WEEK4

Contents

Combinations

Given two integers `n` and `k`, return *all possible combinations of* `k` *numbers chosen from the range* `[1, n]`.

You may return the answer in **any order**.

Example 1:

Input: `n = 4, k = 2` **Output:** `[[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]` **Explanation:** `There are 4 choose 2 = 6 total combinations. Note that combinations are unordered, i.e., [1,2] and [2,1] are considered to be the same combination.`

Example 2:

Input: `n = 1, k = 1` **Output:** `[[1]]` **Explanation:** `There is 1 choose 1 = 1 total combination.`

Constraints:

- `1 <= n <= 20`
- `1 <= k <= n`

class Solution { public: vector> combine(int n, int k) { } };

Combination Sum

Given an array of **distinct** integers `candidates` and a target integer `target`, return *a list of all **unique combinations** of* `candidates` *where the chosen numbers sum to* `target`*.* You may return the combinations in **any order**.

The **same** number may be chosen from `candidates` an **unlimited number of times**. Two combinations are unique if the frequency of at least one of the chosen numbers is different.

The test cases are generated such that the number of unique combinations that sum up to `target` is less than `150` combinations for the given input.

Example 1:

`Input:` `candidates = [2,3,6,7], target = 7` **`Output:`** `[[2,2,3],[7]]` **`Explanation:`** `2 and 3 are candidates, and 2 + 2 + 3 = 7. Note that 2 can be used multiple times. 7 is a candidate, and 7 = 7. These are the only two combinations.`

Example 2:

`Input:` `candidates = [2,3,5], target = 8` **`Output:`** `[[2,2,2,2],[2,3,3],[3,5]]`

Example 3:

`Input:` `candidates = [2], target = 1` **`Output:`** `[]`

Constraints:

- `1 <= candidates.length <= 30`
- `2 <= candidates[i] <= 40`
- All elements of `candidates` are **distinct**.
- `1 <= target <= 40`

class Solution { public: vector> combinationSum(vector& candidates, int target) { } };

Combination Sum II

Given a collection of candidate numbers (`candidates`) and a target number (`target`), find all unique combinations in `candidates` where the candidate numbers sum to `target`.

Each number in `candidates` may only be used **once** in the combination.

Note: The solution set must not contain duplicate combinations.

Example 1:

`Input:`` candidates = [10,1,2,7,6,1,5], target = 8 `**`Output:`**` [ [1,1,6], [1,2,5], [1,7], [2,6] ]`

Example 2:

`Input:`` candidates = [2,5,2,1,2], target = 5 `**`Output:`**` [ [1,2,2], [5] ]`

Constraints:

- `1 <= candidates.length <= 100`
- `1 <= candidates[i] <= 50`
- `1 <= target <= 30`

class Solution { public: vector> combinationSum2(vector& candidates, int target) { } };

Combination Sum III

Find all valid combinations of `k` numbers that sum up to `n` such that the following conditions are true:

- Only numbers `1` through `9` are used.
- Each number is used **at most once**.

Return *a list of all possible valid combinations*. The list must not contain the same combination twice, and the combinations may be returned in any order.

Example 1:

`**Input:** k = 3, n = 7 **Output:** [[1,2,4]] **Explanation:** 1 + 2 + 4 = 7 There are no other valid combinations.`

Example 2:

`**Input:** k = 3, n = 9 **Output:** [[1,2,6],[1,3,5],[2,3,4]] **Explanation:** 1 + 2 + 6 = 9 1 + 3 + 5 = 9 2 + 3 + 4 = 9 There are no other valid combinations.`

Example 3:

`**Input:** k = 4, n = 1 **Output:** [] **Explanation:** There are no valid combinations. Using 4 different numbers in the range [1,9], the smallest sum we can get is 1+2+3+4 = 10 and since 10 > 1, there are no valid combination.`

Constraints:

- `2 <= k <= 9`
- `1 <= n <= 60`

class Solution { public: vector> combinationSum3(int k, int n) { } };

Generate Parentheses

Given n pairs of parentheses, write a function to *generate all combinations of well-formed parentheses*.

Example 1:

`**Input:** n = 3 **Output:** ["((()))","(()())","(())()","()(())","()()()"]`

Input: `n = 3` **Output:** `["((()))","(()())","(())()","()(())","()()()"]`

Example 2:

Input: `n = 1` **Output:** `["()"]`

Constraints:

- `1 <= n <= 8`

class Solution { public: vector generateParenthesis(int n) { } };

Target Sum

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`

class Solution { public: int findTargetSumWays(vector& nums, int target) { } };

Palindrome Partitioning

Given a string `s`, partition `s` such that every substring of the partition is a **palindrome**. Return *all possible palindrome partitioning of* `s`.

Example 1:

`Input:` `s = "aab"` **`Output:`** `[["a","a","b"],["aa","b"]]`

Example 2:

`Input:` `s = "a"` **`Output:`** `[["a"]]`

Constraints:

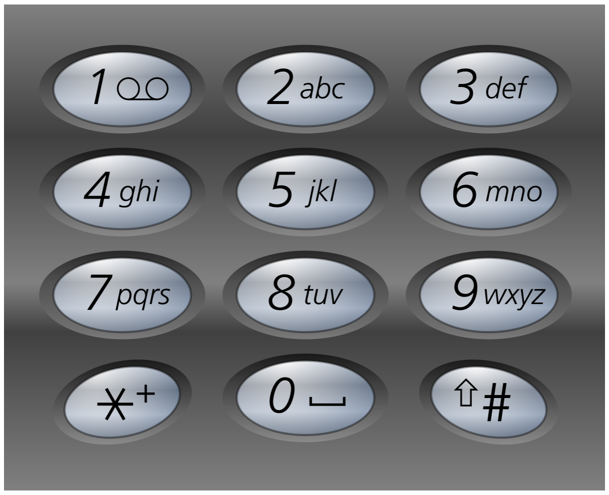
- `1 <= s.length <= 16`
- s contains only lowercase English letters.

class Solution { public: vector> partition(string s) { } };

Letter Combinations of a Phone Number

Given a string containing digits from `2-9` inclusive, return all possible letter combinations that the number could represent. Return the answer in **any order**.

A mapping of digits to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.



Example 1:

`Input:` `digits = "23"` **`Output:`** `["ad","ae","af","bd","be","bf","cd","ce","cf"]`

Example 2:

`Input:` `digits = ""` **`Output:`** `[]`

Example 3:

`Input:` `digits = "2"` **`Output:`** `["a","b","c"]`

Constraints:

- `0 <= digits.length <= 4`
- `digits[i]` is a digit in the range `['2', '9']`.

class Solution { public: vector letterCombinations(string digits) { } };

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`

class Solution { public: int rob(vector& nums) { } };

Maximum Subarray

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

Example 1:

`**Input:** nums = [-2,1,-3,4,-1,2,1,-5,4] **Output:** 6 **Explanation:** The subarray [4,-1,2,1] has the largest sum 6.`

Example 2:

`**Input:** nums = [1] **Output:** 1 **Explanation:** The subarray [1] has the largest sum 1.`

Example 3:

`**Input:** nums = [5,4,-1,7,8] **Output:** 23 **Explanation:** The subarray [5,4,-1,7,8] has the largest sum 23.`

Constraints:

- `1 <= nums.length <=` 10^5
- -10^4 `<= nums[i] <=` 10^4

Follow up: If you have figured out the `O(n)` solution, try coding another solution using the **divide and conquer** approach, which is more subtle.

class Solution { public: int maxSubArray(vector& nums) { } };

House Robber II

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

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

Example 1:

`Input:` `nums = [2,3,2]` **`Output:`** `3` **`Explanation:`** `You cannot rob house 1 (money = 2) and then rob house 3 (money = 2), because they are adjacent houses.`

Example 2:

`Input:` `nums = [1,2,3,1]` **`Output:`** `4` **`Explanation:`** `Rob house 1 (money = 1) and then rob house 3 (money = 3). Total amount you can rob = 1 + 3 = 4.`

Example 3:

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

Constraints:

- `1 <= nums.length <= 100`
- `0 <= nums[i] <= 1000`

class Solution { public: int rob(vector& nums) { } };

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}

class Solution { public: int coinChange(vector& coins, int amount) { } };

Maximum Product Subarray

Given an integer array `nums`, find a subarray that has the largest product, and return *the product*.

The test cases are generated so that the answer will fit in a **32-bit** integer.

Example 1:

`**Input:** nums = [2,3,-2,4] **Output:** 6 **Explanation:** [2,3] has the largest product 6.`

Example 2:

`**Input:** nums = [-2,0,-1] **Output:** 0 **Explanation:** The result cannot be 2, because [-2,-1] is not a subarray.`

Constraints:

- `1 <= nums.length <= 2 * 10`^{4}
- `-10 <= nums[i] <= 10`
- The product of any subarray of `nums` is **guaranteed** to fit in a **32-bit** integer.

class Solution { public: int maxProduct(vector& nums) { } };

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}

Follow up: Can you come up with an algorithm that runs in `O(n log(n))` time complexity?

class Solution { public: int lengthOfLIS(vector& nums) { } };