum II - Return True / False 12
2.3 Two Sum IV - Input is a BST / LeetCode 653 / Easy 13
2.4 TwoSum V / / } 15
2.5 3Sum / Leet Code 15 / Medium} 16
2.6 3 Sum Closest / LeetCode 16 / Medium} 18
2.7 4Sum / Leet Code 18 / Medium} 20
2.8 4Sum II} 22
2.9 Max Gain / Firecode / Level 2} 23
2.10 Pascal's Triangle / Leet Code 118 / Easy} 24
2.11 Search in Rotated Sorted Array / Leet Code 33 / Medium } . . . 26
2.12 Median of Two Sorted Arrays / Leet Code 4 / Hard} 27
2.13 Retrieve List of Elements Appeared at k^{th} Time and in Insertion Order / /} 29
2.14 Permutations / LeetCode 46 / Medium} 31
2.15 Permutations II / LeetCode 47 / Medium} 33
2.16 Subsets / LeetCode 78 / Medium} 35
2.17 Subsets II / LeetCode 90 / Medium} 37
2.18 Sort Array By Parity / LeetCode 906 / Easy} 39
2.19 Merge Intervals / LeetCode 56 / Medium} 40
2.20 Non-overlapping Intervals / LeetCode 435 / Medium} 42
2.21 Interval List Intersections / LeetCode 986 / Medium} 44
2.22 Insert Interval / LeetCode 57 / Hard} 46
2.23 Find Common Characters / LeetCode 1002 / Easy} 49
2.24 Top K Frequently Appeared Elements / Leet Code 347 / Medium} 51
2.25 Count iterations towards filling 1's in an array / / } 52
2.26 Maximum Subarray / Leet Code 53 / Easy} 54
2.27 Maximum Product Subarray / Leet Code 152 / Medium} 56
2.28 Longest Valid Parentheses / Leet Code 32 / Hard} 57

2.29 Longest Increasing Subsequence / Leet Code 300 / Medium} . . 59
2.30 Paint House / Leet Code 256 / Easy} 61
2.31 Climbing Stairs / Leet Code 70 / Easy} 63
2.32 Find the Maximum Number of Repetitions / Firecode / Level 3} 64
2.33 House Robber / Leet Code 198 / Easy} 66
2.34 Best Time to Buy and Sell Stock / Leet Code 121 / Easy} 67
2.35 Best Time to Buy and Sell Stock II / Leet Code 122 / Easy} . . 68
2.36 Best Time to Buy and Sell Stock with Cooldown / Leet Code 309 / Medium} 70
2.37 Minimum Swaps To Make Sequences Increasing / LeetCode 801 / Medium} 71
2.38 Retrieve an Optimal Computation from an Array / / } 73
2.39 Shortest Word Distance / / } 77

3 Literature **81**
3.1 Two Sum IV - Input is a BST / LeetCode 653 / Easy 81

4 Methods **83**

5 Applications **85**
5.1 Example one 85
5.2 Example two 85

6 Final Words **87**

Chapter 1

Introduction

You can label chapter and section titles using `{#label}` after them, e.g., we can reference Chapter 1. If you do not manually label them, there will be automatic labels anyway, e.g., Chapter 4.

1.1 Glossary

1.1.1 Term 1

: This is a definition with two paragraphs. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aliquam hendrerit mi posuere lectus.

`Vestibulum enim wisi, viverra nec, fringilla in, laoreet vitae, risus.`

: Second definition for term 1, also wrapped in a paragraph because of the blank line preceding it.

1.1.2 Term 2

: This definition has a code block, a blockquote and a list.

```
    code block.
```

```
> block quote
> on two lines.
```

```
1.  first list item
2.  second list item
```

Reference: Term 1 1.1.1.and Term 2 1.1.2

Figures and tables with captions will be placed in `figure` and `table` environments, respectively.

```
par(mar = c(4, 4, .1, .1))
plot(pressure, type = 'b', pch = 19)
```

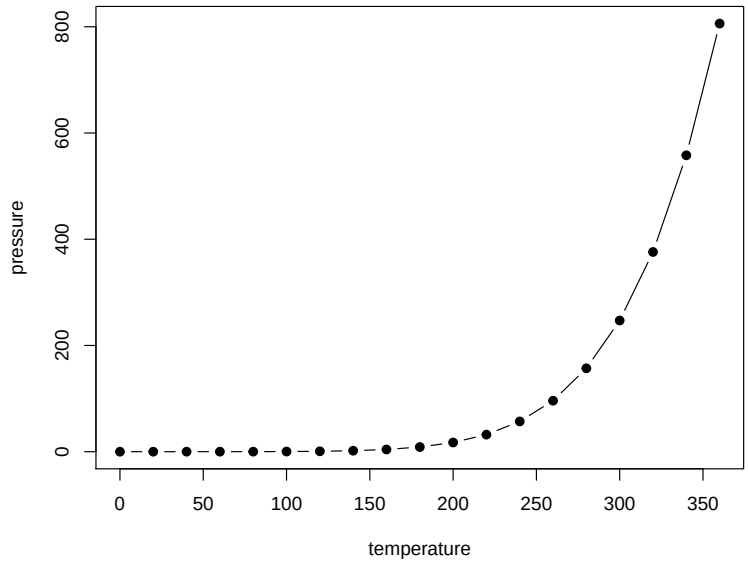

Figure 1.1: Here is a nice figure!

Reference a figure by its code chunk label with the `fig:` prefix, e.g., see Figure 1.1. Similarly, you can reference tables generated from `knitr::kable()`, e.g., see Table 1.1.

```
knitr::kable(
  head(iris, 20), caption = 'Here is a nice table!',
  booktabs = TRUE
)
```

You can write citations, too. For example, we are using the **bookdown** package (Xie, 2020) in this sample book, which was built on top of R Markdown and **knitr** (Joe, 2015).

Table 1.1: Here is a nice table!

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa
5.0	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa
5.4	3.7	1.5	0.2	setosa
4.8	3.4	1.6	0.2	setosa
4.8	3.0	1.4	0.1	setosa
4.3	3.0	1.1	0.1	setosa
5.8	4.0	1.2	0.2	setosa
5.7	4.4	1.5	0.4	setosa
5.4	3.9	1.3	0.4	setosa
5.1	3.5	1.4	0.3	setosa
5.7	3.8	1.7	0.3	setosa
5.1	3.8	1.5	0.3	setosa

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) {
    HashSet<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

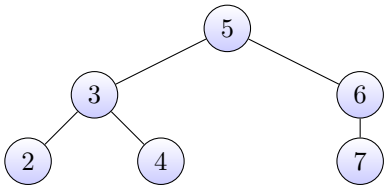

Figure 2.1: Some caption.

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

{recursive}

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}

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

{recursive}

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}, {memo}

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:

1. The current[i] element has been visited.
2. 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}, {memo}

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}, {recursive}

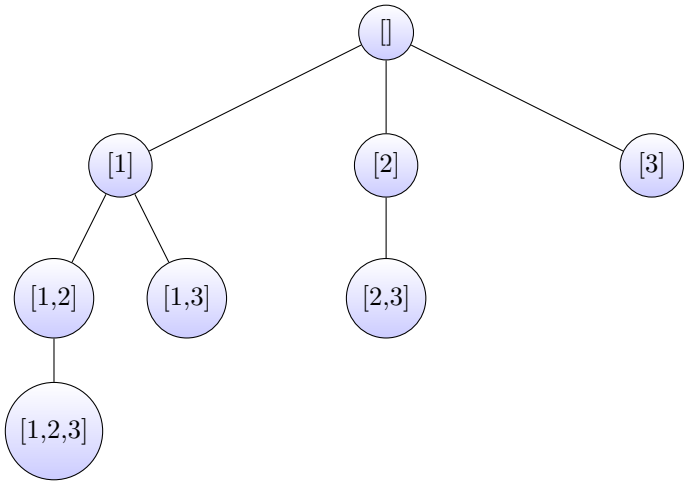

Figure 2.2: Some caption.

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}, {recursive}

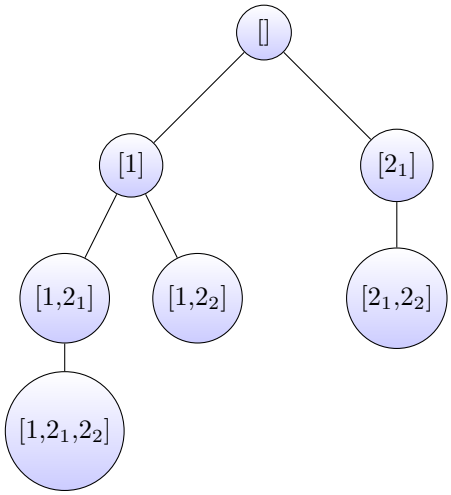

Figure 2.3: Some caption.

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

1. We first gather the frequency of integer by count with a HashMap
2. We sort the frequency map by comparing by value() in descending order and sort them in $List < Entry < Number, Frequency >>$.
3. 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

```
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, 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

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

(a) Case 2: [1, 0, 0, ..., 0, 1] Another 1 follows and makes it a block of 0s with 1s on both ends. Number of iteration needed to flip the 0s become $num_zeo/2$ if number is even otherwise $(num_zero + 1)/2$

(b) 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.

3. 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}

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}

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}

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 longest valid pa
        *
        * 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 startIdx 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}

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}

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

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

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

2.31.3.2 Analysis

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

2.31.3.3 Algorithm

{dp}

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}

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]

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}

2.33.4 Java Code

```
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}

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}

2.35.4 Java Code

```java
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}

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

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

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

```
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:

1. In the base case, where array contains only one element, the function returns the correct value.
2. When array contains more than 1 element, we have to start with either array[0] or array[n-1]. The code computes for each of these cases with recursion, and returns the maximum.

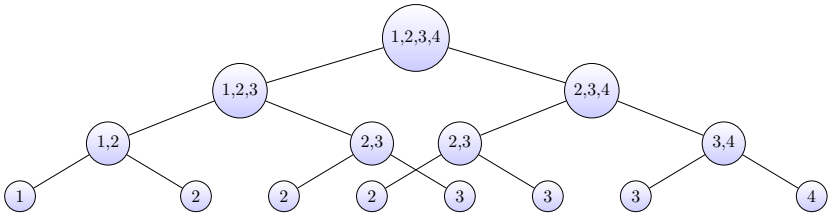

Figure 2.4: Some caption.

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

{recursive}

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

{recursive}, {memo}

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}, {table}

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

Literature

Here is a review of existing methods.

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

3.1.1 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.

3.1.2 Example

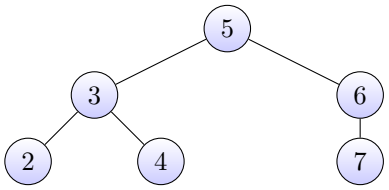

Figure 3.1: Some caption.

\Target = 9, Output = True.

3.1.3 Solution}

3.1.3.1 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.

3.1.3.2 Analysis

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

3.1.3.3 Algorithm

recursive

3.1.4 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
    }

}
```

Chapter 4

Methods

We describe our methods in this chapter.

Chapter 5

Applications

Some *significant* applications are demonstrated in this chapter.

5.1 Example one

5.2 Example two

Chapter 6

Final Words

We have finished a nice book.

Bibliography

Joe, A. (2015). *Hello World*. 2nd edition. ISBN.

Xie, Y. (2020). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.18.