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

```
class Solution:
    def coinChange(self, coins, amount):
        # Initialize the DP array with a large value (amount + 1 is sufficient)
        dp = [amount + 1] * (amount + 1)
        dp[0] = 0  # Base case: no coins are needed to make up amount 0

        # Iterate through all amounts from 1 to `amount`
        for i in range(1, amount + 1):
            for coin in coins:
                if i - coin >= 0:  # If the coin can contribute to the amount
                    dp[i] = min(dp[i], dp[i - coin] + 1)
```

If dp[amount] is still greater than amount, it means it's not possible to make up the amount
return dp[amount] if dp[amount] != amount + 1 else -1

79. Perfect Squares

Given an integer n, return *the least number of perfect square numbers that sum to* n.

A **perfect square** is an integer that is the square of an integer; in other words, it is the product of some integer with itself. For example, 1, 4, 9, and 16 are perfect squares while 3 and 11 are not.

Example 1:

Input: n = 12
Output: 3
Explanation: 12 = 4 + 4 + 4.

Example 2:

Input: n = 13
Output: 2
Explanation: 13 = 4 + 9.

Constraints:

- 1 <= n <= 10^4

```
import math

class Solution:
    def numSquares(self, n: int) -> int:
        # Create a dp array where dp[i] means the least number of perfect squares that sum up to i
        dp = [float('inf')] * (n + 1)
        dp[0] = 0  # Base case: 0 perfect squares needed to sum up to 0

        # Iterate through all numbers from 1 to n
        for i in range(1, n + 1):
```

```
            # For each i, check all perfect squares less than or equal to i
            for square in range(1, int(math.sqrt(i)) + 1):
                dp[i] = min(dp[i], dp[i - square * square] + 1)

        return dp[n]

# Test the Solution class
param_1 = 12
ret = Solution().numSquares(param_1)
print(ret)  # Output: 3
```

994. Rotting Oranges

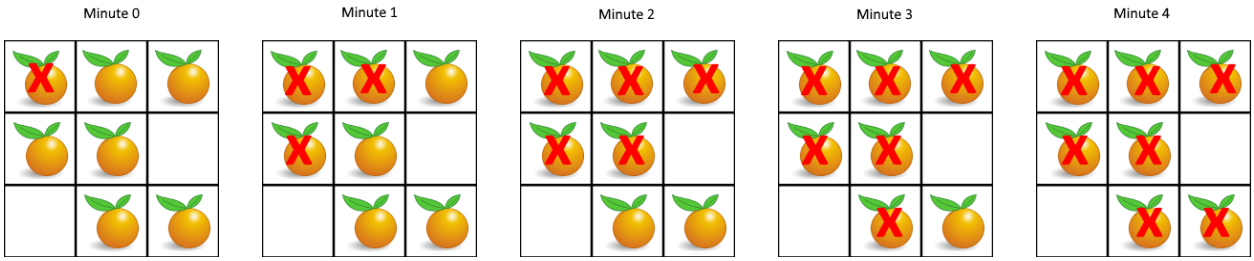
You are given an m x n grid where each cell can have one of three values:

- 0 representing an empty cell,
- 1 representing a fresh orange, or
- 2 representing a rotten orange.

Every minute, any fresh orange that is **4-directionally adjacent** to a rotten orange becomes rotten.

Return *the minimum number of minutes that must elapse until no cell has a fresh orange*. If *this is impossible, return* -1.

Example 1:



Input: grid = [[2,1,1],[1,1,0],[0,1,1]]
Output: 4

Example 2:

Input: grid = [[2,1,1],[0,1,1],[1,0,1]]
Output: -1

Explanation: The orange in the bottom left corner (row 2, column 0) is never rotten, because rotting only happens 4-directionally.

Example 3:

Input: grid = [[0,2]]
Output: 0
Explanation: Since there are already no fresh oranges at minute 0, the answer is just 0.

Constraints:

- m == grid.length
- n == grid[i].length
- 1 <= m, n <= 10
- grid[i][j] is 0, 1, or 2.

```
from collections import deque

class Solution:
    def orangesRotting(self, grid):
        rows, cols = len(grid), len(grid[0])
        queue = deque()  # For BFS
        fresh_oranges = 0

        # Step 1: Initialize the queue with all rotten oranges and count fresh oranges
        for r in range(rows):
            for c in range(cols):
                if grid[r][c] == 2:
                    queue.append((r, c, 0))  # (row, col, minutes_elapsed)
                elif grid[r][c] == 1:
                    fresh_oranges += 1

        # Step 2: Perform BFS to rot adjacent oranges
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Right, Down, Left, Up
        minutes = 0

        while queue:
            r, c, minutes = queue.popleft()
            for dr, dc in directions:
                nr, nc = r + dr, c + dc
                # If adjacent cell is a fresh orange, rot it and add to the queue
                if 0 <= nr < rows and 0 <= nc < cols and grid[nr][nc] == 1:
                    grid[nr][nc] = 2  # Mark as rotten
                    fresh_oranges -= 1
```

```
                queue.append((nr, nc, minutes + 1))

    # Step 3: Check if there are still fresh oranges
    return minutes if fresh_oranges == 0 else -1
```

49. Group Anagrams

Given an array of strings strs, group the

anagrams
together. You can return the answer in **any order**.

Example 1:

Input: strs = ["eat","tea","tan","ate","nat","bat"]

Output: [["bat"],["nat","tan"],["ate","eat","tea"]]

Explanation:

- There is no string in strs that can be rearranged to form "bat".
- The strings "nat" and "tan" are anagrams as they can be rearranged to form each other.
- The strings "ate", "eat", and "tea" are anagrams as they can be rearranged to form each other.

Example 2:

Input: strs = [""]

Output: [[""]]

Example 3:

Input: strs = ["a"]

Output: [["a"]]

Constraints:

- 1 <= strs.length <= 10^4
- 0 <= strs[i].length <= 100
- strs[i] consists of lowercase English letters.

```
from collections import defaultdict

class Solution:
    def groupAnagrams(self, strs):
        anagram_map = defaultdict(list)  # Dictionary to hold lists of anagrams
        for word in strs:
            # Sort the word and use it as the key
            sorted_word = ''.join(sorted(word))
            anagram_map[sorted_word].append(word)

        # Return the values of the dictionary as a list of lists
        return list(anagram_map.values())
```

1. Two Sum

Given an array of integers nums and an integer target, return *indices of the two numbers such that they add up to target*.

You may assume that each input would have ***exactly* one solution**, and you may not use the *same* element twice.

You can return the answer in any order.

Example 1:

Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].

Example 2:

Input: nums = [3,2,4], target = 6
Output: [1,2]

Example 3:

Input: nums = [3,3], target = 6
Output: [0,1]

Constraints:

- 2 <= nums.length <= 10^4
- -10^9 <= nums[i] <= 10^9
- -10^9 <= target <= 10^9
- **Only one valid answer exists.**
-

```
class Solution:
    def twoSum(self, nums, target):
        num_map = {}
        for i, num in enumerate(nums):
            complement = target - num
            if complement in num_map:
                return [num_map[complement], i]
            num_map[num] = i
        return []

# Example usage:
# param_1 = [2, 7, 11, 15]
# param_2 = 9
# ret = Solution().twoSum(param_1, param_2)
# print(ret)  # Output: [0, 1]
```

242. Valid Anagram

Given two strings s and t, return true if t is an

anagram
of s, and false otherwise.

Example 1:

Input: s = "anagram", t = "nagaram"

Output: true

Example 2:

Input: s = "rat", t = "car"

Output: false

Constraints:

- 1 <= s.length, t.length <= 5 * 10^4
- s and t consist of lowercase English letters.

```
from collections import Counter

class Solution:
    def isAnagram(self, s: str, t: str) -> bool:
        if len(s) != len(t):
            return False

        return Counter(s) == Counter(t)

# Driver code to test
def _driver():
    solution = Solution()  # Create an instance of Solution
    param_1 = "anagram"
    param_2 = "nagaram"
    ret = solution.isAnagram(param_1, param_2)  # Call method isAnagram
    print(ret)  # Output: True

_driver()  # Call the driver function
```

202. Happy Number

Write an algorithm to determine if a number n is happy.

A **happy number** is a number defined by the following process:

- Starting with any positive integer, replace the number by the sum of the squares of its digits.
- Repeat the process until the number equals 1 (where it will stay), or it **loops endlessly in a cycle** which does not include 1.
- Those numbers for which this process **ends in 1** are happy.

Return true *if* n *is a happy number, and* false *if not*.

Example 1:

Input: n = 19
Output: true
Explanation:
$1^2 + 9^2 = 82$
$8^2 + 2^2 = 68$

$6^2 + 8^2 = 100$
$1^2 + 0^2 + 0^2 = 1$

Example 2:

Input: n = 2
Output: false

Constraints:

- $1 <= n <= 2^{31} - 1$

```
class Solution:
    def isHappy(self, n: int) -> bool:
        def get_sum_of_squares(num: int) -> int:
            return sum(int(digit) ** 2 for digit in str(num))

        seen = set()

        while n != 1:
            if n in seen:
                return False
            seen.add(n)
            n = get_sum_of_squares(n)

        return True
```

705. Design HashSet

Design a HashSet without using any built-in hash table libraries.

Implement MyHashSet class:

- void add(key) Inserts the value key into the HashSet.
- bool contains(key) Returns whether the value key exists in the HashSet or not.
- void remove(key) Removes the value key in the HashSet. If key does not exist in the HashSet, do nothing.

Example 1:

Input

["MyHashSet", "add", "add", "contains", "contains", "add", "contains", "remove", "contains"]
[[], [1], [2], [1], [3], [2], [2], [2], [2]]
Output
[null, null, null, true, false, null, true, null, false]

Explanation
MyHashSet myHashSet = new MyHashSet();
myHashSet.add(1);   // set = [1]
myHashSet.add(2);   // set = [1, 2]
myHashSet.contains(1); // return True
myHashSet.contains(3); // return False, (not found)
myHashSet.add(2);   // set = [1, 2]
myHashSet.contains(2); // return True
myHashSet.remove(2);   // set = [1]
myHashSet.contains(2); // return False, (already removed)

Constraints:

- $0 <= key <= 10^6$
- At most 10^4 calls will be made to add, remove, and contains.

```
class MyHashSet:

    def __init__(self):
        # Initialize a list of buckets; each bucket is an empty list
        self.size = 1000  # A reasonable size for the hash table
        self.hash_table = [[] for _ in range(self.size)]

    def _hash(self, key: int) -> int:
        # Hash function that maps the key to an index in the hash table
        return key % self.size

    def add(self, key: int) -> None:
        # Calculate the index for the key
        index = self._hash(key)
        # Check if the key already exists in the bucket
        if key not in self.hash_table[index]:
            # If not, add it to the bucket
            self.hash_table[index].append(key)

    def contains(self, key: int) -> bool:
```

```
        # Calculate the index for the key
        index = self._hash(key)
        # Return True if the key exists in the corresponding bucket, False
otherwise
        return key in self.hash_table[index]

    def remove(self, key: int) -> None:
        # Calculate the index for the key
        index = self._hash(key)
        # If the key exists in the bucket, remove it
        if key in self.hash_table[index]:
            self.hash_table[index].remove(key)

# Example usage
myHashSet = MyHashSet()
myHashSet.add(1)
myHashSet.add(2)
print(myHashSet.contains(1))  # Output: True
print(myHashSet.contains(3))  # Output: False
myHashSet.add(2)
print(myHashSet.contains(2))  # Output: True
myHashSet.remove(2)
print(myHashSet.contains(2))  # Output: False
```

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

```
class Solution:
    def climbStairs(self, n: int) -> int:
        if n == 1:
            return 1
        # Initialize the first two steps
        a, b = 1, 1
        for i in range(2, n + 1):
            # Calculate the number of ways to reach the current step
            a, b = b, a + b
        return b
```

LeetCode #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

```
class Solution:
    def lengthOfLIS(self, nums):
        if not nums:
            return 0

        # Initialize the dp array with 1, since each number is a subsequence of length 1.
        dp = [1] * len(nums)

        # Loop through each element in the array to fill the dp array.
        for i in range(1, len(nums)):
            for j in range(i):
                if nums[i] > nums[j]:
                    dp[i] = max(dp[i], dp[j] + 1)

        # The length of the longest increasing subsequence will be the max value in dp.
        return max(dp)
```

Coin Change (LeetCode #322)

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:
   def coinChange(self, coins, amount):
       # Initialize the DP array with a large value (amount + 1 is sufficient)
       dp = [amount + 1] * (amount + 1)
       dp[0] = 0  # Base case: no coins are needed to make up amount 0

       # Iterate through all amounts from 1 to `amount`
       for i in range(1, amount + 1):
           for coin in coins:
               if i - coin >= 0:  # If the coin can contribute to the amount
                   dp[i] = min(dp[i], dp[i - coin] + 1)

       # If dp[amount] is still greater than amount, it means it's not possible
to make up the amount
       return dp[amount] if dp[amount] != amount + 1 else -1
```

Knapsack Problem (LeetCode #416)

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:
    def canPartition(self, nums):
        total_sum = sum(nums)

        # If total sum is odd, we cannot partition it into two equal subsets
        if total_sum % 2 != 0:
            return False

        target = total_sum // 2

        # Initialize the dp array, with dp[0] = True (we can always have a sum of 0)
        dp = [False] * (target + 1)
        dp[0] = True

        # Iterate through each number in the array
        for num in nums:
            # Update the dp array in reverse order
            for j in range(target, num - 1, -1):
                dp[j] = dp[j] or dp[j - num]

        # If dp[target] is True, we can partition the array into two equal subsets
        return dp[target]

# Example usage:
solution = Solution()
print(solution.canPartition([1, 5, 11, 5]))  # Output: True
print(solution.canPartition([1, 2, 3, 5]))   # Output: False
```

House Robber (LeetCode #198)

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:
    def rob(self, nums):
        if not nums:
            return 0
        if len(nums) == 1:
            return nums[0]

        # Initialize the first two base cases
        dp = [0] * len(nums)
        dp[0] = nums[0]
        dp[1] = max(nums[0], nums[1])

        # Fill the dp array for all other houses
        for i in range(2, len(nums)):
            dp[i] = max(dp[i-1], nums[i] + dp[i-2])
```

```
        # The answer is the maximum amount that can be robbed from all houses
        return dp[-1]
```

Word Break (LeetCode #139)

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:
    def wordBreak(self, s, wordDict):
        # Create a set of dictionary words for faster lookup
        wordSet = set(wordDict)

        # Initialize a dp array where dp[i] means s[0..i-1] can be segmented
        dp = [False] * (len(s) + 1)
```

```
        dp[0] = True  # Base case: an empty string can always be segmented

        # Iterate over each position in the string
        for i in range(1, len(s) + 1):
            for j in range(i):
                # Check if s[j..i] is a word in the dictionary and dp[j] is True
                if dp[j] and s[j:i] in wordSet:
                    dp[i] = True
                    break

        # The result is whether the entire string can be segmented
        return dp[len(s)]

# Test the solution
solution = Solution()
print(solution.wordBreak("leetcode", ["leet", "code"]))  # Output: True
print(solution.wordBreak("applepenapple", ["apple", "pen"]))  # Output: True
print(solution.wordBreak("catsandog", ["cats", "dog", "sand", "and", "cat"]))  # Output: False
```