Step 16 : Dynamic programming[Patterns and Problems]

1.1 Introduction to DP

🔰 Introduction to Dynamic Programming (DP)

Definition:
Dynamic Programming is an optimization technique used to solve problems by storing the results of subproblems (overlapping subproblems) to avoid redundant calculations.

✌️ Two Main Approaches in DP

Approach	Also Called	Direction	Idea
Memoization	Top-Down	Recursion + Caching	Store results during recursion
Tabulation	Bottom-Up	Iterative	Solve smaller subproblems first

---

📚 Pre-requisite

- **Recursion understanding** is essential before diving into DP.

🌀 Fibonacci Problem (Classic DP Example)

Fibonacci Series:
0, 1, 1, 2, 3, 5, 8, 13, ...
Formula: f(n) = f(n-1) + f(n-2) with base cases f(0)=0, f(1)=1

✅ Part 1: Memoization (Top-Down)

Steps:

1. Create a dp[n+1] array and initialize with -1
2. Before recursive calls, check: if(dp[n] != -1) return dp[n]
3. If not calculated yet, do the recursive call and store in dp[n]

java

```
static int f(int n, int[] dp){
```

```
    if(n <= 1) return n;
    if(dp[n] != -1) return dp[n];
    return dp[n] = f(n-1, dp) + f(n-2, dp);
}
```

Main Function:

```
java
CopyEdit
int n = 5;
int dp[] = new int[n + 1];
Arrays.fill(dp, -1);
System.out.println(f(n, dp));  // Output: 5
```

🕒 **Time Complexity:** O(N)
🧠 **Space Complexity:** O(N) — due to recursion + dp array

✅ **Part 2: Tabulation (Bottom-Up)**

Steps:

1. Create dp[n+1] array
2. Set base cases dp[0] = 0, dp[1] = 1
3. Loop from 2 to n, use dp[i] = dp[i-1] + dp[i-2]

```
java
CopyEdit
int n = 5;
int dp[] = new int[n + 1];
dp[0] = 0;
dp[1] = 1;

for (int i = 2; i <= n; i++) {
    dp[i] = dp[i-1] + dp[i-2];
```

}

System.out.println(dp[n]); // Output: 5

🕒 **Time Complexity:** O(N)
🧠 **Space Complexity:** O(N)

✅ **Part 3: Space Optimization**

Observation: We only need last 2 values ⇒ No need for full array.

java

CopyEdit

int n = 5;

int prev2 = 0, prev = 1;

for (int i = 2; i <= n; i++) {

  int cur_i = prev2 + prev;

  prev2 = prev;

  prev = cur_i;

}

System.out.println(prev); // Output: 5

🕒 **Time Complexity:** O(N)
🧠 **Space Complexity:** O(1)

✅ **Final Comparison Table**

Approach	Time Complexity	Space Complexity	Style
Memoization	O(N)	O(N)	Recursive
Tabulation	O(N)	O(N)	Iterative
Space Optimum	O(N)	O(1)	Iterative

📌 **Key Takeaways**

- Use **memoization** when converting a recursive solution.
- Use **tabulation** when you can think iteratively.
- Use **space optimization** when only last few values are needed.

2.1 70. Climbing Stairs

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

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

Example 1:

Input: n = 2

Output: 2

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

1. 1 step + 1 step

2. 2 steps

Example 2:

Input: n = 3

Output: 3

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

1. 1 step + 1 step + 1 step

2. 1 step + 2 steps

3. 2 steps + 1 step

Constraints:

- 1 <= n <= 45

2.2 **403. Frog Jump** [Leetcode]

A frog is crossing a river. The river is divided into some number of units, and at each unit, there may or may not exist a stone. The frog can jump on a stone, but it must not jump into the water.

Given a list of stones positions (in units) in sorted **ascending order**, determine if the frog can cross the river by landing on the last stone. Initially, the frog is on the first stone and assumes the first jump must be 1 unit.

If the frog's last jump was k units, its next jump must be either k - 1, k, or k + 1 units. The frog can only jump in the forward direction.

Example 1:

Input: stones = [0,1,3,5,6,8,12,17]

Output: true

Explanation: The frog can jump to the last stone by jumping 1 unit to the 2nd stone, then 2 units to the 3rd stone, then 2 units to the 4th stone, then 3 units to the 6th stone, 4 units to the 7th stone, and 5 units to the 8th stone.

Example 2:

Input: stones = [0,1,2,3,4,8,9,11]

Output: false

Explanation: There is no way to jump to the last stone as the gap between the 5th and 6th stone is too large.

Constraints:

- 2 <= stones.length <= 2000
- 0 <= stones[i] <= 2^{31} - 1
- stones[0] == 0
- stones is sorted in a strictly increasing order.

2.2.1 **Frog Jump [GFG]**

Difficulty: **Medium**Accuracy: **49.55%**Submissions: **136K+**Points: **4**Average Time: **15m**

Given an integer array **height[]** where **height[i]** represents the height of the **i-th** stair, a frog starts from the **first stair** and wants to reach the **top**. From any stair **i**, the frog has two options: it can either jump to the **(i+1)th** stair or the **(i+2)th** stair. The cost of a jump is the absolute difference in height between the two stairs. Determine the minimum total cost required for the frog to reach the top.

Example:

Input: heights[] = [20, 30, 40, 20]
Output: 20
Explanation: Minimum cost is incurred when the frog jumps from stair 0 to 1 then 1 to 3:
jump from stair 0 to 1: cost = |30 - 20| = 10
jump from stair 1 to 3: cost = |20-30| = 10
Total Cost = 10 + 10 = 20

Input: heights[] = [30, 20, 50, 10, 40]
Output: 30
Explanation: Minimum cost will be incurred when frog jumps from stair 0 to 2 then 2 to 4:
jump from stair 0 to 2: cost = |50 - 30| = 20
jump from stair 2 to 4: cost = |40-50| = 10
Total Cost = 20 + 10 = 30

Constraints:

- 1 <= height.size() <= 10^5
- 0 <= height[i]<=10^4

2.3 2498. Frog Jump II

You are given a **0-indexed** integer array stones sorted in **strictly increasing order** representing the positions of stones in a river.

A frog, initially on the first stone, wants to travel to the last stone and then return to the first stone. However, it can jump to any stone **at most once**.

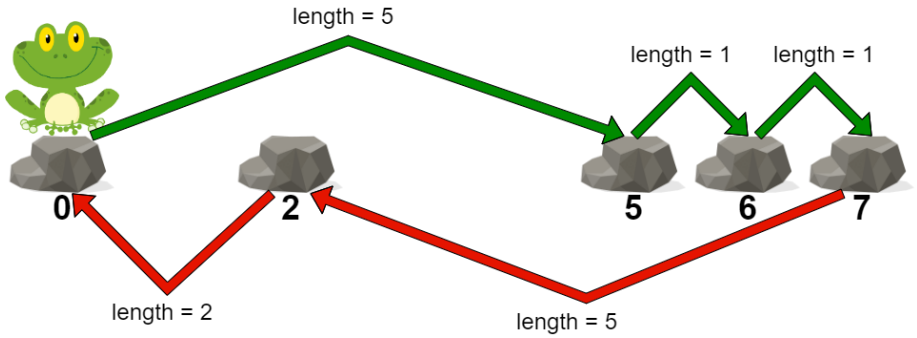
The **length** of a jump is the absolute difference between the position of the stone the frog is currently on and the position of the stone to which the frog jumps.

- More formally, if the frog is at stones[i] and is jumping to stones[j], the length of the jump is |stones[i] - stones[j]|.

The **cost** of a path is the **maximum length of a jump** among all jumps in the path.

Return *the **minimum** cost of a path for the frog*.

Example 1:



Input: stones = [0,2,5,6,7]

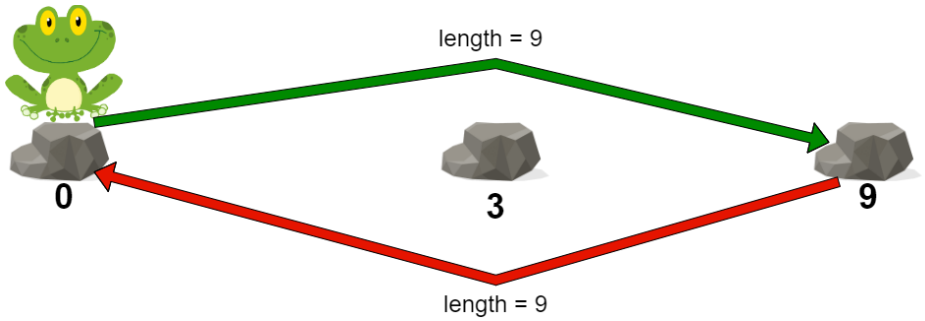
Output: 5

Explanation: The above figure represents one of the optimal paths the frog can take.

The cost of this path is 5, which is the maximum length of a jump.

Since it is not possible to achieve a cost of less than 5, we return it.

Example 2:



Input: stones = [0,3,9]

Output: 9

Explanation:

The frog can jump directly to the last stone and come back to the first stone.

In this case, the length of each jump will be 9. The cost for the path will be max(9, 9) = 9.

It can be shown that this is the minimum achievable cost.

Constraints:

- 2 <= stones.length <= 10^5
- 0 <= stones[i] <= 10^9
- stones[0] == 0
- stones is sorted in a strictly increasing order.

2.4 198. House Robber

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

Given an integer array nums representing the amount of money of each house, return *the maximum amount of money you can rob tonight* ***without alerting the police***.

Example 1:

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

Output: 4

Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).

Total amount you can rob = 1 + 3 = 4.

Example 2:

Input: nums = [2,7,9,3,1]

Output: 12

Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1).

Total amount you can rob = 2 + 9 + 1 = 12.

Constraints:

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

2.5 **213. House Robber II**

Medium

Topics

Companies

Hint

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. All houses at this place are **arranged in a circle.** That means the first house is the neighbor of the last one. Meanwhile, adjacent houses have a security system connected, and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given an integer array nums representing the amount of money of each house, return *the maximum amount of money you can rob tonight* ***without alerting the police***.

Example 1:

Input: nums = [2,3,2]

Output: 3

Explanation: You cannot rob house 1 (money = 2) and then rob house 3 (money = 2), because they are adjacent houses.

Example 2:

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

Output: 4

Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).

Total amount you can rob = 1 + 3 = 4.

Example 3:

Input: nums = [1,2,3]

Output: 3

Constraints:

- 1 <= nums.length <= 100

3.1 **Geek's Training**

Difficulty: **Medium**Accuracy: **49.98%**Submissions: **116K+**Points: **4**Average Time: **20m**

Geek is going for a training program for n days. He can perform any of these activities: **Running**, **Fighting**, and **Learning** Practice. Each activity has some point on each day. As Geek wants to improve all his skills, he can't do the same activity on two consecutive days. Given a 2D array arr[][] of size n where arr[i][0], arr[i][1], and arr[i][2] represent the merit points for **Running**, **Fighting**, and **Learning** on the i-th day, determine the maximum total merit points Geek can achieve .

Example:

Input: arr[]= [[1, 2, 5], [3, 1, 1], [3, 3, 3]]

Output: 11

Explanation: Geek will learn a new move and earn 5 point then on second day he will do running and earn 3 point and on third day he will do fighting and earn 3 points so, maximum merit point will be 11.

Input: arr[]= [[1, 1, 1], [2, 2, 2], [3, 3, 3]]

Output: 6

Explanation: Geek can perform any activity each day while adhering to the constraints, in order to maximize his total merit points as 6.

Input: arr[]= [[4, 2, 6]]

Output: 6

Explanation: Geek will learn a new move to make his merit points as 6.

Constraint:
$1 <= n <= 10^5$
$1 <= arr[i][j] <= 100$

3.2 62. Unique Paths

There is a robot on an m x n grid. The robot is initially located at the **top-left corner** (i.e., grid[0][0]). The robot tries to move to the **bottom-right corner** (i.e., grid[m - 1][n - 1]). The robot can only move either down or right at any point in time.

Given the two integers m and n, return *the number of possible unique paths that the robot can take to reach the bottom-right corner*.

The test cases are generated so that the answer will be less than or equal to $2 * 10^9$.

Example 1:

Input: m = 3, n = 7

Output: 28

Example 2:

Input: m = 3, n = 2

Output: 3

Explanation: From the top-left corner, there are a total of 3 ways to reach the bottom-right corner:

1. Right -> Down -> Down

2. Down -> Down -> Right

3. Down -> Right -> Down

Constraints:

- 1 <= m, n <= 100

3.3 **63. Unique Paths II**

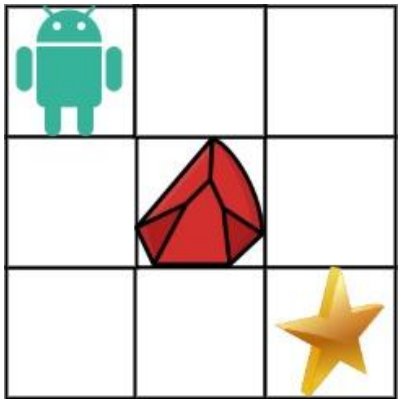
You are given an m x n integer array grid. There is a robot initially located at the **top-left corner** (i.e., grid[0][0]). The robot tries to move to the **bottom-right corner** (i.e., grid[m - 1][n - 1]). The robot can only move either down or right at any point in time.

An obstacle and space are marked as 1 or 0 respectively in grid. A path that the robot takes cannot include **any** square that is an obstacle.

Return *the number of possible unique paths that the robot can take to reach the bottom-right corner*.

The testcases are generated so that the answer will be less than or equal to $2 * 10^9$.

Example 1:



Input: obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]

Output: 2

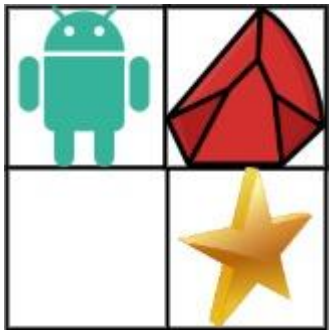
Explanation: There is one obstacle in the middle of the 3x3 grid above.

There are two ways to reach the bottom-right corner:

1. Right -> Right -> Down -> Down

2. Down -> Down -> Right -> Right

Example 2:



Input: obstacleGrid = [[0,1],[0,0]]

Output: 1

Constraints:

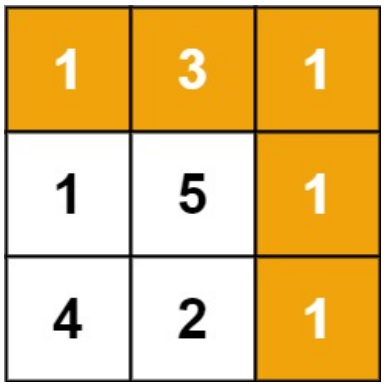
- m == obstacleGrid.length
- n == obstacleGrid[i].length
- 1 <= m, n <= 100
- obstacleGrid[i][j] is 0 or 1.

3.4 64. Minimum Path Sum

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.

Note: You can only move either down or right at any point in time.

Example 1:



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

Output: 7

Explanation: Because the path 1 → 3 → 1 → 1 → 1 minimizes the sum.

Example 2:

Input: grid = [[1,2,3],[4,5,6]]

Output: 12

Constraints:

- m == grid.length
- n == grid[i].length
- 1 <= m, n <= 200
- 0 <= grid[i][j] <= 200

3.5 **120. Triangle**

Given a triangle array, return *the minimum path sum from top to bottom*.

For each step, you may move to an adjacent number of the row below. More formally, if you are on index i on the current row, you may move to either index i or index i + 1 on the next row.

Example 1:

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

Output: 11

Explanation: The triangle looks like:

2

3 4

6 5 7

4 1 8 3

The minimum path sum from top to bottom is 2 + 3 + 5 + 1 = 11 (underlined above).

Example 2:

Input: triangle = [[-10]]

Output: -10

Constraints:

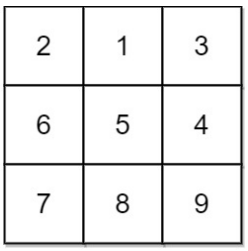
- 1 <= triangle.length <= 200
- triangle[0].length == 1
- triangle[i].length == triangle[i - 1].length + 1
- -10^4 <= triangle[i][j] <= 10^4

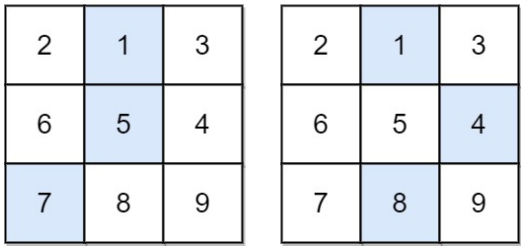
3.6 **931. Minimum Falling Path Sum**

Given an n x n array of integers matrix, return *the* ***minimum sum*** *of any* ***falling path*** *through* matrix.

A **falling path** starts at any element in the first row and chooses the element in the next row that is either directly below or diagonally left/right. Specifically, the next element from position (row, col) will be (row + 1, col - 1), (row + 1, col), or (row + 1, col + 1).

Example 1:



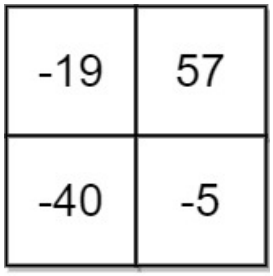


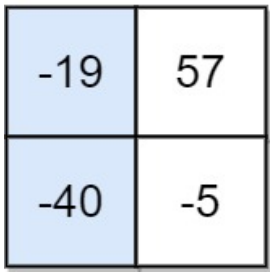
Input: matrix = [[2,1,3],[6,5,4],[7,8,9]]

Output: 13

Explanation: There are two falling paths with a minimum sum as shown.

Example 2:





Input: matrix = [[-19,57],[-40,-5]]

Output: -59

Explanation: The falling path with a minimum sum is shown.

Constraints:

- n == matrix.length == matrix[i].length
- 1 <= n <= 100
- -100 <= matrix[i][j] <= 100

3.7 . **Problem statement**

Send feedback

Ninja has a 'GRID' of size 'R' X 'C'. Each cell of the grid contains some chocolates. Ninja has two friends Alice and Bob, and he wants to collect as many chocolates as possible with the help of his friends.

Initially, Alice is in the top-left position i.e. (0, 0), and Bob is in the top-right place i.e. (0, ‘C’ - 1) in the grid. Each of them can move from their current cell to the cells just below them. When anyone passes from any cell, he will pick all chocolates in it, and then the number of chocolates in that cell will become zero. If both stay in the same cell, only one of them will pick the chocolates in it.

If Alice or Bob is at (i, j) then they can move to (i + 1, j), (i + 1, j - 1) or (i + 1, j + 1). They will always stay inside the ‘GRID’.

Your task is to find the maximum number of chocolates Ninja can collect with the help of his friends by following the above rules.

Example:

Input: ‘R’ = 3, ‘C’ = 4

‘GRID’ = [[2, 3, 1, 2], [3, 4, 2, 2], [5, 6, 3, 5]]

Output: 21

Initially Alice is at the position (0,0) he can follow the path (0,0) -> (1,1) -> (2,1) and will collect 2 + 4 + 6 = 12 chocolates.

Initially Bob is at the position (0, 3) and he can follow the path (0, 3) -> (1,3) -> (2, 3) and will colllect 2 + 2 + 5 = 9 chocolates.

Hence the total number of chocolates collected will be 12 + 9 = 21. there is no other possible way to collect a greater number of chocolates than 21.

Detailed explanation (Input/output format, Notes, Images)

Constraints :

1 <= ‘T’ <= 10

2 <= 'R', 'C' <= 50

0 <= 'GRID[i][j]'<= 10^2

Time Limit: 1sec

Sample Input 1 :

2

3 4

2 3 1 2

3 4 2 2

5 6 3 5

2 2

1 1

1 2

Sample Output 1 :

21

5

Explanation Of Sample Input 1 :

For the first test case, Initially Alice is at the position (0, 0) he can follow the path (0, 0) -> (1, 1) -> (2, 1) and will collect 2 + 4 + 6 = 12 chocolates.

Initially Bob is at the position (0, 3) and he can follow the path (0, 3) -> (1, 3) -> (2, 3) and will collect 2 + 2 + 5 = 9 chocolates.

Hence the total number of chocolates collected will be 12 + 9 = 21.

For the second test case, Alice will follow the path (0, 0) -> (1, 0) and Bob will follow the path (0, 1) -> (1, 1). total number of chocolates collected will be 1 + 1 + 1 + 2 = 5

Sample Input 2 :

2

2 2

3 7

7 6

3 2

4 5

3 7

4 2

Sample Output 2 :

23

25

4.1 **494. Target Sum** [Leetcode]

You are given an integer array nums and an integer target.

You want to build an **expression** out of nums by adding one of the symbols '+' and '-' before each integer in nums and then concatenate all the integers.

- For example, if nums = [2, 1], you can add a '+' before 2 and a '-' before 1 and concatenate them to build the expression "+2-1".

Return the number of different **expressions** that you can build, which evaluates to target.

Example 1:

Input: nums = [1,1,1,1,1], target = 3

Output: 5

Explanation: There are 5 ways to assign symbols to make the sum of nums be target 3.

-1 + 1 + 1 + 1 + 1 = 3

+1 - 1 + 1 + 1 + 1 = 3

+1 + 1 - 1 + 1 + 1 = 3

+1 + 1 + 1 - 1 + 1 = 3

+1 + 1 + 1 + 1 - 1 = 3

Example 2:

Input: nums = [1], target = 1

Output: 1

Constraints:

- 1 <= nums.length <= 20
- 0 <= nums[i] <= 1000
- 0 <= sum(nums[i]) <= 1000
- -1000 <= target <= 1000

4.1.1 **Subset Sum Problem**

Difficulty: **Medium**Accuracy: **32.0%**Submissions: **356K+**Points: **4**

Given an array of positive integers **arr[]** and a value **sum**, determine if there is a subset of **arr[]** with sum equal to given **sum**.

Examples:

Input: arr[] = [3, 34, 4, 12, 5, 2], sum = 9
Output: true

Explanation: Here there exists a subset with target sum = 9, 4+3+2 = 9.

Input: arr[] = [3, 34, 4, 12, 5, 2], sum = 30

Output: false

Explanation: There is no subset with target sum 30.

Input: arr[] = [1, 2, 3], sum = 6

Output: true
Explanation: The entire array can be taken as a subset, giving 1 + 2 + 3 = 6.

Constraints:
1 <= arr.size() <= 200
1<= arr[i] <= 200
1<= sum <= 10^4

4.2 416. Partition Equal Subset Sum

Given an integer array nums, return true *if you can partition the array into two subsets such that the sum of the elements in both subsets is equal or* false *otherwise*.

Example 1:

Input: nums = [1,5,11,5]

Output: true

Explanation: The array can be partitioned as [1, 5, 5] and [11].

Example 2:

Input: nums = [1,2,3,5]

Output: false

Explanation: The array cannot be partitioned into equal sum subsets.

Constraints:

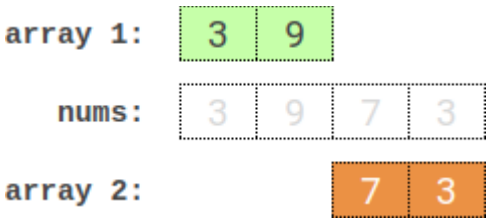
- 1 <= nums.length <= 200
- 1 <= nums[i] <= 100

4.3 [2035. Partition Array Into Two Arrays to Minimize Sum Difference]

You are given an integer array nums of 2 * n integers. You need to partition nums into **two** arrays of length n to **minimize the absolute difference** of the **sums** of the arrays. To partition nums, put each element of nums into **one** of the two arrays.

Return *the **minimum** possible absolute difference*.

Example 1:



Input: nums = [3,9,7,3]

Output: 2

Explanation: One optimal partition is: [3,9] and [7,3].

The absolute difference between the sums of the arrays is abs((3 + 9) - (7 + 3)) = 2.

Example 2:

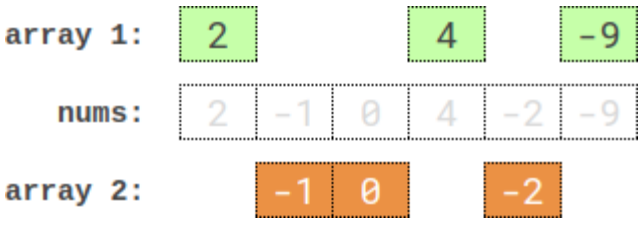
Input: nums = [-36,36]

Output: 72

Explanation: One optimal partition is: [-36] and [36].

The absolute difference between the sums of the arrays is abs((-36) - (36)) = 72.

Example 3:



Input: nums = [2,-1,0,4,-2,-9]

Output: 0

Explanation: One optimal partition is: [2,4,-9] and [-1,0,-2].

The absolute difference between the sums of the arrays is abs((2 + 4 + -9) - (-1 + 0 + -2)) = 0.

Constraints:

- 1 <= n <= 15
- nums.length == 2 * n
- -10^7 <= nums[i] <= 10^7

4.4 **Perfect Sum Problem**

Difficulty: **Medium**Accuracy: **20.58%**Submissions: **525K+**Points: **4**

Given an array **arr** of non-negative integers and an integer **target**, the task is to count all subsets of the array whose sum is equal to the given target.

Examples:

Input: arr[] = [5, 2, 3, 10, 6, 8], target = 10

Output: 3

Explanation: The subsets {5, 2, 3}, {2, 8}, and {10} sum up to the target 10.

Input: arr[] = [2, 5, 1, 4, 3], target = 10

Output: 3

Explanation: The subsets {2, 1, 4, 3}, {5, 1, 4}, and {2, 5, 3} sum up to the target 10.

Input: arr[] = [5, 7, 8], target = 3
Output: 0

Explanation: There are no subsets of the array that sum up to the target 3.

Input: arr[] = [35, 2, 8, 22], target = 0
Output: 1

Explanation: The empty subset is the only subset with a sum of 0.

Constraints:
$1 \leq arr.size() \leq 10^3$
$0 \leq arr[i] \leq 10^3$
$0 \leq target \leq 10^3$

4.5 **Partitions with Given Difference**

Difficulty: **Medium**Accuracy: **36.76%**Submissions: **181K+**Points: **4**Average Time: **20m**

Given an array **arr[]**, partition it into two subsets(possibly empty) such that each element must belong to only one subset. Let the sum of the elements of these two subsets be **sum1** and **sum2**. Given a difference **d**, count the number of partitions in which **sum1** is greater than or equal to **sum2** and the difference between **sum1** and **sum2** is equal to **d**.

Examples :

Input: arr[] = [5, 2, 6, 4], d = 3

Output: 1

Explanation: There is only one possible partition of this array. Partition : {6, 4}, {5, 2}. The subset difference between subset sum is: (6 + 4) - (5 + 2) = 3.

Input: arr[] = [1, 1, 1, 1], d = 0
Output: 6
Explanation: We can choose two 1's from indices {0,1}, {0,2}, {0,3}, {1,2}, {1,3}, {2,3} and put them in sum1 and remaning two 1's in sum2.
Thus there are total 6 ways for partition the array arr.

Input: arr[] = [1, 2, 1, 0, 1, 3, 3], d = 11
Output: 2

Constraint:
1 <= arr.size() <= 50
0 <= d <= 50
0 <= arr[i] <= 6

4.6 455. Assign Cookies

Assume you are an awesome parent and want to give your children some cookies. But, you should give each child at most one cookie.

Each child i has a greed factor g[i], which is the minimum size of a cookie that the child will be content with; and each cookie j has a size s[j]. If s[j] >= g[i], we can assign the cookie j to the child i, and the child i will be content. Your goal is to maximize the number of your content children and output the maximum number.

Example 1:

Input: g = [1,2,3], s = [1,1]

Output: 1

Explanation: You have 3 children and 2 cookies. The greed factors of 3 children are 1, 2, 3.

And even though you have 2 cookies, since their size is both 1, you could only make the child whose greed factor is 1 content.

You need to output 1.

Example 2:

Input: g = [1,2], s = [1,2,3]

Output: 2

Explanation: You have 2 children and 3 cookies. The greed factors of 2 children are 1, 2.

You have 3 cookies and their sizes are big enough to gratify all of the children,

You need to output 2.

Constraints:

- $1 <= g.length <= 3 * 10^4$
- $0 <= s.length <= 3 * 10^4$
- $1 <= g[i], s[j] <= 2^{31} - 1$

Note: This question is the same as 2410: Maximum Matching of Players With Trainers.

4.7 322. Coin Change

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

Return *the fewest number of coins that you need to make up that amount*. If that amount of money cannot be made up by any combination of the coins, return -1.

You may assume that you have an infinite number of each kind of coin.

Example 1:

Input: coins = [1,2,5], amount = 11

Output: 3

Explanation: 11 = 5 + 5 + 1

Example 2:

Input: coins = [2], amount = 3

Output: -1

Example 3:

Input: coins = [1], amount = 0

Output: 0

Constraints:

- 1 <= coins.length <= 12
- 1 <= coins[i] <= 2^{31} - 1
- 0 <= amount <= 10^4

4.9 **518. Coin Change II**

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

Return *the number of combinations that make up that amount*. If that amount of money cannot be made up by any combination of the coins, return 0.

You may assume that you have an infinite number of each kind of coin.

The answer is **guaranteed** to fit into a signed **32-bit** integer.

Example 1:

Input: amount = 5, coins = [1,2,5]

Output: 4

Explanation: there are four ways to make up the amount:

5=5

5=2+2+1

5=2+1+1+1

5=1+1+1+1+1

Example 2:

Input: amount = 3, coins = [2]

Output: 0

Explanation: the amount of 3 cannot be made up just with coins of 2.

Example 3:

Input: amount = 10, coins = [10]

Output: 1

Constraints:

- 1 <= coins.length <= 300
- 1 <= coins[i] <= 5000
- All the values of coins are **unique**.
- 0 <= amount <= 5000

4.10 **Knapsack with Duplicate Items**

Difficulty: **Medium**Accuracy: **52.13%**Submissions: **189K+**Points: **4**

Given a set of items, each with a weight and a value, represented by the array **wt** and **val** respectively. Also, a knapsack with a weight limit **capacity**.
The task is to fill the knapsack in such a way that we can get the maximum profit. Return the maximum profit.
Note: Each item can be taken any number of times.

Examples:

Input: val = [1, 1], wt = [2, 1], capacity = 3

Output: 3

Explanation: The optimal choice is to pick the 2nd element 3 times.

Input: val[] = [6, 1, 7, 7], wt[] = [1, 3, 4, 5], capacity = 8

Output: 48

Explanation: The optimal choice is to pick the 1st element 8 times.

Input: val[] = [6, 8, 7, 100], wt[] = [2, 3, 4, 5], capacity = 1

Output: 0

Explanation: We can't pick any element .hence, total profit is 0.

Constraints:
1 <= val.size() = wt.size() <= 1000
1 <= capacity <= 1000
1 <= val[i], wt[i] <= 100

4.11 **Rod Cutting**

Difficulty: **Medium**Accuracy: **60.66%**Submissions: **170K+**Points: **4**Average Time: **20m**

Given a rod of length **n** inches and an array **price[]**, where **price[i]** denotes the value of a piece of length **i**. Your task is to determine the **maximum** value obtainable by **cutting up** the rod and selling the **pieces**.

Note: n = size of price, and price[] is **1-indexed** array.

Example:

Input: price[] = [1, 5, 8, 9, 10, 17, 17, 20]
Output: 22
Explanation: The maximum obtainable value is 22 by cutting in two pieces of lengths 2 and 6, i.e., 5 + 17 = 22.

Input: price[] = [3, 5, 8, 9, 10, 17, 17, 20]
Output: 24
Explanation: The maximum obtainable value is 24 by cutting the rod into 8 pieces of length 1, i.e, 8*price[1] = 8*3 = 24.

Input: price[] = [3]
Output: 3
Explanation: There is only 1 way to pick a piece of length 1.

Constraints:
$1 \leq price.size() \leq 10^3$
$1 \leq price[i] \leq 10^6$

5.1 1143. Longest Common Subsequence

Given two strings text1 and text2, return *the length of their longest* ***common subsequence***. If there is no **common subsequence**, return 0.

A **subsequence** of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

- For example, "ace" is a subsequence of "abcde".

A **common subsequence** of two strings is a subsequence that is common to both strings.

Example 1:

Input: text1 = "abcde", text2 = "ace"

Output: 3

Explanation: The longest common subsequence is "ace" and its length is 3.

Example 2:

Input: text1 = "abc", text2 = "abc"

Output: 3

Explanation: The longest common subsequence is "abc" and its length is 3.

Example 3:

Input: text1 = "abc", text2 = "def"

Output: 0

Explanation: There is no such common subsequence, so the result is 0.

Constraints:

- 1 <= text1.length, text2.length <= 1000
- text1 and text2 consist of only lowercase English characters.

5.2 **Print all LCS sequences**

Difficulty: **Hard**Accuracy: **30.64%**Submissions: **56K+**Points: **8**

You are given two strings **s1** and **s2**. Your task is to print all **distinct** longest common subsequences in lexicographical order.

Note: A subsequence is derived from another string by deleting some or none of the elements without changing the order of the remaining elements.

Examples:

Input: s1 = "abaaa", s2 = "baabaca"

Output: ["aaaa", "abaa", "baaa"]
Explanation: Length of lcs is 4, in lexicographical order they are "aaaa", "abaa", "baaa".

Input: s1 = "aaa", s2 = "a"

Output: ["a"]
Explanation: Length of lcs is 1 and it is "a".

Constraints:
1 ≤ s1.size(), s2.size() ≤ 50

5.3 **Longest Common Substring**

Difficulty: **Medium**Accuracy: **42.69%**Submissions: **280K+**Points: **4**

You are given two strings **s1** and **s2**. Your task is to find the length of the **longest common substring** among the given strings.

Examples:

Input: s1 = "ABCDGH", s2 = "ACDGHR"

Output: 4

Explanation: The longest common substring is "CDGH" with a length of 4.

Input: s1 = "abc", s2 = "acb"

Output: 1

Explanation: The longest common substrings are "a", "b", "c" all having length 1.

Input: s1 = "YZ", s2 = "yz"

Output: 0

Constraints:
1 <= s1.size(), s2.size() <= 10^3
Both strings may contain upper and lower case alphabets.

5.4 516. Longest Palindromic Subsequence

Given a string s, find *the longest palindromic* ***subsequence****'s length in* s.

A **subsequence** is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

Example 1:

Input: s = "bbbab"

Output: 4

Explanation: One possible longest palindromic subsequence is "bbbb".

Example 2:

Input: s = "cbbd"

Output: 2

Explanation: One possible longest palindromic subsequence is "bb".

Constraints:

- 1 <= s.length <= 1000
- s consists only of lowercase English letters.

5.5 **1312. Minimum Insertion Steps to Make a String Palindrome**

Hard

Topics

Companies

Hint

Given a string s. In one step you can insert any character at any index of the string.

Return *the minimum number of steps* to make s palindrome.

A **Palindrome String** is one that reads the same backward as well as forward.

Example 1:

Input: s = "zzazz"

Output: 0

Explanation: The string "zzazz" is already palindrome we do not need any insertions.

Example 2:

Input: s = "mbadm"

Output: 2

Explanation: String can be "mbdadbm" or "mdbabdm".

Example 3:

Input: s = "leetcode"

Output: 5

Explanation: Inserting 5 characters the string becomes "leetcodocteel".

Constraints:

- 1 <= s.length <= 500
- s consists of lowercase English letters.

5.6 **583. Delete Operation for Two Strings**

Given two strings word1 and word2, return *the minimum number of* ***steps*** *required to make* word1 *and* word2 *the same*.

In one **step**, you can delete exactly one character in either string.

Example 1:

Input: word1 = "sea", word2 = "eat"

Output: 2

Explanation: You need one step to make "sea" to "ea" and another step to make "eat" to "ea".

Example 2:

Input: word1 = "leetcode", word2 = "etco"

Output: 4

Constraints:

- 1 <= word1.length, word2.length <= 500
- word1 and word2 consist of only lowercase English letters.

5.7 1092. Shortest Common Supersequence

Given two strings str1 and str2, return *the shortest string that has both* str1 *and* str2 *as* ***subsequences***. If there are multiple valid strings, return **any** of them.

A string s is a **subsequence** of string t if deleting some number of characters from t (possibly 0) results in the string s.

Example 1:

Input: str1 = "abac", str2 = "cab"

Output: "cabac"

Explanation:

str1 = "abac" is a subsequence of "cabac" because we can delete the first "c".

str2 = "cab" is a subsequence of "cabac" because we can delete the last "ac".

The answer provided is the shortest such string that satisfies these properties.

Example 2:

Input: str1 = "aaaaaaaa", str2 = "aaaaaaaa"

Output: "aaaaaaaa"

Constraints:

- 1 <= str1.length, str2.length <= 1000
- str1 and str2 consist of lowercase English letters.

5.8 115. Distinct Subsequences

Given two strings s and t, return *the number of distinct* ***subsequences*** *of* s *which equals* t.

The test cases are generated so that the answer fits on a 32-bit signed integer.

Example 1:

Input: s = "rabbbit", t = "rabbit"

Output: 3

Explanation:

As shown below, there are 3 ways you can generate "rabbit" from s.

rabbb**it**

rab**bbit**

rabb**bit**

Example 2:

Input: s = "babgbag", t = "bag"

Output: 5

Explanation:

As shown below, there are 5 ways you can generate "bag" from s.

bab**g**bag

babgba**g**

babgb**ag**

ba**b**gb**ag**

babg**bag**

Constraints:

- 1 <= s.length, t.length <= 1000
- s and t consist of English letters.

5.9 **72. Edit Distance**

Given two strings word1 and word2, return *the minimum number of operations required to convert word1 to word2*.

You have the following three operations permitted on a word:

- Insert a character
- Delete a character
- Replace a character

Example 1:

Input: word1 = "horse", word2 = "ros"

Output: 3

Explanation:

horse -> rorse (replace 'h' with 'r')

rorse -> rose (remove 'r')

rose -> ros (remove 'e')

Example 2:

Input: word1 = "intention", word2 = "execution"

Output: 5

Explanation:

intention -> inention (remove 't')

inention -> enention (replace 'i' with 'e')

enention -> exention (replace 'n' with 'x')

exention -> exection (replace 'n' with 'c')

exection -> execution (insert 'u')

Constraints:

- 0 <= word1.length, word2.length <= 500
- word1 and word2 consist of lowercase English letters.

5.10 44. Wildcard Matching

Given an input string (s) and a pattern (p), implement wildcard pattern matching with support for '?' and '*' where:

- '?' Matches any single character.
- '*' Matches any sequence of characters (including the empty sequence).

The matching should cover the **entire** input string (not partial).

Example 1:

Input: s = "aa", p = "a"

Output: false

Explanation: "a" does not match the entire string "aa".

Example 2:

Input: s = "aa", p = "*"

Output: true

Explanation: '*' matches any sequence.

Example 3:

Input: s = "cb", p = "?a"

Output: false

Explanation: '?' matches 'c', but the second letter is 'a', which does not match 'b'.

Constraints:

- 0 <= s.length, p.length <= 2000
- s contains only lowercase English letters.
- p contains only lowercase English letters, '?' or '*'.

6.1 121. Best Time to Buy and Sell Stock

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

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return 0.

Example 1:

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

Output: 5

Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.

Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Example 2:

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

Output: 0

Explanation: In this case, no transactions are done and the max profit = 0.

Constraints:

- 1 <= prices.length <= 10^5
- 0 <= prices[i] <= 10^4

6.2 122. Best Time to Buy and Sell Stock II

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

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

Find and return *the **maximum** profit you can achieve*.

Example 1:

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

Output: 7

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

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

Total profit is 4 + 3 = 7.

Example 2:

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

Output: 4

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

Total profit is 4.

Example 3:

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

Output: 0

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

Constraints:

- 1 <= prices.length <= $3 * 10^4$
- 0 <= prices[i] <= 10^4

6.3 23. Best Time to Buy and Sell Stock III

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

Find the maximum profit you can achieve. You may complete **at most two transactions**.

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

Example 1:

Input: prices = [3,3,5,0,0,3,1,4]

Output: 6

Explanation: Buy on day 4 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3.

Then buy on day 7 (price = 1) and sell on day 8 (price = 4), profit = 4-1 = 3.

Example 2:

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

Output: 4

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

Note that you cannot buy on day 1, buy on day 2 and sell them later, as you are engaging multiple transactions at the same time. You must sell before buying again.

Example 3:

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

Output: 0

Explanation: In this case, no transaction is done, i.e. max profit = 0.

Constraints:

- 1 <= prices.length <= 10^5
- 0 <= prices[i] <= 10^5

6.4 188. Best Time to Buy and Sell Stock IV

You are given an integer array prices where prices[i] is the price of a given stock on the i^{th} day, and an integer k.

Find the maximum profit you can achieve. You may complete at most k transactions: i.e. you may buy at most k times and sell at most k times.

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

Example 1:

Input: k = 2, prices = [2,4,1]

Output: 2

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

Example 2:

Input: k = 2, prices = [3,2,6,5,0,3]

Output: 7

Explanation: Buy on day 2 (price = 2) and sell on day 3 (price = 6), profit = 6-2 = 4. Then buy on day 5 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3.

Constraints:

- 1 <= k <= 100
- 1 <= prices.length <= 1000
- 0 <= prices[i] <= 1000

6.5 **309. Best Time to Buy and Sell Stock with Cooldown**

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

Find the maximum profit you can achieve. You may complete as many transactions as you like (i.e., buy one and sell one share of the stock multiple times) with the following restrictions:

- After you sell your stock, you cannot buy stock on the next day (i.e., cooldown one day).

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

Example 1:

Input: prices = [1,2,3,0,2]

Output: 3

Explanation: transactions = [buy, sell, cooldown, buy, sell]

Example 2:

Input: prices = [1]

Output: 0

Constraints:

- 1 <= prices.length <= 5000
- 0 <= prices[i] <= 1000

6.6 714. Best Time to Buy and Sell Stock with Transaction Fee

You are given an array prices where prices[i] is the price of a given stock on the i^{th} day, and an integer fee representing a transaction fee.

Find the maximum profit you can achieve. You may complete as many transactions as you like, but you need to pay the transaction fee for each transaction.

Note:

- You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).
- The transaction fee is only charged once for each stock purchase and sale.

Example 1:

Input: prices = [1,3,2,8,4,9], fee = 2

Output: 8

Explanation: The maximum profit can be achieved by:

- Buying at prices[0] = 1

- Selling at prices[3] = 8

- Buying at prices[4] = 4

- Selling at prices[5] = 9

The total profit is ((8 - 1) - 2) + ((9 - 4) - 2) = 8.

Example 2:

Input: prices = [1,3,7,5,10,3], fee = 3

Output: 6

Constraints:

- 1 <= prices.length <= $5 * 10^4$
- 1 <= prices[i] < $5 * 10^4$
- 0 <= fee < $5 * 10^4$

7.1 300. Longest Increasing Subsequence

Given an integer array nums, return *the length of the longest* ***strictly increasing subsequence***.

Example 1:

Input: nums = [10,9,2,5,3,7,101,18]

Output: 4

Explanation: The longest increasing subsequence is [2,3,7,101], therefore the length is 4.

Example 2:

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

Output: 4

Example 3:

Input: nums = [7,7,7,7,7,7,7]

Output: 1

Constraints:

- 1 <= nums.length <= 2500
- -10^4 <= nums[i] <= 10^4

7.2 **Print Longest Increasing Subsequence**

Difficulty: **Medium**Accuracy: **51.81%**Submissions: **36K+**Points: **4**Average Time: **20m**

You are given an array of integers **arr[]**, return the **Longest Increasing Subsequence (LIS)** of the given array.
LIS is the longest subsequence where each element is **strictly** greater than the previous one.
Note: If multiple **LIS** of the same maximum length exist, return the one that appears **first** based on the **lexicographical order** of their **indices** (i.e., the earliest combination of positions from the original sequence).

Examples:

Input: arr[] = [10, 20, 3, 40]

Output: [10, 20, 40]

Explanation: [10, 20, 40] is the longest subsequence where each number is greater than the previous one, maintaining the original order.

Input: arr[] = [10, 22, 9, 33, 21, 50, 41, 60, 80]

Output: [10, 22, 33, 50, 60, 80]
Explanation: There are multiple longest Increasing subsequence of length 6, that is [10, 22, 33, 50, 60, 80] and [10 22 33 41 60 80]. The first one has lexicographic smallest order of indices.

Constraint:
$1 <= n < = 5*10^3$
$0 <= arr[i] <= 10$

7.3 300. Longest Increasing Subsequence

Given an integer array nums, return *the length of the longest* ***strictly increasing subsequence***.

Example 1:

Input: nums = [10,9,2,5,3,7,101,18]

Output: 4

Explanation: The longest increasing subsequence is [2,3,7,101], therefore the length is 4.

Example 2:

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

Output: 4

Example 3:

Input: nums = [7,7,7,7,7,7,7]

Output: 1

Constraints:

- 1 <= nums.length <= 2500
- -10^4 <= nums[i] <= 10^4

7.4 **368. Largest Divisible Subset**

Given a set of **distinct** positive integers nums, return the largest subset answer such that every pair (answer[i], answer[j]) of elements in this subset satisfies:

- answer[i] % answer[j] == 0, or
- answer[j] % answer[i] == 0

If there are multiple solutions, return any of them.

Example 1:

Input: nums = [1,2,3]

Output: [1,2]

Explanation: [1,3] is also accepted.

Example 2:

Input: nums = [1,2,4,8]

Output: [1,2,4,8]

Constraints:

- 1 <= nums.length <= 1000
- 1 <= nums[i] <= $2 * 10^9$
- All the integers in nums are **unique**.

7.5 1048. Longest String Chain

You are given an array of words where each word consists of lowercase English letters.

$word_A$ is a **predecessor** of $word_B$ if and only if we can insert **exactly one** letter anywhere in $word_A$ **without changing the order of the other characters** to make it equal to $word_B$.

- For example, "abc" is a **predecessor** of "abac", while "cba" is not a **predecessor** of "bcad".

A **word chain** is a sequence of words $[word_1, word_2, ..., word_k]$ with k >= 1, where $word_1$ is a **predecessor** of $word_2$, $word_2$ is a **predecessor** of $word_3$, and so on. A single word is trivially a **word chain** with k == 1.

Return *the **length** of the **longest possible word chain** with words chosen from the given list of* words.

Example 1:

Input: words = ["a","b","ba","bca","bda","bdca"]

Output: 4

Explanation: One of the longest word chains is ["a","ba","bda","bdca"].

Example 2:

Input: words = ["xbc","pcxbcf","xb","cxbc","pcxbc"]

Output: 5

Explanation: All the words can be put in a word chain ["xb", "xbc", "cxbc", "pcxbc", "pcxbcf"].

Example 3:

Input: words = ["abcd","dbqca"]

Output: 1

Explanation: The trivial word chain ["abcd"] is one of the longest word chains.

["abcd","dbqca"] is not a valid word chain because the ordering of the letters is changed.

Constraints:

- 1 <= words.length <= 1000
- 1 <= words[i].length <= 16

- words[i] only consists of lowercase English letters

7.6 **Longest Bitonic subsequence**

Difficulty: **Medium**Accuracy: **47.34%**Submissions: **120K+**Points: **4**

Given an array of positive integers. Find the **maximum** length of **Bitonic subsequence.** A subsequence of array is called Bitonic if it is first strictly increasing, then strictly decreasing. Return the maximum length of bitonic subsequence.

Note : A strictly increasing or a **strictly** decreasing sequence should not be considered as a bitonic sequence

Examples :

Input: n = 5, nums[] = [1, 2, 5, 3, 2]

Output: 5

Explanation: The sequence {1, 2, 5} is increasing and the sequence {3, 2} is decreasing so merging both we will get length 5.

Input: n = 8, nums[] = [1, 11, 2, 10, 4, 5, 2, 1]

Output: 6

Explanation: The bitonic sequence {1, 2, 10, 4, 2, 1} has length 6.

Input: n = 3, nums[] = [10, 20, 30]

Output: 0

Explanation: The decreasing or increasing part cannot be empty

Input: n = 3, nums[] = [10, 10, 10]

Output: 0

Explanation: The subsequences must be strictly increasing or decreasing

Constraints:
$1 \leq$ length of array $\leq 10^3$
$1 \leq arr[i] \leq 10^4$

7.7 673. Number of Longest Increasing Subsequence

Given an integer array nums, return *the number of longest increasing subsequences*.

Notice that the sequence has to be **strictly** increasing.

Example 1:

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

Output: 2

Explanation: The two longest increasing subsequences are [1, 3, 4, 7] and [1, 3, 5, 7].

Example 2:

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

Output: 5

Explanation: The length of the longest increasing subsequence is 1, and there are 5 increasing subsequences of length 1, so output 5.

Constraints:

- 1 <= nums.length <= 2000
- -10^6 <= nums[i] <= 10^6
- The answer is guaranteed to fit inside a 32-bit integer.

8.1 **Matrix Chain Multiplication**

Difficulty: **Hard**Accuracy: **49.64%**Submissions: **159K+**Points: **8**

Given an array **arr[]** which represents **the** dimensions of a sequence of matrices where the **i^{th}** matrix has the dimensions **(arr[i-1] x arr[i])** for i>=1, find the most efficient way to multiply these matrices together. The efficient way is the one that involves the least number of multiplications.

Examples:

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

Output: 20

Explanation: There are 3 matrices of dimensions 2 × 1, 1 × 3, and 3 × 4, Let this 3 input matrices be M1, M2, and M3. There are two ways to multiply: ((M1 x M2) x M3) and (M1 x (M2 x M3)), note that the result of (M1 x M2) is a 2 x 3 matrix and result of (M2 x M3) is a 1 x 4 matrix.
((M1 x M2) x M3) requires (2 x 1 x 3) + (2 x 3 x 4) = 30

(M1 x (M2 x M3)) requires (1 x 3 x 4) + (2 x 1 x 4) = 20.
The minimum of these two is 20.

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

Output: 30

Explanation: There are 4 matrices of dimensions 1 × 2, 2 × 3, 3 × 4, 4 × 3. Let this 4 input matrices be M1, M2, M3 and M4. The minimum number of multiplications are obtained by ((M1 x M2) x M3) x M4). The minimum number is (1 x 2 x 3) + (1 x 3 x 4) + (1 x 4 x 3) = 30.

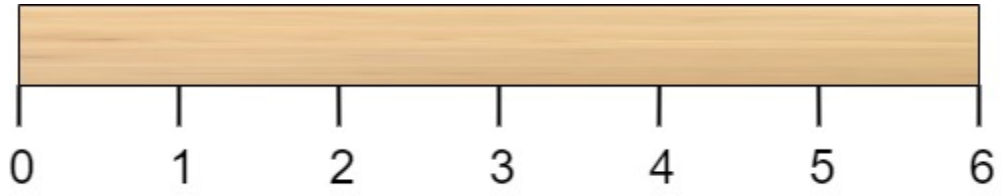
Input: arr[] = [3, 4]

Output: 0
Explanation: As there is only one matrix so, there is no cost of multiplication.

Constraints:
2 ≤ arr.size() ≤ 100
1 ≤ arr[i] ≤ 200

8.3 **1547. Minimum Cost to Cut a Stick**

Given a wooden stick of length n units. The stick is labelled from 0 to n. For example, a stick of length **6** is labelled as follows:



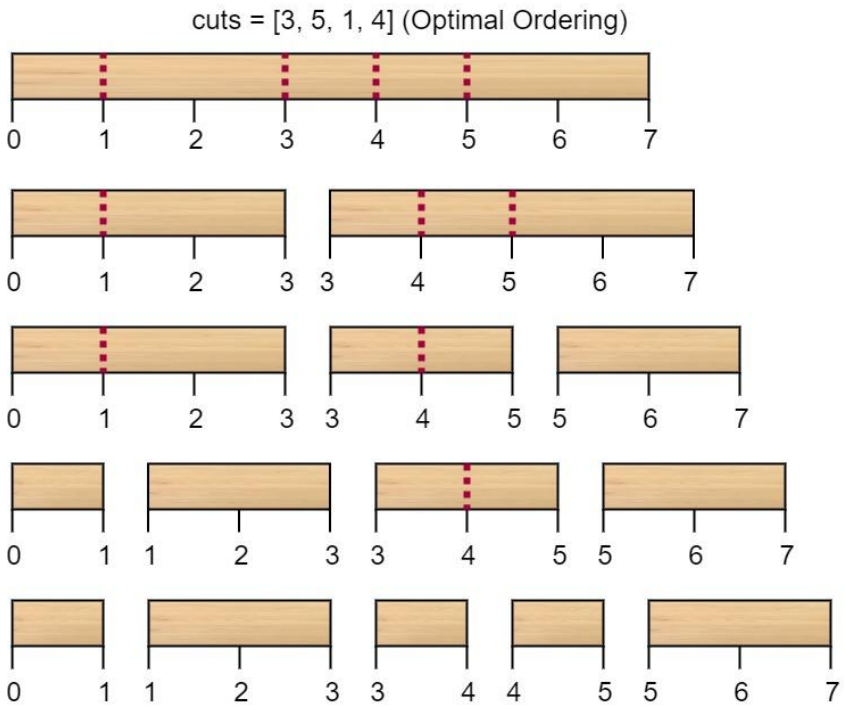
Given an integer array cuts where cuts[i] denotes a position you should perform a cut at.

You should perform the cuts in order, you can change the order of the cuts as you wish.

The cost of one cut is the length of the stick to be cut, the total cost is the sum of costs of all cuts. When you cut a stick, it will be split into two smaller sticks (i.e. the sum of their lengths is the length of the stick before the cut). Please refer to the first example for a better explanation.

Return *the minimum total cost* of the cuts.

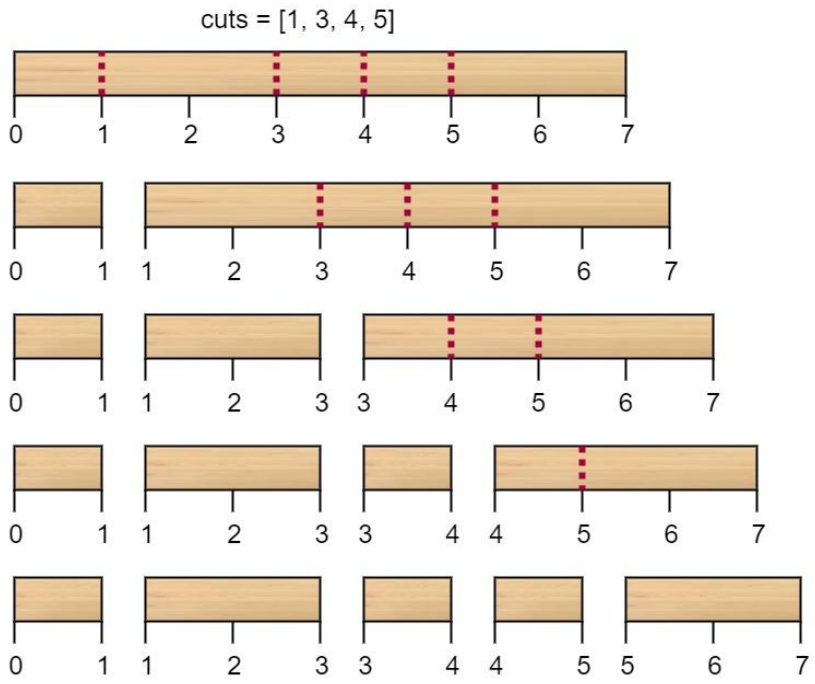
Example 1:



Input: n = 7, cuts = [1,3,4,5]

Output: 16

Explanation: Using cuts order = [1, 3, 4, 5] as in the input leads to the following scenario:



The first cut is done to a rod of length 7 so the cost is 7. The second cut is done to a rod of length 6 (i.e. the second part of the first cut), the third is done to a rod of length 4 and the last cut is to a rod of length 3. The total cost is 7 + 6 + 4 + 3 = 20.

Rearranging the cuts to be [3, 5, 1, 4] for example will lead to a scenario with total cost = 16 (as shown in the example photo 7 + 4 + 3 + 2 = 16).

Example 2:

Input: n = 9, cuts = [5,6,1,4,2]

Output: 22

Explanation: If you try the given cuts ordering the cost will be 25.

There are much ordering with total cost <= 25, for example, the order [4, 6, 5, 2, 1] has total cost = 22 which is the minimum possible.

Constraints:

- $2 <= n <= 10^6$
- 1 <= cuts.length <= min(n - 1, 100)
- 1 <= cuts[i] <= n - 1
- All the integers in cuts array are **distinct**.

8.4 **312. Burst Balloons**

You are given n balloons, indexed from 0 to n - 1. Each balloon is painted with a number on it represented by an array nums. You are asked to burst all the balloons.

If you burst the i^{th} balloon, you will get nums[i - 1] * nums[i] * nums[i + 1] coins. If i - 1 or i + 1 goes out of bounds of the array, then treat it as if there is a balloon with a 1 painted on it.

Return *the maximum coins you can collect by bursting the balloons wisely*.

Example 1:

Input: nums = [3,1,5,8]

Output: 167

Explanation:

nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []

coins = 3*1*5 + 3*5*8 + 1*3*8 + 1*8*1 = 167

Example 2:

Input: nums = [1,5]

Output: 10

Constraints:

- n == nums.length
- 1 <= n <= 300
- 0 <= nums[i] <= 100

8.5 1106. Parsing A Boolean Expression

A **boolean expression** is an expression that evaluates to either true or false. It can be in one of the following shapes:

- 't' that evaluates to true.
- 'f' that evaluates to false.
- '!(subExpr)' that evaluates to **the logical NOT** of the inner expression subExpr.
- '&(subExpr$_1$, subExpr$_2$, ..., subExpr$_n$)' that evaluates to **the logical AND** of the inner expressions subExpr$_1$, subExpr$_2$, ..., subExpr$_n$ where n >= 1.
- '|(subExpr$_1$, subExpr$_2$, ..., subExpr$_n$)' that evaluates to **the logical OR** of the inner expressions subExpr$_1$, subExpr$_2$, ..., subExpr$_n$ where n >= 1.

Given a string expression that represents a **boolean expression**, return *the evaluation of that expression*.

It is **guaranteed** that the given expression is valid and follows the given rules.

Example 1:

Input: expression = "&(|(f))"

Output: false

Explanation:

First, evaluate |(f) --> f. The expression is now "&(f)".

Then, evaluate &(f) --> f. The expression is now "f".

Finally, return false.

Example 2:

Input: expression = "|(f,f,f,t)"

Output: true

Explanation: The evaluation of (false OR false OR false OR true) is true.

Example 3:

Input: expression = "!(&(f,t))"

Output: true

Explanation:

First, evaluate &(f,t) --> (false AND true) --> false --> f. The expression is now "!(f)".

Then, evaluate !(f) --> NOT false --> true. We return true.

Constraints:

- 1 <= expression.length <= $2 * 10^4$
- expression[i] is one following characters: '(', ')', '&', '|', '!', 't', 'f', and ','.

8.6 **132. Palindrome Partitioning II**

Given a string s, partition s such that every substring of the partition is a palindrome.

Return *the* ***minimum*** *cuts needed for a palindrome partitioning of* s.

Example 1:

Input: s = "aab"

Output: 1

Explanation: The palindrome partitioning ["aa","b"] could be produced using 1 cut.

Example 2:

Input: s = "a"

Output: 0

Example 3:

Input: s = "ab"

Output: 1

Constraints:

- 1 <= s.length <= 2000
- s consists of lowercase English letters only.

8.7 **1043. Partition Array for Maximum Sum**

Given an integer array arr, partition the array into (contiguous) subarrays of length **at most** k. After partitioning, each subarray has their values changed to become the maximum value of that subarray.

Return *the largest sum of the given array after partitioning. Test cases are generated so that the answer fits in a* ***32-bit*** *integer.*

Example 1:

Input: arr = [1,15,7,9,2,5,10], k = 3

Output: 84

Explanation: arr becomes [15,15,15,9,10,10,10]

Example 2:

Input: arr = [1,4,1,5,7,3,6,1,9,9,3], k = 4

Output: 83

Example 3:

Input: arr = [1], k = 1

Output: 1

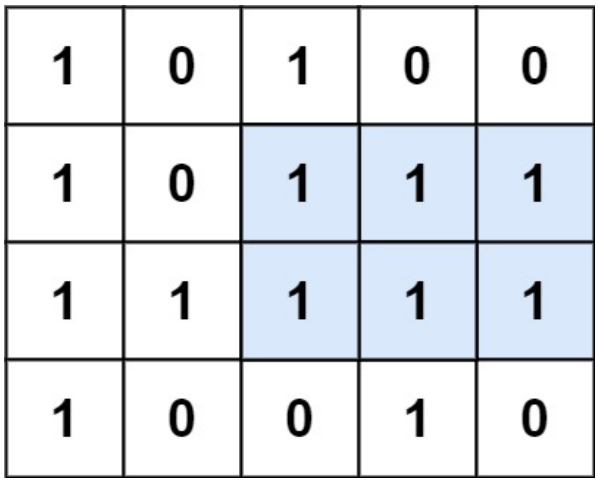
Constraints:

- 1 <= arr.length <= 500
- 0 <= arr[i] <= 10^9
- 1 <= k <= arr.length

9.1 85. Maximal Rectangle

Given a rows x cols binary matrix filled with 0's and 1's, find the largest rectangle containing only 1's and return *its area*.

Example 1:



Input: matrix = [["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],["1","0","0","1","0"]]

Output: 6

Explanation: The maximal rectangle is shown in the above picture.

Example 2:

Input: matrix = [["0"]]

Output: 0

Example 3:

Input: matrix = [["1"]]

Output: 1

Constraints:

- rows == matrix.length
- cols == matrix[i].length
- 1 <= row, cols <= 200
- matrix[i][j] is '0' or '1'.

9.2 **Ones**

Given a m * n matrix of ones and zeros, return how many **square** submatrices have all ones.

Example 1:

Input: matrix =

[

[0,1,1,1],

[1,1,1,1],

[0,1,1,1]

]

Output: 15

Explanation:

There are **10** squares of side 1.

There are **4** squares of side 2.

There is **1** square of side 3.

Total number of squares = 10 + 4 + 1 = **15**.

Example 2:

Input: matrix =

[

[1,0,1],

[1,1,0],

[1,1,0]

]

Output: 7

Explanation:

There are **6** squares of side 1.

There is **1** square of side 2.

Total number of squares = 6 + 1 = **7**.

Constraints:

- 1 <= arr.length <= 300
- 1 <= arr[0].length <= 300
- 0 <= arr[i][j] <= 1