WEEK5

Contents

Longest Palindromic Substring

Given a string `s`, return *the longest palindromic substring* in `s`.

Example 1:

`Input:`` s = "babad" `**`Output:`**` "bab" `**`Explanation:`**` "aba" is also a valid answer.`

Example 2:

`Input:`` s = "cbbd" `**`Output:`**` "bb"`

Constraints:

- `1 <= s.length <= 1000`
- s consist of only digits and English letters.

class Solution { public: string longestPalindrome(string s) { } };

Word Break

Given a string `s` and a dictionary of strings `wordDict`, return `true` if `s` can be segmented into a space-separated sequence of one or more dictionary words.

Note that the same word in the dictionary may be reused multiple times in the segmentation.

Example 1:

Input: `s = "leetcode", wordDict = ["leet","code"]` **Output:** `true` **Explanation:** `Return true because "leetcode" can be segmented as "leet code".`

Example 2:

Input: `s = "applepenapple", wordDict = ["apple","pen"]` **Output:** `true` **Explanation:** `Return true because "applepenapple" can be segmented as "apple pen apple". Note that you are allowed to reuse a dictionary word.`

Example 3:

Input: `s = "catsandog", wordDict = ["cats","dog","sand","and","cat"]` **Output:** `false`

Constraints:

- `1 <= s.length <= 300`
- `1 <= wordDict.length <= 1000`
- `1 <= wordDict[i].length <= 20`
- `s` and `wordDict[i]` consist of only lowercase English letters.
- All the strings of `wordDict` are **unique**.

class Solution { public: bool wordBreak(string s, vector& wordDict) { } };

Combination Sum IV

Given an array of **distinct** integers `nums` and a target integer `target`, return *the number of possible combinations that add up to* `target`.

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

Example 1:

Input: nums = [1,2,3], target = 4 **Output:** 7 **Explanation:** The possible combination ways are: (1, 1, 1, 1) (1, 1, 2) (1, 2, 1) (1, 3) (2, 1, 1) (2, 2) (3, 1) Note that different sequences are counted as different combinations.

Example 2:

Input: nums = [9], target = 3 **Output:** 0

Constraints:

- `1 <= nums.length <= 200`
- `1 <= nums[i] <= 1000`
- All the elements of `nums` are **unique**.
- `1 <= target <= 1000`

Follow up: What if negative numbers are allowed in the given array? How does it change the problem? What limitation we need to add to the question to allow negative numbers?

class Solution { public: int combinationSum4(vector& nums, int target) { } };

Decode Ways

You have intercepted a secret message encoded as a string of numbers. The message is **decoded** via the following mapping:

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

However, while decoding the message, you realize that there are many different ways you can decode the message because some codes are contained in other codes (`"2"` and `"5"` vs `"25"`).

For example, `"11106"` can be decoded into:

- `"AAJF"` with the grouping `(1, 1, 10, 6)`
- `"KJF"` with the grouping `(11, 10, 6)`
- The grouping `(1, 11, 06)` is invalid because `"06"` is not a valid code (only `"6"` is valid).

Note: there may be strings that are impossible to decode.

Given a string s containing only digits, return the **number of ways** to **decode** it. If the entire string cannot be decoded in any valid way, return `0`.

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

Example 1:

Input: s = "12"

Output: 2

Explanation:

"12" could be decoded as "AB" (1 2) or "L" (12).

Example 2:

Input: s = "226"

Output: 3

Explanation:

"226" could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2 2 6).

Example 3:

Input: s = "06"

Output: 0

Explanation:

"06" cannot be mapped to "F" because of the leading zero ("6" is different from "06"). In this case, the string is not a valid encoding, so return 0.

Constraints:

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

```
class Solution { public: int numDecodings(string s) { } };
```

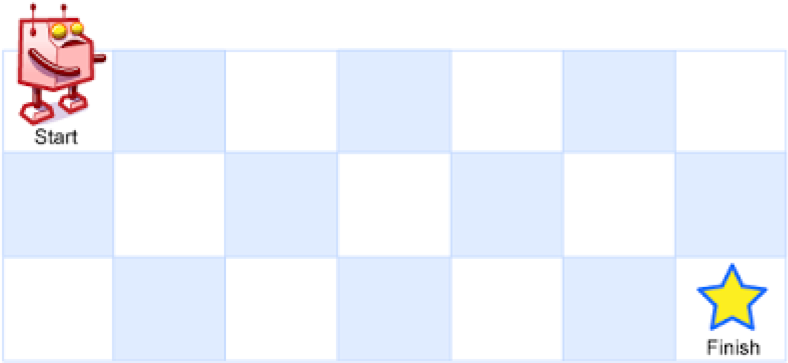
Unique Paths

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`

class Solution { public: int uniquePaths(int m, int n) { } };

Jump Game

You are given an integer array `nums`. You are initially positioned at the array's **first index**, and each element in the array represents your maximum jump length at that position.

Return `true` *if you can reach the last index, or* `false` *otherwise*.

Example 1:

```
Input: nums = [2,3,1,1,4] Output: true Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.
```

Example 2:

```
Input: nums = [3,2,1,0,4] Output: false Explanation: You will always arrive at index 3 no matter what. Its maximum jump length is 0, which makes it impossible to reach the last index.
```

Constraints:

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

class Solution { public: bool canJump(vector& nums) { } };

Palindromic Substrings

Given a string `s`, return *the number of* ***palindromic substrings*** *in it*.

A string is a **palindrome** when it reads the same backward as forward.

A **substring** is a contiguous sequence of characters within the string.

Example 1:

```
**Input:** s = "abc" **Output:** 3 **Explanation:** Three palindromic strings: "a", "b", "c".
```

Example 2:

```
**Input:** s = "aaa" **Output:** 6 **Explanation:** Six palindromic strings: "a", "a", "a", "aa", "aa", "aaa".
```

Constraints:

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

class Solution { public: int countSubstrings(string s) { } };

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.

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

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`

class Solution { public: bool canPartition(vector& nums) { } };

Partition to K Equal Sum Subsets

Given an integer array `nums` and an integer `k`, return `true` if it is possible to divide this array into `k` non-empty subsets whose sums are all equal.

Example 1:

```
Input: nums = [4,3,2,3,5,2,1], k = 4 Output: true Explanation: It is possible to divide it into 4 subsets (5), (1, 4), (2,3), (2,3) with equal sums.
```

Example 2:

```
Input: nums = [1,2,3,4], k = 3 Output: false
```

Constraints:

- `1 <= k <= nums.length <= 16`
- `1 <= nums[i] <=` 10^4
- The frequency of each element is in the range `[1, 4]`.

class Solution { public: bool canPartitionKSubsets(vector& nums, int k) { } };

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`

class Solution { public: int maxProfit(vector& prices) { } };

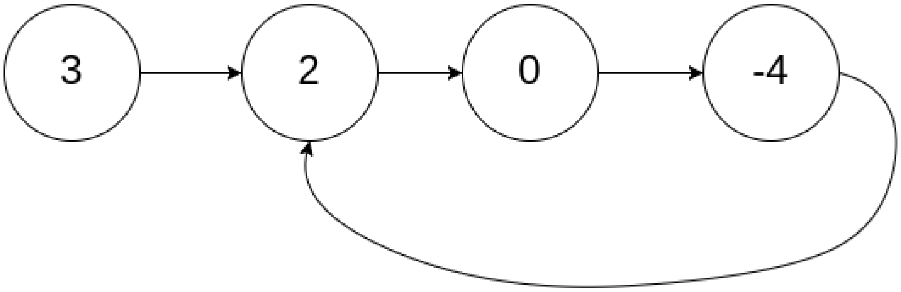
Linked List Cycle II

Given the `head` of a linked list, return *the node where the cycle begins. If there is no cycle, return* `null`.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the `next` pointer. Internally, `pos` is used to denote the index of the node that tail's `next` pointer is connected to (**0-indexed**). It is `-1` if there is no cycle. **Note that** `pos` **is not passed as a parameter**.

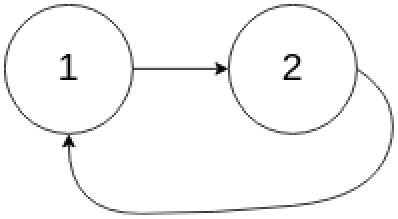
Do not modify the linked list.

Example 1:



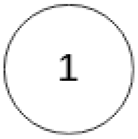
Input: head = [3,2,0,-4], pos = 1 **Output:** tail connects to node index 1 **Explanation:** There is a cycle in the linked list, where tail connects to the second node.

Example 2:



Input: head = [1,2], pos = 0 **Output:** tail connects to node index 0 **Explanation:** There is a cycle in the linked list, where tail connects to the first node.

Example 3:



Input: head = [1], pos = -1 **Output:** no cycle **Explanation:** There is no cycle in the linked list.

Constraints:

- The number of the nodes in the list is in the range `[0, 10`^{4}`]`.
- $-10^5 <= \text{Node.val} <= 10^5$
- `pos` is `-1` or a **valid index** in the linked-list.

Follow up: Can you solve it using `O(1)` (i.e. constant) memory?

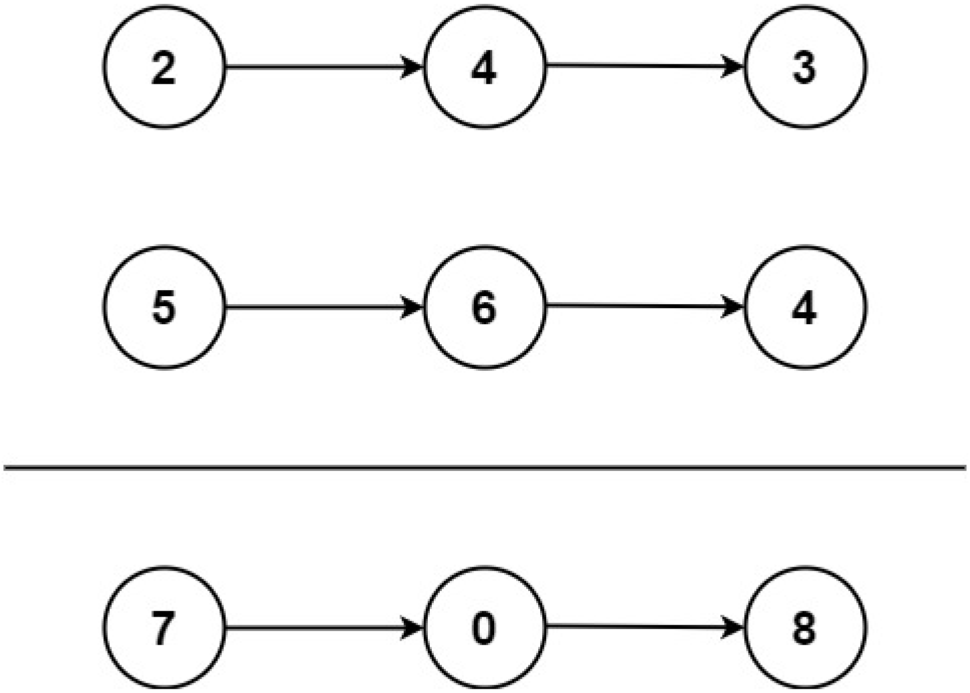
```
/** * Definition for singly-linked list. * struct ListNode { * int val; * ListNode *next; * ListNode(int x) : val(x), next(NULL) {} * }; */ class Solution { public: ListNode *detectCycle(ListNode *head) { } };
```

Add Two Numbers

You are given two **non-empty** linked lists representing two non-negative integers. The digits are stored in **reverse order**, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

Example 1:



Input: `l1 = [2,4,3], l2 = [5,6,4]` **Output:** `[7,0,8]` **Explanation:** `342 + 465 = 807.`

Example 2:

Input: `l1 = [0], l2 = [0]` **Output:** `[0]`

Example 3:

Input: `l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]` **Output:** `[8,9,9,9,0,0,0,1]`

Constraints:

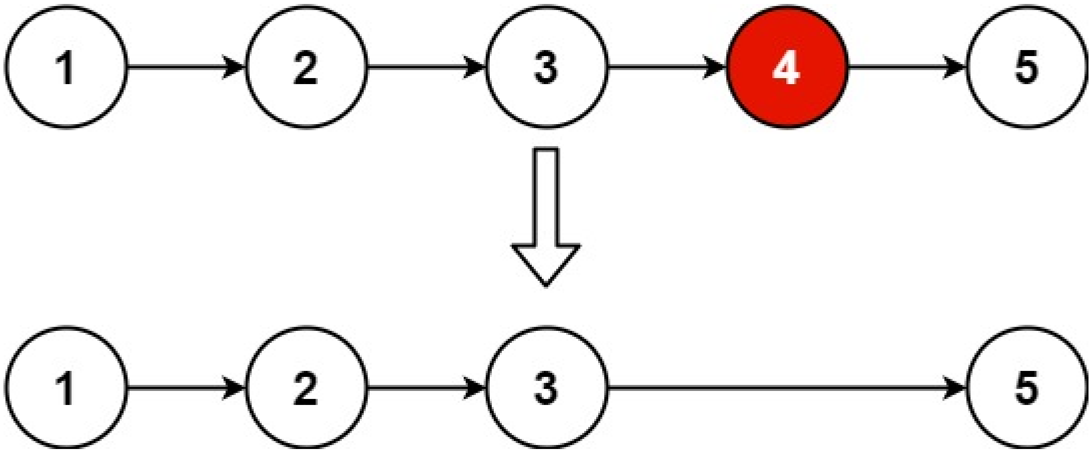
- The number of nodes in each linked list is in the range `[1, 100]`.
- `0 <= Node.val <= 9`
- It is guaranteed that the list represents a number that does not have leading zeros.

/** * Definition for singly-linked list. * struct ListNode { * int val; * ListNode *next; * ListNode() : val(0), next(nullptr) {} * ListNode(int x) : val(x), next(nullptr) {} * ListNode(int x, ListNode *next) : val(x), next(next) {} * }; */ class Solution { public: ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) { } };

Remove Nth Node From End of List

Given the `head` of a linked list, remove the n^{th} node from the end of the list and return its head.

Example 1:



Input: `head = [1,2,3,4,5], n = 2` **Output:** `[1,2,3,5]`

Example 2:

Input: `head = [1], n = 1` **Output:** `[]`

Example 3:

Input: `head = [1,2], n = 1` **Output:** `[1]`

Constraints:

- The number of nodes in the list is `sz`.
- `1 <= sz <= 30`
- `0 <= Node.val <= 100`
- `1 <= n <= sz`

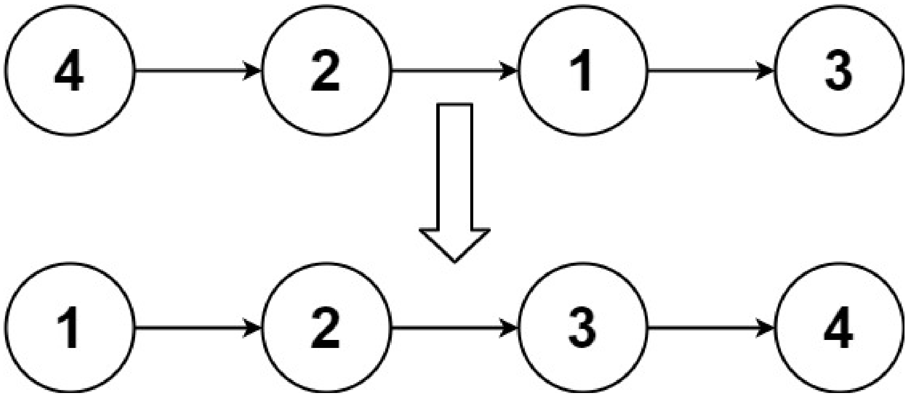
Follow up: Could you do this in one pass?

/** * Definition for singly-linked list. * struct ListNode { * int val; * ListNode *next; * ListNode() : val(0), next(nullptr) {} * ListNode(int x) : val(x), next(nullptr) {} * ListNode(int x, ListNode *next) : val(x), next(next) {} * }; */ class Solution { public: ListNode* removeNthFromEnd(ListNode* head, int n) { } };

Sort List

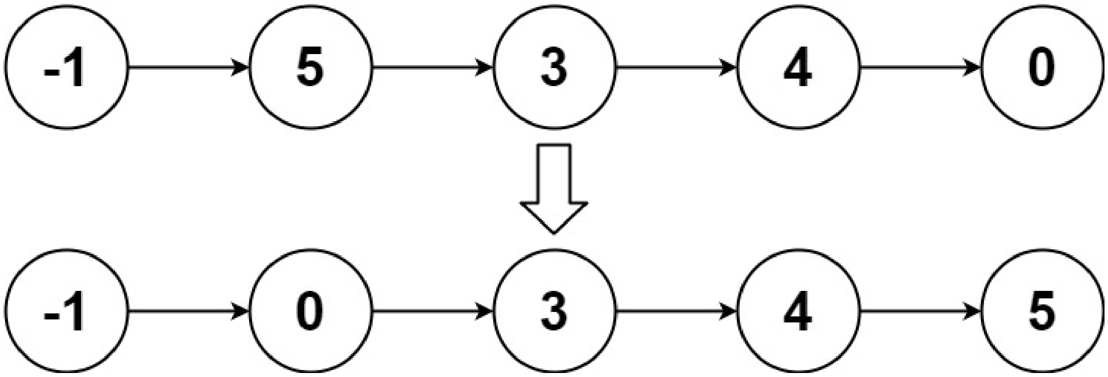
Given the `head` of a linked list, return *the list after sorting it in **ascending order***.

Example 1:



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

Example 2:



Input: `head = [-1,5,3,4,0]` **Output:** `[-1,0,3,4,5]`

Example 3:

Input: `head = []` **Output:** `[]`

Constraints:

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

Follow up: Can you sort the linked list in `O(n logn)` time and `O(1)` memory (i.e. constant space)?

```
/** * Definition for singly-linked list. * struct ListNode { * int val; * ListNode *next; * ListNode() : val(0), next(nullptr) {} * ListNode(int x) : val(x),
next(nullptr) {} * ListNode(int x, ListNode *next) : val(x), next(next) {} * }; */ class Solution { public: ListNode* sortList(ListNode* head) { } };
```