**TOPIC-1**

1. Given an array of strings words, return the first palindromic string in the array. If there is no such string, return an empty string "". A string is palindromic if it reads the same forward and backward.

Example 1:

Input: words = ["abc","car","ada","racecar","cool"]

Output: "ada"

Explanation: The first string that is palindromic is "ada".

Note that "racecar" is also palindromic, but it is not the first.

Example 2:

Input: words = ["notapalindrome","racecar"]

Output: "racecar"

Explanation: The first and only string that is palindromic is "racecar".

**def find(words):**

**s=""**

**for w in words:**

**if ispalindrome(w)==True:**

**s=w**

**return s**

**def ispalindrome(w):**

**l=0**

**r=len(w)-1**

**while l<r:**

**if w[l]!=w[r]:**

**return False**

**l=l+1**

**r=r-1**

**return True**

**words=["abc","ada"]**

**print(find(words))**

1. You are given two integer arrays nums1 and nums2 of sizes n and m, respectively. Calculate the following values: answer1 : the number of indices i such that nums1[i] exists in nums2. answer2 : the number of indices i such that nums2[i] exists in nums1 Return [answer1,answer2].

Example 1:

Input: nums1 = [2,3,2], nums2 = [1,2]

Output: [2,1]

Explanation:

Example 2:

Input: nums1 = [4,3,2,3,1], nums2 = [2,2,5,2,3,6]

Output: [3,4]

Explanation:

The elements at indices 1, 2, and 3 in nums1 exist in nums2 as well. So answer1 is 3.

The elements at indices 0, 1, 3, and 4 in nums2 exist in nums1. So answer2 is 4.

**def check(n1,n2):**

**s1=0**

**s2=0**

**for i in n1:**

**if i in n2:**

**s1=s1+1**

**for j in n2:**

**if j in n1:**

**s2=s2+1**

**return(s1,s2)**

**n1=[2,3,2]**

**n2=[1,2]**

**print(check(n1,n2))**

1. You are given a 0-indexed integer array nums. The distinct count of a subarray of nums is defined as: Let nums[i..j] be a subarray of nums consisting of all the indices from i to j such that 0 <= i <= j < nums.length. Then the number of distinct values in nums[i..j] is called the distinct count of nums[i..j]. Return the sum of the squares of distinct counts of all subarrays of nums. A subarray is a contiguous non-empty sequence of elements within an array.

Example 1:

Input: nums = [1,2,1]

Output: 15

Explanation: Six possible subarrays are:

[1]: 1 distinct value

[2]: 1 distinct value

[1]: 1 distinct value

[1,2]: 2 distinct values

[2,1]: 2 distinct values

[1,2,1]: 2 distinct values

The sum of the squares of the distinct counts in all subarrays is equal to 12 + 12 + 12 + 22 + 22 + 22 = 15.

Example 2:

Input: nums = [1,1]

Output: 3

Explanation: Three possible subarrays are:

[1]: 1 distinct value

[1]: 1 distinct value

[1,1]: 1 distinct value

The sum of the squares of the distinct counts in all subarrays is equal to 12 + 12 + 12 = 3.

**def distinct(nums):**

**n=len(nums)**

**sum=0**

**for i in range(n):**

**for j in range(i,n):**

**sub=nums[i:j+1]**

**c=len(set(sub))**

**sum=sum+c\*\*2**

**return sum**

**nums=[1,2,1]**

**r=distinct(nums)**

**print(r)**

1. Given a 0-indexed integer array nums of length n and an integer k, return *the number of pairs* (i, j) *where* 0 <= i < j < n, *such that* nums[i] == nums[j] *and* (i \* j) *is divisible by* k.

Example 1:

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

Output: 4

Explanation:

There are 4 pairs that meet all the requirements:

- nums[0] == nums[6], and 0 \* 6 == 0, which is divisible by 2.

- nums[2] == nums[3], and 2 \* 3 == 6, which is divisible by 2.

- nums[2] == nums[4], and 2 \* 4 == 8, which is divisible by 2.

- nums[3] == nums[4], and 3 \* 4 == 12, which is divisible by 2.

Example 2:

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

Output: 0

Explanation: Since no value in nums is repeated, there are no pairs (i,j) that meet all the requirements.

**def number(num,k):**

**n=len(num)**

**c=0**

**for i in range(n):**

**for j in range(i+1,n):**

**if num[i]==num[j] and (i\*j)%k==0:**

**c=c+1**

**return c**

**num=[3,1,2,2,2,1,3]**

**k=2**

**print(number(num,k))**

1. Write a program FOR THE BELOW TEST CASES with least time complexity Test Cases: -
2. **Input:** {1, 2, 3, 4, 5} **Expected Output:** 5
3. **Input:** {7, 7, 7, 7, 7} **Expected Output:** 7
4. **Input:** {-10, 2, 3, -4, 5} **Expected Output:** 5

**def search(nums,target):**

**if target in nums:**

**return nums.index(target)**

**nums=[1,2,3,4,5]**

**target=5**

**print(search(nums,target))**

1. You have an algorithm that process a list of numbers. It firsts sorts the list using an efficient sorting algorithm and then finds the maximum element in sorted list. Write the code for the same.

Test Cases

1. Empty List
   * 1. Input: []
     2. Expected Output: None or an appropriate message indicating that the list is empty.
2. Single Element List
   * 1. Input: [5]
     2. Expected Output: 5
3. All Elements are the Same
   * 1. Input: [3, 3, 3, 3, 3]
     2. Expected Output: 3
4. Write a program that takes an input list of n numbers and creates a new list containing only the unique elements from the original list. What is the space complexity of the algorithm?

Test Cases

Some Duplicate Elements

* Input: [3, 7, 3, 5, 2, 5, 9, 2]
* Expected Output: [3, 7, 5, 2, 9] (Order may vary based on the algorithm used)

**def duplicate(nums):**

**duplicate\_set = set(nums)**

**duplicate\_list = list(duplicate\_set)**

**return duplicate\_list**

**input\_list = [3, 7, 3, 5, 2, 5, 9, 2]**

**result = duplicate(input\_list)**

**print("Actual Output:", result)**

1. Sort an array of integers using the bubble sort technique. Analyze its time complexity using Big-O notation. Write the code

**def bubblesort(arr):**

**n = len(arr)**

**for i in range(n):**

**for j in range(0, n - i - 1):**

**if arr[j] > arr[j + 1]:**

**arr[j], arr[j + 1] = arr[j + 1], arr[j]**

**arr = [5, 1, 4, 2, 8]**

**bubblesort(arr)**

**print("Sorted array is:", arr)**

1. Checks if a given number x exists in a sorted array arr using binary search. Analyze its time complexity using Big-O notation.

**Test Case:**

**Example X={ 3,4,6,-9,10,8,9,30} KEY=10**

**Output: Element 10 is found at position 5**

**def binary\_search(nums, target):**

**left, right = 0, len(nums) - 1**

**while left <= right:**

**mid = (left + right) // 2**

**if nums[mid] == target:**

**return mid**

**elif nums[mid] < target:**

**left = mid + 1**

**else:**

**right = mid - 1**

**return -1**

**numbers = [2, 3, 5, 7, 11, 13, 17, 19, 23]**

**target = 11**

**index = binary\_search(numbers, target)**

**print(f" element {target} found at index {index}.")**

1. Given an array of integers nums, sort the array in ascending order and return it. You must solve the problem without using any built-in functions in O(nlog(n)) time complexity and with the smallest space complexity possible.

**def merge\_sort(arr):**

**if len(arr) <= 1: return arr**

**mid = len(arr) // 2**

**left\_half = merge\_sort(arr[:mid])**

**right\_half = merge\_sort(arr[mid:])**

**return merge(left\_half, right\_half)**

**def merge(left, right):**

**sorted\_arr = []**

**while left and right:**

**sorted\_arr.append((left if left[0] <= right[0] else right).pop(0))**

**return sorted\_arr + left + right**

**arr = [5, 2, 9, 1, 5, 6]**

**sorted\_arr = merge\_sort(arr)**

**print(sorted\_arr)**

1. 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 security systems connected, and it will automatically contact the police if two adjacent houses were broken into on the same night.

Examples:

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

Output : The maximum money you can rob without alerting the

police is 3(robbing house 1).

**def rob(nums):**

**def rob\_linear(nums):**

**prev, curr = 0, 0**

**for num in nums:**

**prev, curr = curr, max(curr, prev + num)**

**return curr**

**if len(nums) == 1:**

**return nums[0]**

**return max(rob\_linear(nums[:-1]), rob\_linear(nums[1:]))**

**nums1 = [2, 3, 2]**

**print(rob(nums1))**

1. 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?

Examples:

1. **Input**: n=4 **Output**: 5
2. **Input**: n=3 **Output**: 3

**def climbStairs(n):**

**if n == 0:**

**return 1**

**if n == 1:**

**return 1**

**dp = [0] \* (n + 1)**

**dp[1] = 1**

**dp[2] = 2**

**for i in range(3, n + 1):**

**dp[i] = dp[i - 1] + dp[i - 2]**

**return dp[n]**

**print(climbStairs(4))**

1. A robot is located at the top-left corner of a m×n grid .The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid. How many possible unique paths are there?

Examples:

1. **Input**: m=7,n=3 **Output**: 28
2. **Input**: m=3,n=2 **Output**: 3

def uniquePaths(m, n):

dp = [[0] \* n for \_ in range(m)]

dp[0][0] = 1

for i in range(m):

for j in range(n):

if i > 0:

dp[i][j] += dp[i-1][j]

if j > 0:

dp[i][j] += dp[i][j-1]

return dp[m-1][n-1]

m = 7

n = 3

print(uniquePaths(m, n))

1. In a string S of lowercase letters, these letters form consecutive groups of the same character. For example, a string like s = "abbxxxxzyy" has the groups "a", "bb", "xxxx", "z", and "yy". A group is identified by an interval [start, end], where start and end denote the start and end indices (inclusive) of the group. In the above example, "xxxx" has the interval [3,6]. A group is considered large if it has 3 or more characters. Return the intervals of every large group sorted in increasing order by start index.

Example 1:

Input: s = "abbxxxxzzy"

Output: [[3,6]]

Explanation: "xxxx" is the only large group with start index 3 and end index 6.

**def largeGroupPositions(s):**

**result = []**

**i = 0**

**while i < len(s):**

**count = 1**

**while i + 1 < len(s) and s[i] == s[i + 1]:**

**count += 1**

**i += 1**

**if count >= 3:**

**result.append([i - count + 1, i])**

**i += 1**

**return result**

**s = "abbxxxxzzy"**

**print(largeGroupPositions(s))**

1. We stack glasses in a pyramid, where the first row has 1 glass, the second row has 2 glasses, and so on until the 100th row.  Each glass holds one cup of champagne. Then, some champagne is poured into the first glass at the top.  When the topmost glass is full, any excess liquid poured will fall equally to the glass immediately to the left and right of it.  When those glasses become full, any excess champagne will fall equally to the left Example 1:

Input: poured = 1, query\_row = 1, query\_glass = 1

Output: 0.00000

Explanation: We poured 1 cup of champange to the top glass of the tower (which is indexed as (0, 0)). There will be no excess liquid so all the glasses under the top glass will remain empty.

def champagneTower(poured, q\_row, q\_glass):

**dp = [[0] \* k for k in range(1, 102)]**

**dp[0][0] = poured**

**for r in range(100):**

**for c in range(r + 1):**

**if dp[r][c] > 1:**

**excess = (dp[r][c] - 1.0) / 2.0**

**dp[r + 1][c] += excess**

**dp[r + 1][c + 1] += excess**

**dp[r][c] = 1.0**

**return min(1, dp[q\_row][q\_glass])**

**poured = 1**

**q\_row = 1**

**q\_glass = 1**

**print(champagneTower(poured, q\_row, q\_glass))**

**TOPIC 2 : BRUTE FORCE**

* 1. Write a program to perform the following
  + An empty list
  + A list with one element
  + A list with all identical elements
  + A list with negative numbers

**Test Cases:**

1. **Input:** []
   * **Expected Output:** []
2. **Input:** [1]
   * **Expected Output:** [1]
3. **Input:** [7, 7, 7, 7]
   * **Expected Output:** [7, 7, 7, 7]
4. **Input:** [-5, -1, -3, -2, -4]
   * **Expected Output:** [-5, -4, -3, -2, -1]

**a=[-5,-1,-3,-2,-4]**

**if len(a)==0:**

**print("[]")**

**else:**

**for i in range(len(a)):**

**for j in range(len(a)):**

**if a[j]>a[i]:**

**temp=a[i]**

**a[i]=a[j]**

**a[j]=temp**

**print(a)**

* 1. Describe the Selection Sort algorithm's process of sorting an array. Selection Sort works by dividing the array into a sorted and an unsorted region. Initially, the sorted region is empty, and the unsorted region contains all elements. The algorithm repeatedly selects the smallest element from the unsorted region and swaps it with the leftmost unsorted element, then moves the boundary of the sorted region one element to the right. Explain why Selection Sort is simple to understand and implement but is inefficient for large datasets. Provide examples to illustrate step-by-step how Selection Sort rearranges the elements into ascending order, ensuring clarity in your explanation of the algorithm's mechanics and effectiveness.

**Sorting a Random Array**:

**Input**: [5, 2, 9, 1, 5, 6]

**Output**: [1, 2, 5, 5, 6, 9]

**def selection\_sort(arr):**

**n = len(arr)**

**for i in range(n):**

**min\_index = i**

**for j in range(i + 1, n):**

**if arr[j] < arr[min\_index]:**

**min\_index = j**

**arr[i], arr[min\_index] = arr[min\_index], arr[i]**

**return arr**

**arr = [5, 2, 9, 1, 5, 6]**

**sorted\_arr = selection\_sort(arr)**

**print("Sorted array:", sorted\_arr)**

* 1. Write code to modify bubble\_sort function to stop early if the list becomes sorted before all passes are completed.

**a=[5,3,9,8,7,1]**

**n=len(a)**

**for i in range(n-1):**

**for j in range(i+1,n):**

**if a[i]>a[j]:**

**temp=a[i]**

**a[i]=a[j]**

**a[j]=temp**

**print(a)**

* 1. **Test Cases:**
* Test your optimized function with the following lists:
  1. **Input:** [64, 25, 12, 22, 11]
     + **Expected Output:** [11, 12, 22, 25, 64]
  2. **Input:** [29, 10, 14, 37, 13]
     + **Expected Output:** [10, 13, 14, 29, 37]
  3. **Input:** [3, 5, 2, 1, 4]
     + **Expected Output:** [1, 2, 3, 4, 5]
  4. **Input:** [1, 2, 3, 4, 5] (Already sorted list)
     + **Expected Output:** [1, 2, 3, 4, 5]
  5. **Input:** [5, 4, 3, 2, 1] (Reverse sorted list)
     + **Expected Output:** [1, 2, 3, 4, 5]

1. Write code for Insertion Sort that manages arrays with duplicate elements during the sorting process. Ensure the algorithm's behavior when encountering duplicate values, including whether it preserves the relative order of duplicates and how it affects the overall sorting outcome.

**Examples:**

**1. Array with Duplicates**:

* + **Input**: [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
  + **Output**: [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]

**def insertion\_sort(arr):**

**for i in range(1, len(arr)):**

**key = arr[i]**

**j = i - 1**

**while j >= 0 and arr[j] > key:**

**arr[j + 1] = arr[j]**

**j -= 1**

**arr[j + 1] = key**

**arr = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]**

**insertion\_sort(arr)**

**print("Sorted array:", arr)**

* 1. Given an array arr of positive integers sorted in a strictly increasing order, and an integer k. return the kth positive integer that is missing from this array.

Example 1:

Input: arr = [2,3,4,7,11], k = 5

Output: 9

Explanation: The missing positive integers are [1,5,6,8,9,10,12,13,...]. The 5th missing positive integer is 9.

**arr=[1,2,3,4]**

**k=2**

**f=0**

**n=1**

**c=0**

**while f==0:**

**if n not in arr:**

**c+=1**

**if c==k:**

**print(n)**

**f=1**

**n+=1**

* 1. A peak element is an element that is strictly greater than its neighbors. Given a 0-indexed integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to any of the peaks. You may imagine that nums[-1] = nums[n] = -∞. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array. You must write an algorithm that runs in O(log n) time.

Example 1:

Inp ut: nums = [1,2,3,1]

Output: 2

Explanation: 3 is a peak element and your function should return the index number 2.

**a=[1,2,3,1]**

**maxi=a[0]**

**index=0**

**for i in range (0,len(a)):**

**if a[i]>maxi:**

**maxi=a[i]**

**index=i**

**print(index)**

* 1. Given two strings needle and haystack, return the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

Example 1:

Input: haystack = "sadbutsad", needle = "sad"

Output: 0

Explanation: "sad" occurs at index 0 and 6.

The first occurrence is at index 0, so we return 0.

**def str\_str(haystack, needle):**

**if not needle:**

**return 0**

**n = len(haystack)**

**m = len(needle)**

**for i in range(n - m + 1):**

**if haystack[i:i + m] == needle:**

**return i**

**return -1**

**haystack = "sadbutsad"**

**needle = "sad"**

print(str\_str(haystack, needle)) # Output: 0

* 1. Given an array of string words, return all strings in words that is a substring of another word. You can return the answer in any order. A substring is a contiguous sequence of characters within a string

Example 1:

Input: words = ["mass","as","hero","superhero"]

Output: ["as","hero"]

def string(words):

result = []

for i in range(len(words)):

for j in range(len(words)):

if i != j and words[j] in words[i]:

result.append(words[j])

break

return result

words = ["mass", "as", "hero", "superhero"]

print(string(words))

* 1. Write a program that finds the closest pair of points in a set of 2D points using the brute force approach.

Input:

* A list or array of points represented by coordinates (x, y).

Points: [(1, 2), (4, 5), (7, 8), (3, 1)]

Output:

* The two points with the minimum distance between them.
* The minimum distance itself.

Closest pair: (1, 2) - (3, 1) Minimum distance: 1.4142135623730951

points = [(2, 3), (12, 30), (40, 50), (5, 1), (12, 10), (3, 4)]

**points.sort()**

**n = len(points)**

**min\_distance = float('inf')**

**for i in range(n):**

**for j in range(i+1, n):**

**d = ((points[i][0] - points[j][0])\*\*2 + (points[i][1] - points[j][1])\*\*2)\*\*0.5**

**if d < min\_distance:**

**min\_distance = d**

**min\_point1 = points[i]**

**min\_point2 = points[j]**

**print("Closest pair:", min\_point1, min\_point2)**

**print("Distance:", min\_distance**

* 1. Write a program that finds the convex hull of a set of 2D points using the brute force approach.

**Input:**

* A list or array of points represented by coordinates (x, y).

Points: [(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]

**Output:**

* The list of points that form the convex hull in counter-clockwise order.

Convex Hull: [(0, 0), (1, 1), (8, 1), (4, 6)]

**def orientation(p, q, r):**

**val = (q[1] - p[1]) \* (r[0] - q[0]) - (q[0] - p[0]) \* (r[1] - q[1])**

**if val == 0:**

**return 0**

**elif val > 0:**

**return 1**

**else:**

**return 2**

**def convex\_hull(points):**

**n = len(points)**

**if n < 3:**

**return points**

**points = sorted(points)**

**hull = []**

**for p in points:**

**while len(hull) >= 2 and orientation(hull[-2], hull[-1], p) != 2:**

**hull.pop()**

**hull.append(p)**

**for p in reversed(points[:-1]):**

**while len(hull) >= 2 and orientation(hull[-2], hull[-1], p) != 2:**

**hull.pop()**

**hull.append(p)**

**hull = list(dict.fromkeys(hull))**

**return hull**

**points = [(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]**

**convex\_hull\_points = convex\_hull(points)**

**print("Convex hull points:", convex\_hull\_points)**

* 1. You are given a list of cities represented by their coordinates. Develop a program that utilizes exhaustive search to solve the TSP. The program should:

1. Define a function distance(city1, city2) to calculate the distance between two cities (e.g., Euclidean distance).
2. **Simple Case:** Four cities with basic coordinates (e.g., [(1, 2), (4, 5), (7, 1), (3, 6)])
3. **More Complex Case:** Five cities with more intricate coordinates (e.g., [(2, 4), (8, 1), (1, 7), (6, 3), (5, 9)])

**Output:**

**Test Case 1:**

Shortest Distance: 7.0710678118654755

Shortest Path: [(1, 2), (4, 5), (7, 1), (3, 6), (1, 2)]

**Test Case 2:**

Shortest Distance: 14.142135623730951

Shortest Path: [(2, 4), (1, 7), (6, 3), (5, 9), (8, 1), (2, 4)]

**def distance(city1, city2):**

**return ((city2[0] - city1[0])\*2 + (city2[1] - city1[1])2)\*0.5**

**def tsp(cities):**

**n = len(cities)**

**shortest\_path = None**

**min\_distance = float('inf')**

**def generate\_permutations(cities):**

**if len(cities) <= 1:**

**return [cities]**

**permutations = []**

**for i, city in enumerate(cities):**

**rest = cities[:i] + cities[i+1:]**

**for perm in generate\_permutations(rest):**

**permutations.append([city] + perm)**

**return permutations**

**for perm in generate\_permutations(cities[1:]):**

**path = [cities[0]] + list(perm) + [cities[0]]**

**total\_distance = sum(distance(path[i], path[i+1]) for i in range(n))**

**if total\_distance < min\_distance:**

**min\_distance = total\_distance**

**shortest\_path = path**

**return min\_distance, shortest\_path**

**cities1 = [(1, 2), (4, 5), (7, 1), (3, 6)]**

**cities2 = [(2, 4), (8, 1), (1, 7), (6, 3), (5, 9)]**

**min\_distance1, shortest\_path1 = tsp(cities1)**

**min\_distance2, shortest\_path2 = tsp(cities2)**

**print("Test Case 1:")**

**print(f"Shortest Distance: {min\_distance1}")**

**print(f"Shortest Path: {shortest\_path1}\n")**

**print("Test Case 2:")**

**print(f"Shortest Distance: {min\_distance2}")**

**print(f"Shortest Path: {shortest\_path2}")**

* 1. You are given a cost matrix where each element cost[i][j] represents the cost of assigning worker i to task j. Develop a program that utilizes exhaustive search to solve the assignment problem. The program should Define a function total\_cost(assignment, cost\_matrix) that takes an assignment (list representing worker-task pairings) and the cost matrix as input. It iterates through the assignment and calculates the total cost by summing the corresponding costs from the cost matrix Implement a function assignment\_problem(cost\_matrix) that takes the cost matrix as input and performs the following Generate all possible permutations of worker indices (excluding repetitions).

**Test Cases:**

**Input**

1. **Simple Case:** Cost Matrix:

[[3, 10, 7],

[8, 5, 12],

[4, 6, 9]]

1. **More Complex Case:** Cost Matrix:

[[15, 9, 4],

[8, 7, 18],

[6, 12, 11]]

Output:

**Test Case 1:**

Optimal Assignment: [(worker 1, task 2), (worker 2, task 1), (worker 3, task 3)]

Total Cost: 19

**Test Case 2:**

Optimal Assignment: [(worker 1, task 3), (worker 2, task 1), (worker 3, task 2)]

Total Cost: 24

import itertools

def total\_cost(assignment, cost\_matrix):

return sum(cost\_matrix[worker][task] for worker, task in enumerate(assignment))

def assignment\_problem(cost\_matrix):

num\_workers = len(cost\_matrix)

num\_tasks = len(cost\_matrix[0]) if cost\_matrix else 0

min\_cost = float('inf')

best\_assignment = []

for perm in itertools.permutations(range(num\_tasks)):

current\_cost = total\_cost(perm, cost\_matrix)

if current\_cost < min\_cost:

min\_cost = current\_cost

best\_assignment = list(perm)

return min\_cost, best\_assignment

# Example usage:

cost\_matrix = [

[9, 2, 7, 8],

[6, 4, 3, 7],

[5, 8, 1, 8],

[7, 6, 9, 4]

]

min\_cost, best\_assignment = assignment\_problem(cost\_matrix)

print("Minimum Cost:", min\_cost)

print("Best Assignment:", best\_assignment)

* 1. You are given a list of items with their weights and values. Develop a program that utilizes exhaustive search to solve the 0-1 Knapsack Problem. The program should:

1. Define a function total\_value(items, values) that takes a list of selected items (represented by their indices) and the value list as input. It iterates through the selected items and calculates the total value by summing the corresponding values from the value list.

**Test Cases:**

1. **Simple Case:**

* Items: 3 (represented by indices 0, 1, 2)
* Weights: [2, 3, 1]
* Values: [4, 5, 3]
* Capacity: 4

1. **More Complex Case:**

* Items: 4 (represented by indices 0, 1, 2, 3)
* Weights: [1, 2, 3, 4]
* Values: [2, 4, 6, 3]
* Capacity: 6

Output:

**Test Case 1:**

Optimal Selection: [0, 2] (Items with indices 0 and 2)

Total Value: 7

**Test Case 2:**

Optimal Selection: [0, 1, 2] (Items with indices 0, 1, and 2)

Total Value: 10

**def knapsack\_brute\_force(weights, values, capacity):**

**n = len(weights)**

**max\_value = 0**

**best\_set = []**

**def recursive\_subset(index, current\_weight, current\_value, selected\_items):**

**nonlocal max\_value, best\_set**

**if index == n:**

**if current\_weight <= capacity and current\_value > max\_value:**

**max\_value = current\_value**

**best\_set = selected\_items[:]**

**return**

**selected\_items.append(index)**

**recursive\_subset(index + 1, current\_weight + weights[index], current\_value + values[index], selected\_items)**

**selected\_items.pop()**

**recursive\_subset(index + 1, current\_weight, current\_value, selected\_items)**

**recursive\_subset(0, 0, 0, [])**

**return max\_value, best\_set**

**weights = [2, 3, 4, 5]**

**values = [3, 4, 5, 6]**

**capacity = 8**

**max\_value, best\_items = knapsack\_brute\_force(weights, values, capacity)**

**print("Maximum value:", max\_value)**

**print("Selected items indices:", best\_items)**

**TOPIC 3 : DIVIDE AND CONQUER**

1. Write a Program to find both the maximum and minimum values in the array. Implement using any programming language of your choice. Execute your code and provide the maximum and minimum values found.

Input : N= 8, a[] = {5,7,3,4,9,12,6,2}

Output : Min = 2, Max = 12

**def find\_max\_min(arr):**

**if not arr:**

**return None, None**

**max\_value = arr[0]**

**min\_value = arr[0]**

**for num in arr:**

**if num > max\_value:**

**max\_value = num**

**if num < min\_value:**

**min\_value = num**

**return max\_value, min\_value**

**# Example usage:**

**arr = [3, 7, 2, 9, 1, 4, 6, 8, 5]**

**max\_val, min\_val = find\_max\_min(arr)**

**print(f"The maximum value in the array is: {max\_val}")**

**print(f"The minimum value in the array is: {min\_val}")**

1. You are given an unsorted array 31,23,35,27,11,21,15,28. Write a program for Merge Sort and implement using any programming language of your choice.

Test Cases :

Input : N= 8, a[] = {31,23,35,27,11,21,15,28}

Output : 11,15,21,23,27,28,31,35

**def merge\_sort(arr):**

**if len(arr) <= 1:**

**return arr**

**mid = len(arr) // 2**

**left\_sorted = merge\_sort(arr[:mid])**

**right\_sorted = merge\_sort(arr[mid:])**

**return merge(left\_sorted, right\_sorted)**

**def merge(left, right):**

**sorted\_arr = []**

**while left and right:**

**sorted\_arr.append(left.pop(0) if left[0] <= right[0] else right.pop(0))**

**return sorted\_arr + left + right**

# Test case

arr = [31, 23, 35, 27, 11, 21, 15, 28]

sorted\_arr = merge\_sort(arr)

print("Sorted array:", sorted\_arr)

1. Given an unsorted array 10,16,8,12,15,6,3,9,5 Write a program to perform Quick Sort. Choose the first element as the pivot and partition the array accordingly. Show the array after this partition. Recursively apply Quick Sort on the sub-arrays formed. Display the array after each recursive call until the entire array is sorted.

Input : N= 9, a[]= {10,16,8,12,15,6,3,9,5}

Output : 3,5,6,8,9,10,12,15,16

**def quick\_sort(arr, low, high):**

**if low < high:**

**pivot\_index = partition(arr, low, high)**

**quick\_sort(arr, low, pivot\_index - 1)**

**quick\_sort(arr, pivot\_index + 1, high)**

**def partition(arr, low, high):**

**pivot = arr[low]**

**left = low + 1**

**right = high**

**done = False**

**while not done:**

**while left <= right and arr[left] <= pivot:**

**left += 1**

**while arr[right] >= pivot and right >= left:**

**right -= 1**

**if right < left:**

**done = True**

**else:**

**arr[left], arr[right] = arr[right], arr[left]**

**arr[low], arr[right] = arr[right], arr[low]**

**return right**

**arr = [10, 16, 8, 12, 15, 6, 3, 9, 5]**

**print("Original array:", arr)**

**quick\_sort(arr, 0, len(arr) - 1)**

**print("Sorted array:", arr)**

Output : 13,17,22,25,34,36,43,52,65,67

1. Implement the Binary Search algorithm in a programming language of your choice and test it on the array 5,10,15,20,25,30,35,40,45 to find the position of the element 20.Execute your code and provide the index of the element 20. Modify your implementation to count the number of comparisons made during the search process. Print this count along with the result.

Input : N= 9, a[] = {5,10,15,20,25,30,35,40,45}, search key = 20

Output : 4

**def b\_s(arr, key):**

**left = 0**

**right = len(arr) - 1**

**comparisons = 0**

**while left <= right:**

**mid = (left + right) // 2**

**comparisons += 1**

**if arr[mid] == key:**

**return mid, comparisons**

**elif arr[mid] < key:**

**left = mid + 1**

**else:**

**right = mid - 1**

**return -1, comparisons**

**arr = [5, 10, 15, 20, 25, 30, 35, 40, 45]**

**key = 20**

**index, comparisons = b\_s(arr, key)**

**if index != -1:**

**print(f"Element {key} found at index {index}.")**

**else:**

**print(f"Element {key} not found in the array.")**

**print(f"Number of comparisons made: {comparisons}")Output : 2**

Output :

1. Given an array of points where points[i] = [xi, yi] represents a point on the X-Y plane and an integer k, return the k closest points to the origin (0, 0).
2. Input : points = [[**1**,**3**],[-**2**,**2**],[**5**,**8**],[**0**,**1**]],k=2

Output:[[-2, 2], [0, 1]]

**def distance(point):**

**return point[0]\*\*2 + point[1]\*\*2**

**def k\_closest\_points(points, k):**

**distances = [(distance(point), point) for point in points]**

**distances.sort()**

**closest\_points = [distances[i][1] for i in range(k)]**

**return closest\_points**

**points = [[1, 3], [-2, 2], [5, 8], [0, 1]]**

**k = 2**

**closest = k\_closest\_points(points, k)**

**print(f"The {k} closest points to the origin are: {closest}")**

1. Given four lists A, B, C, D of integer values,Write a program to compute how many tuples n(i, j, k, l) there are such that A[i] + B[j] + C[k] + D[l] is zero.
2. **Input**: A = [1, 2], B = [-2, -1], C = [-1, 2], D = [0, 2]

**Output**: 2

**def four\_list(A, B, C, D):**

**n = len(A)**

**s\_count = [0] \* (2 \* n + 1)**

**for numA in A:**

**for numB in B:**

**s\_count[numA + numB + n] += 1**

**count\_tuples = 0**

**for numC in C:**

**for numD in D:**

**complement = -(numC + numD) + n**

**if 0 <= complement <= 2 \* n:**

**count\_tuples += s\_count[complement]**

**return count\_tuples**

**A = [1, 2]**

**B = [-2, -1]**

**C = [-1, 2]**

**D = [0, 2]**

**result = four\_list(A, B, C, D)**

**print("Number of tuples:", result)**

1. To Implement the Median of Medians algorithm ensures that you handle the worst-case time complexity efficiently while finding the k-th smallest element in an unsorted array.

arr = [12, 3, 5, 7, 19] k = 2 Expected Output:5

**def find\_kth\_smallest(arr, k):**

**if len(arr) <= 5:**

**return sorted(arr)[k - 1]**

**groups = [arr[i:i+5] for i in range(0, len(arr), 5)]**

**medians = [sorted(group)[len(group) // 2] for group in groups]**

**pivot = find\_kth\_smallest(medians, len(medians) // 2 + 1)**

**left = [x for x in arr if x < pivot]**

**right = [x for x in arr if x > pivot]**

**if len(left) >= k:**

**return find\_kth\_smallest(left, k)**

**elif len(left) + 1 == k:**

**return pivot**

**else:**

**return find\_kth\_smallest(right, k - len(left) - 1)**

**arr = [12, 3, 5, 7, 19]**

**k = 2**

**print(find\_kth\_smallest(arr, k))**

1. Write a program to implement Meet in the Middle Technique. Given a large array of integers and an exact sum E, determine if there is any subset that sums exactly to E. Utilize the Meet in the Middle technique to handle the potentially large size of the array. Return true if there is a subset that sums exactly to E, otherwise return false.

a) E = {1, 3, 9, 2, 7, 12} exact Sum = 15

**def s\_sum(arr):**

**n = len(arr)**

**s = []**

**for i in range(1 << n):**

**subset\_sum = 0**

**subset = []**

**for j in range(n):**

**if i & (1 << j):**

**subset\_sum += arr[j]**

**subset.append(arr[j])**

**s.append((subset\_sum, subset))**

**return s**

**def mm(arr, E):**

**n = len(arr)**

**mid = n // 2**

**s1 = s\_sum(arr[:mid])**

**s2 = s\_sum(arr[mid:])**

**for sum1, subset1 in s1:**

**if sum1 == E:**

**return True**

**for sum2, subset2 in s2:**

**if sum2 == E:**

**return True**

**for sum1, subset1 in s1:**

**required\_sum = E - sum1**

**for sum2, subset2 in s2:**

**if sum2 == required\_sum:**

**return True**

**return False**

**arr = [1, 3, 9, 2, 7, 12]**

**exact\_sum = 15**

**result = mm(arr, exact\_sum)**

**if result:**

**print(f"There exists a subset that sums exactly to {exact\_sum}.")**

**else:**

**print(f"No subset sums exactly to {exact\_sum}.")**

15 Given two 2×2 Matrices A and B

A=(1 7​ B=( 1 3

3 5​) 7 5)

Use Strassen's matrix multiplication algorithm to compute the product matrix C such that C=A×B.

**def strassen(A, B):**

**a11, a12, a21, a22 = A[0][0], A[0][1], A[1][0], A[1][1]**

**b11, b12, b21, b22 = B[0][0], B[0][1], B[1][0], B[1][1]**

**P1 = a11 \* (b12 - b22)**

**P2 = (a11 + a12) \* b22**

**P3 = (a21 + a22) \* b11**

**P4 = a22 \* (b21 - b11)**

**P5 = (a11 + a22) \* (b11 + b22)**

**P6 = (a12 - a22) \* (b21 + b22)**

**P7 = (a11 - a21) \* (b11 + b12)**

**c11 = P5 + P4 - P2 + P6**

**c12 = P1 + P2**

**c21 = P3 + P4**

**c22 = P5 + P1 - P3 - P7**

**C = [[c11, c12], [c21, c22]]**

**return C**

**A = [[1, 7], [3, 5]]**

**B = [[1, 3], [7, 5]]**

**C = strassen(A, B)**

**print("Matrix C =")**

**for row in C:**

**print(row)**

16 Given two integers X=1234 and Y=5678: Use the Karatsuba algorithm to compute the product Z=X x Y

**Test Case 1:**

Input: x=1234,y=5678

Expected Output: z=1234×5678=7016652

**def karatsuba\_multiply(X, Y):**

**if X < 10 or Y < 10:**

**return X \* Y**

**X\_str = str(X)**

**Y\_str = str(Y)**

**n = max(len(X\_str), len(Y\_str))**

**m = n // 2**

**high1, low1 = int(X\_str[:-m] or 0), int(X\_str[-m:] or 0)**

**high2, low2 = int(Y\_str[:-m] or 0), int(Y\_str[-m:] or 0)**

**z0 = karatsuba\_multiply(low1, low2)**

**z1 = karatsuba\_multiply((low1 + high1), (low2 + high2))**

**z2 = karatsuba\_multiply(high1, high2)**

**intermediate = z1 - z2 - z0**

**Z = z2 \* 10\*\*(2\*m) + intermediate \* 10\*\*m + z0**

**return Z**

**X = 1234**

**Y = 5678**

**Z = karatsuba\_multiply(X, Y)**

**print(f"The product of {X} and {Y} using Karatsuba algorithm is: {Z}")**

**TOPIC 4 : DYNAMIC PROGRAMMING**

* 1. You are given the number of sides on a die (num\_sides), the number of dice to throw (num\_dice), and a target sum (target). Develop a program that utilizes dynamic programming to solve the Dice Throw Problem.

Test Cases:

1.Simple Case:

•Number of sides: 6

•Number of dice: 2

•Target sum: 7

def count\_ways(ndice, nside, target, memo={}):

if ndice == 0:

return 1 if target == 0 else 0

if target < 0:

return 0

if (ndice, target) in memo:

return memo[(ndice, target)]

total\_ways = 0

for side in range(1, nside + 1):

total\_ways += count\_ways(ndice - 1, nside, target - side, memo)

memo[(ndice, target)] = total\_ways

return total\_ways

nside = 6

ndice = 2

target = 7

result = count\_ways(ndice, nside, target)

print(f"Number of ways to get sum {target} with {ndice} dice of {nside} sides: {result}")

* 1. In a factory, there are two assembly lines, each with n stations. Each station performs a specific task and takes a certain amount of time to complete. The task must go through each station in order, and there is also a transfer time for switching from one line to another. Given the time taken at each station on both lines and the transfer time between the lines, the goal is to find the minimum time required to process a product from start to end.

def schedule(n, a1, a2, t1, t2, e1, e2, x1, x2):

f1 = [0] \* (n + 1)

f2 = [0] \* (n + 1)

f1[1] = e1 + a1[0]

f2[1] = e2 + a2[0]

for i in range(2, n + 1):

f1[i] = min(f1[i-1] + a1[i-1], f2[i-1] + t2[i-2] + a1[i-1])

f2[i] = min(f2[i-1] + a2[i-1], f1[i-1] + t1[i-2] + a2[i-1])

min\_time = min(f1[n] + x1, f2[n] + x2)

return min\_time

n = 4

a1 = [4, 5, 3, 2]

a2 = [2, 10, 1, 4]

t1 = [0, 7, 4, 5]

t2 = [0, 9, 2, 8]

e1 = 10

e2 = 12

x1 = 18

x2 = 7

min\_time = schedule(n, a1, a2, t1, t2, e1, e2, x1, x2)

print(f"The minimum time required to process the product: {min\_time}")

* 1. Write a python program to find the minimum path distance by using matrix form.

Test Cases:

1)

{0,10,15,20}

{10,0,35,25}

{15,35,0,30}

{20,25,30,0}

Output: 80

def min(matrix):

n = len(matrix)

dist = [row[:] for row in matrix]

for k in range(n):

for i in range(n):

for j in range(n):

if dist[i][j] > dist[i][k] + dist[k][j]:

dist[i][j] = dist[i][k] + dist[k][j]

return dist[0][n-1]

matrix = [

[0, 10, 15, 20],

[10, 0, 35, 25],

[15, 35, 0, 30],

[20, 25, 30, 0]

]

result = min(matrix)

print(f"The minimum path distance: {result}")

* 1. Assume you are solving the Traveling Salesperson Problem for 4 cities (A, B, C, D) with known distances between each pair of cities. Now, you need to add a fifth city (E) to the problem.

Distances:

A-B: 10, A-C: 15, A-D: 20, A-E: 25 B-C: 35, B-D: 25, B-E: 30 C-D: 30, C-E: 20 D-E: 15

import sys

def tsp(distances):

n = len(distances)

INF = sys.maxsize

# Initialize the DP table with infinity

dp = [[INF] \* n for \_ in range(1 << n)]

# Base case: Starting at any city and visiting only that city

for i in range(n):

dp[1 << i][i] = 0

# Fill the DP table

for mask in range(1 << n):

for i in range(n):

if mask & (1 << i): # Check if city 'i' is in the current subset 'mask'

for j in range(n):

if i != j and mask & (1 << j): # Check if city 'j' is also in 'mask'

dp[mask][i] = min(dp[mask][i], dp[mask ^ (1 << i)][j] + distances[j][i])

# Finding the minimum cost to return to the starting city

min\_cost = INF

for i in range(n):

if dp[(1 << n) - 1][i] < INF: # If all cities are visited in the full mask

min\_cost = min(min\_cost, dp[(1 << n) - 1][i] + distances[i][0]) # Add the cost to return to city '0'

return min\_cost

# Test Case: Distances between cities A, B, C, D, E

distances = [

[0, 10, 15, 20, 25],

[10, 0, 35, 25, 30],

[15, 35, 0, 30, 20],

[20, 25, 30, 0, 15],

[25, 30, 20, 15, 0]

]

min\_cost = tsp(distances)

print(f"The minimum cost of the Traveling Salesperson Problem: {min\_cost}").

* 1. 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.

def palindrom(s):

if len(s) < 2:

return s

start = 0

max\_len = 1

def m(left, right):

nonlocal start, max\_len

while left >= 0 and right < len(s) and s[left] == s[right]:

c\_len = right - left + 1

if c\_len > max\_len:

max\_len = c\_len

start = left

left -= 1

right += 1

for i in range(len(s)):

m(i, i)

m(i, i + 1)

return s[start:start + max\_len]

s = "babad"

print(palindrom(s))

* 1. Given a string s, find the length of the longest substring without repeating characters.

Example 1: Input: s = "abcabcbb" Output: 3

Explanation: The answer is "abc", with the length of 3.

def lengthOfLongestSubstring(s):

char\_index = {}

max\_length = start = 0

for i, char in enumerate(s):

if char in char\_index and start <= char\_index[char]:

start = char\_index[char] + 1

else:

max\_length = max(max\_length, i - start + 1)

char\_index[char] = i

return max\_length

input\_string = input("Enter a string: ")

print(f"The length of the longest substring without repeating characters is: {lengthOfLongestSubstring(input\_string)}")

* 1. 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

def wordBreak(s, wordDict):

n = len(s)

dp = [False] \* (n + 1)

dp[0] = True

for i in range(1, n + 1):

for word in wordDict:

if dp[i - len(word)] and s[i - len(word):i] == word:

dp[i] = True

break

return dp[n]

s = "leetcode"

wordDict = ["leet", "code"]

print(wordBreak(s, wordDict))

* 1. Given an input string and a dictionary of words, find out if the input string can be segmented into a space-separated sequence of dictionary words.Consider the following dictionary { i, like, sam, sung, samsung, mobile, ice, cream, icecream, man, go, mango}

Input: ilike

Output: Yes

The string can be segmented as "i like".

def wordBreak(s, wordDict):

n = len(s)

dp = [False] \* (n + 1)

dp[0] = True

for i in range(1, n + 1):

for j in range(i):

if dp[j] and s[j:i] in wordDict:

dp[i] = True

break

return dp[n]

dictionary = {"i", "like", "sam", "sung", "samsung", "mobile", "ice", "cream", "icecream", "man", "go", "mango"}

s = "ilike"

print(wordBreak(s, dictionary)) # Output: True

* 1. Implement Floyd's Algorithm to find the shortest path between all pairs of cities. Display the distance matrix before and after applying the algorithm. Identify and print the shortest path

Input: n = 4, edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]], distanceThreshold = 4

Output: 3

* 1. Implement the Optimal Binary Search Tree algorithm for the keys A,B,C,D with frequencies 0.1,0.2,0.4,0.3 Write the code using any programming language to construct the OBST for the given keys and frequencies. Execute your code and display the resulting OBST and its cost. Print the cost and root matrix.

Input N =4, Keys = {A,B,C,D} Frequencies = {01.02.,0.3,0.4}

Output : 1.7

def optCost(freq, i, j):

if j < i:

return 0

if j == i:

return freq[i]

fsum = Sum(freq, i, j)

Min = 999999999999

for r in range(i, j + 1):

cost = (optCost(freq, i, r - 1) +

optCost(freq, r + 1, j))

if cost < Min:

Min = cost

return Min + fsum

def optimalSearchTree(keys, freq, n):

return optCost(freq, 0, n - 1)

def Sum(freq, i, j):

s = 0

for k in range(i, j + 1):

s += freq[k]

return s

if \_\_name\_\_ == '\_\_main\_\_':

keys = [10, 12]

freq = [34, 50]

n = len(keys)

print("Cost of Optimal BST is",

optimalSearchTree(keys, freq, n))

* 1. A game on an undirected graph is played by two players, Mouse and Cat, who alternate turns. The graph is given as follows: graph[a] is a list of all nodes b such that ab is an edge of the graph. The mouse starts at node 1 and goes first, the cat starts at node 2 and goes second, and there is a hole at node 0. During each player's turn, they must travel along one edge of the graph that meets where they are. For example, if the Mouse is at node 1, it must travel to any node in graph[1]. Additionally, it is not allowed for the Cat to travel to the Hole (node 0).Then, the game can end in three ways:

def catMouseGame(graph):

n = len(graph)

dp = [[[None, None] for \_ in range(n)] for \_ in range(n)]

# Initialize the base cases

for c in range(n):

dp[0][c][0] = 1 # Mouse wins if it reaches the hole

for m in range(1, n):

dp[m][m][1] = 2 # Cat wins if it catches mouse at the same position

queue = []

for m in range(n):

for c in range(1, n): # Cat cannot start at hole (node 0)

if dp[m][c][0] is not None:

queue.append((m, c, 0))

if dp[m][c][1] is not None:

queue.append((m, c, 1))

while queue:

m, c, turn = queue.pop(0)

if turn == 0: # Mouse's turn

for next\_m in graph[m]:

if dp[next\_m][c][1] is None: # Mouse can move to an undecided state

if dp[m][c][0] == 1:

dp[next\_m][c][0] = 1 # Mouse wins

queue.append((next\_m, c, 0))

elif all(dp[next\_m][c][1] == 1 for c in graph[c]):

dp[next\_m][c][0] = 0 # Draw

queue.append((next\_m, c, 0))

else: # Cat's turn

for next\_c in graph[c]:

if next\_c != 0: # Cat cannot move to hole

if dp[m][next\_c][0] is None: # Cat can move to an undecided state

if dp[m][c][1] == 2:

dp[m][next\_c][1] = 2 # Cat wins

queue.append((m, next\_c, 1))

elif all(dp[m][new\_c][0] == 2 for new\_c in graph[c]):

dp[m][next\_c][1] = 0 # Draw

queue.append((m, next\_c, 1))

# Check the final state (1, 2, 0) to determine the result

if dp[1][2][0] == 1:

return 1 # Mouse wins

elif dp[1][2][1] == 2:

return 2 # Cat wins

else:

return 0 # Draw

# Test Input

graph = [[2,5],[3],[0,4,5],[1,4,5],[2,3],[0,2,3]]

print(catMouseGame(graph)) # Output: 0

* 1. 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:

START

FINISH

Input: m = 3, n = 7

Output: 28

def uniquePaths(m, n):

# Initialize a 2D array for DP

dp = [[0] \* n for \_ in range(m)]

# Base case: there is 1 way to be at the starting position

dp[0][0] = 1

# Fill the DP table

for i in range(m):

for j in range(n):

if i == 0 and j == 0:

continue # Skip the starting position

if i == 0:

dp[i][j] = dp[i][j-1] # Only one way to reach from the left

elif j == 0:

dp[i][j] = dp[i-1][j] # Only one way to reach from above

else:

dp[i][j] = dp[i-1][j] + dp[i][j-1] # Sum of ways from left and above

# The answer is the number of unique paths to reach dp[m-1][n-1]

return dp[m-1][n-1]

# Test the function

m = 3

n = 7

print(uniquePaths(m, n)) # Output: 28

* 1. Given an array of integers nums, return the number of good pairs. A pair (i, j) is called good if nums[i] == nums[j] and i < j.

Example 1:

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

Output: 4

Explanation: There are 4 good pairs (0,3), (0,4), (3,4), (2,5) 0-indexed.

def n(nums):

count = {}

pair = 0

for num in nums:

if num in count:

count[num] += 1

else:

count[num] = 1

for num in count:

if count[num] > 1:

pair += count[num] \* (count[num] - 1) // 2

return pair

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

print(n(nums))

**TOPIC 5 : GREEDY**

* 1. **T**here are 3n piles of coins of varying size, you and your friends will take piles of coins as follows: In each step, you will choose any 3 piles of coins (not necessarily consecutive). Of your choice, Alice will pick the pile with the maximum number of coins. You will pick the next pile with the maximum number of coins. Your friend Bob will pick the last pile. Repeat until there are no more piles of coins. Given an array of integers piles where piles[i] is the number of coins in the ith pile. Return the maximum number of coins that you can have.

Example 1:

Input: piles = [2,4,1,2,7,8]

Output: 9

**def maxCoins(piles):**

**sorted\_piles = sorted(piles, reverse=True)**

**your\_coins = 0**

**index = 1**

**while index < len(sorted\_piles):**

**your\_coins += sorted\_piles[index]**

**index += 2**

**return your\_coins**

**piles = [2, 4, 1, 2, 7, 8]**

**print(maxCoins(piles)) .**

1. You are given a 0-indexed integer array coins, representing the values of the coins available, and an integer target. An integer x is obtainable if there exists a subsequence of coins that sums to x. Return the minimum number of coins of any value that need to be added to the array so that every integer in the range [1, target] is obtainable. A subsequence of an array is a new non-empty array that is formed from the original array by deleting some (possibly none) of the elements without disturbing the relative positions of the remaining elements.

Example 1:

Input: coins = [1,4,10], target = 19

def minAddToMakeTarget(coins, target):

coins.sort()

max\_reachable = 0

additions = 0

for coin in coins:

while coin > max\_reachable + 1:

max\_reachable += (max\_reachable + 1)

additions += 1

if max\_reachable >= target:

return additions

max\_reachable += coin

if max\_reachable >= target:

return additions

while max\_reachable < target:

max\_reachable += (max\_reachable + 1)

additions += 1

return additions

# Test case

coins = [1, 4, 10]

target = 19

print(minAddToMakeTarget(coins, target)) # Output:

.

1. You are given an integer array jobs, where jobs[i] is the amount of time it takes to complete the ith job. There are k workers that you can assign jobs to. Each job should be assigned to exactly one worker. The working time of a worker is the sum of the time it takes to complete all jobs assigned to them. Your goal is to devise an optimal assignment such that the maximum working time of any worker is minimized. Return the minimum possible maximum working time of any assignment.

Example 1:

Input: jobs = [3,2,3], k = 3

Output: 3

Explanation: By assigning each person one job, the maximum time is 3.

def maxtime(jobs, k):

left, right = max(jobs), sum(jobs)

while left < right:

mid = (left + right) // 2

if can\_assign(jobs, k, mid):

right = mid

else:

left = mid + 1

return left

def can\_assign(jobs, k, max\_time):

workers = [0] \* k

for job in jobs:

assigned = False

for i in range(k):

if workers[i] + job <= max\_time:

workers[i] += job

assigned = True

break

if not assigned:

return False

return True

jobs = [3, 2, 3]

k = 3

print(maxtime(jobs, k)) # Output: 3

1. We have n jobs, where every job is scheduled to be done from startTime[i] to endTime[i], obtaining a profit of profit[i]. You're given the startTime, endTime and profit arrays, return the maximum profit you can take such that there are no two jobs in the subset with overlapping time range. If you choose a job that ends at time X you will be able to start another job that starts at time X.

Example 1:

Input: startTime = [1,2,3,3], endTime = [3,4,5,6], profit = [50,10,40,70]

Output: 120

def jobseq(startTime, endTime, profit):

jobs = sorted(zip(startTime, endTime, profit), key=lambda x: x[1])

max\_profit = current\_end\_time = 0

for start, end, p in jobs:

if start >= current\_end\_time:

max\_profit += p

current\_end\_time = end

return max\_profit

startTime = [1, 2, 3, 3]

endTime = [3, 4, 5, 6]

profit = [50, 10, 40, 70]

print(jobseq(startTime, endTime, profit))

1. Given a graph represented by an edge list, implement Dijkstra's Algorithm to find the shortest path from a given source vertex to a target vertex. The graph is represented as a list of edges where each edge is a tuple (u, v, w) representing an edge from vertex u to vertex v with weight w.

Test Case 1:

Input:

n = 6

edges = [(0, 1, 7), (0, 2, 9), (0, 5, 14), (1, 2, 10), (1, 3, 15),

(2, 3, 11), (2, 5, 2), (3, 4, 6), (4, 5, 9) ]

source = 0

target = 4

Output: 20

def dijkstra(edges, n, source, target):

# Step 1: Create adjacency list from edge list

graph = [[] for \_ in range(n)]

for u, v, w in edges:

graph[u].append((v, w))

graph[v].append((u, w)) # Consider undirected graph

# Step 2: Initialize distance and parent arrays

distance = [float('inf')] \* n

parent = [-1] \* n

distance[source] = 0

# Step 3: Priority queue initialization

pq = [(0, source)] # (distance, vertex)

# Step 4: Dijkstra's algorithm

while pq:

# Extract minimum distance vertex from priority queue

min\_dist = float('inf')

min\_vertex = -1

for i in range(len(pq)):

if pq[i][0] < min\_dist:

min\_dist = pq[i][0]

min\_vertex = pq[i][1]

pq.remove((min\_dist, min\_vertex))

u = min\_vertex

# Early exit if we reach the target vertex

if u == target:

break

for v, weight in graph[u]:

if distance[u] + weight < distance[v]:

distance[v] = distance[u] + weight

parent[v] = u

pq.append((distance[v], v))

# Step 5: Reconstruct the shortest path

path = []

if distance[target] != float('inf'):

# Trace back from target to source

node = target

while node != -1:

path.append(node)

node = parent[node]

path.reverse()

return distance[target], path

# Test case

n = 6

edges = [(0, 1, 7), (0, 2, 9), (0, 5, 14), (1, 2, 10), (1, 3, 15),

(2, 3, 11), (2, 5, 2), (3, 4, 6), (4, 5, 9)]

source = 0

target = 4

shortest\_distance, shortest\_path = dijkstra(edges, n, source, target)

print(f"Shortest Distance from {source} to {target}: {shortest\_distance}")

print(f"Shortest Path: {' -> '.join(map(str, shortest\_path))}")

1. Given a set of characters and their corresponding frequencies, construct the Huffman Tree and generate the Huffman Codes for each character.

Test Case 1:

Input:

n = 4

characters = ['a', 'b', 'c', 'd']

frequencies = [5, 9, 12, 13]

Output: [('a', '110'), ('b', '10'), ('c', '0'), ('d', '111')]

Test Case 2:

Input:

n = 6

characters = ['f', 'e', 'd', 'c', 'b', 'a']

frequencies = [5, 9, 12, 13, 16, 45]

Output: [ ('a', '0'), ('b', '101'), ('c', '100'), ('d', '111'), ('e', '1101'), ('f', '1100')]

1. Given a list of item weights and the maximum capacity of a container, determine the maximum weight that can be loaded into the container using a greedy approach. The greedy approach should prioritize loading heavier items first until the container reaches its capacity.

Test Case 1:

Input:

n = 5

weights = [10, 20, 30, 40, 50]

max\_capacity = 60

Output: 50

Test Case 2:

Input:

n = 6

weights = [5, 10, 15, 20, 25, 30]

max\_capacity = 50

Output: 50

1. Given a list of item weights and a maximum capacity for each container, determine the minimum number of containers required to load all items using a greedy approach. The greedy approach should prioritize loading items into the current container until it is full before moving to the next container.

Test Case 1:

Input:

n = 7

weights = [5, 10, 15, 20, 25, 30, 35]

max\_capacity = 50

Output: 4

def container\_loading(items, container\_capacity):

container = []

current\_weight = 0

for item in items:

if current\_weight + item <= container\_capacity:

container.append(item)

current\_weight += item

return container

container=[1,2,3,4,5,6]

1. Given a graph represented by an edge list, implement Kruskal's Algorithm to find the Minimum Spanning Tree (MST) and its total weight.

Test Case 1:

Input:

n = 4

m = 5

edges = [ (0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4) ]

Output:

Edges in MST: [(2, 3, 4), (0, 3, 5), (0, 1, 10)]

Total weight of MST: 19

Test Case 2:

Input:

n = 5

m = 7

edges = [ (0, 1, 2), (0, 3, 6), (1, 2, 3), (1, 3, 8), (1, 4, 5), (2, 4, 7), (3, 4, 9) ]

Output:

Edges in MST: [(0, 1, 2), (1, 2, 3), (1, 4, 5), (0, 3, 6)]

Total weight of MST: 16

1. Given a graph with weights and a potential Minimum Spanning Tree (MST), verify if the given MST is unique. If it is not unique, provide another possible MST.

Test Case 1:

Input:

n = 4

m = 5

edges = [ (0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4) ]

given\_mst = [(2, 3, 4), (0, 3, 5), (0, 1, 10)]

Output: Is the given MST unique? True

def min\_spanning\_tree(graph):

n = len(graph)

# Initialize lists to keep track of visited vertices and MST edges

visited = [False] \* n

parent = [-1] \* n # To store parent nodes in MST

min\_weights = [float('inf')] \* n # To store minimum weight edges

# Start from vertex 0

parent[0] = -1 # No parent for the starting vertex

min\_weights[0] = 0 # Starting vertex has weight 0

# Construct MST

for \_ in range(n):

# Find the vertex with the minimum weight edge that is not yet in MST

min\_weight = float('inf')

u = -1

for v in range(n):

if not visited[v] and min\_weights[v] < min\_weight:

min\_weight = min\_weights[v]

u = v

visited[u] = True

# Update min\_weights and parent for adjacent vertices of u

for v in range(n):

if graph[u][v] != 0 and not visited[v] and graph[u][v] < min\_weights[v]:

parent[v] = u

min\_weights[v] = graph[u][v]

# Prepare the MST edges

mst = []

for v in range(1, n): # Start from vertex 1 because parent[0] = -1

mst.append((parent[v], v, graph[parent[v]][v]))

return mst

# Test the function with an example graph

graph = [

[0, 2, 0, 6, 0],

[2, 0, 3, 8, 5],

[0, 3, 0, 0, 7],

[6, 8, 0, 0, 9],

[0, 5, 7, 9, 0]

]

mst = min\_spanning\_tree(graph)

total\_weight = sum(edge[2] for edge in mst)

print("Minimum Spanning Tree Edges:")

for edge in mst:

print(edge)

print("Total Weight of MST:", total\_weight)

BACKTRACKING

* 1. Discuss the importance of visualizing the solutions of the N-Queens Problem to understand the placement of queens better. Use a graphical representation to show how queens are placed on the board for different values of N. Explain how visual tools can help in debugging the algorithm and gaining insights into the problem's complexity. Provide examples of visual representations for N = 4, N = 5, and N = 8, showing different valid solutions.

1. Visualization for 4-Queens:

Input: N = 4

Output:

global N

N = 4

def queen(board):

for i in range(N):

for j in range(N):

print (board[i][j],end=' ')

print()

def s(board, row, col):

for i in range(col):

if board[row][i] == 1:

return False

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

for i, j in zip(range(row, N, 1), range(col, -1, -1)):

if board[i][j] == 1:

return False

return True

def h(board, col):

if col >= N:

return True

for i in range(N):

if s(board, i, col):

board[i][col] = 1

if h(board, col + 1) == True:

return True

board[i][col] = 0

return False

def solveNQ():

board = [ [0, 0, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0]

]

if h(board, 0) == False:

print ("Solution does not exist")

return False

queen(board)

return True

solveNQ()

* 1. Write a program to solve a Sudoku puzzle by filling the empty cells.A sudoku solution must satisfy all of the following rules:Each of the digits 1-9 must occur exactly once in each row.Each of the digits 1-9 must occur exactly once in each column.Each of the digits 1-9 must occur exactly once in each of the 9 3x3 sub-boxes of the grid.The '.' character indicates empty cells.

Example 1:

def s(board, row, col, num):

return all(board[row][i] != str(num) for i in range(9)) and \

all(board[i][col] != str(num) for i in range(9)) and \

all(board[3 \* (row // 3) + i // 3][3 \* (col // 3) + i % 3] != str(num) for i in range(9))

def sudoku(board):

for row in range(9):

for col in range(9):

if board[row][col] == '.':

for num in range(1, 10):

if s(board, row, col, num):

board[row][col] = str(num)

if sudoku(board):

return True

board[row][col] = '.'

return False

return True

def s\_board(board):

print('\n'.join([' '.join(row) for row in board]))

board = [

["5","3",".",".","7",".",".",".","."],

["6",".",".","1","9","5",".",".","."],

[".","9","8",".",".",".",".","6","."],

["8",".",".",".","6",".",".",".","3"],

["4",".",".","8",".","3",".",".","1"],

["7",".",".",".","2",".",".",".","6"],

[".","6",".",".",".",".","2","8","."],

[".",".",".","4","1","9",".",".","5"],

[".",".",".",".","8",".",".","7","9"]

]

print("Initial board:")

s\_board(board)

sudoku(board)

print("\nSolved board:")

s\_board(board)

* 1. Given an array nums of distinct integers, return all the possible permutations. You can return the answer in any order.

Example 1:

Input: nums = [1,2,3]

Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

def permute(nums):

def backtrack(start):

if start == len(nums):

result.append(nums[:])

return

for i in range(start, len(nums)):

nums[start], nums[i] = nums[i], nums[start]

backtrack(start + 1)

nums[start], nums[i] = nums[i], nums[start]

result = []

backtrack(0)

return result

input\_nums = input("Enter the numbers (separated by spaces): ").split()

nums = [int(num) for num in input\_nums]

print(permute(nums))

* 1. You and your friends are assigned the task of coloring a map with a limited number of colors. The map is represented as a list of regions and their adjacency relationships. The rules are as follows: At each step, you can choose any uncolored region and color it with any available color. Your friend Alice follows the same strategy immediately after you, and then your friend Bob follows suit. You want to maximize the number of regions you personally color. Write a function that takes the map's adjacency list representation and returns the maximum number of regions you can color before all regions are colored. Write a program to implement the Graph coloring technique for an undirected graph. Implement an algorithm with minimum number of colors. edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)] No. of vertices, n = 4

def g\_color(edges, n):

adj\_list = [[] for \_ in range(n)]

for u, v in edges:

adj\_list[u].append(v)

adj\_list[v].append(u)

colors = [-1] \* n

def s(vertex, color):

for neighbor in adj\_list[vertex]:

if colors[neighbor] == color:

return False

return True

def graph(vertex):

if vertex == n:

return 0

for color in range(n):

if s(vertex, color):

colors[vertex] = color

colour = graph(vertex + 1)

if colour != -1:

return colour

colors[vertex] = -1

return -1

colour = graph(0)

if colour == -1:

return 0

else:

return colour + 1

edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)]

n = 4

print(g\_color(edges, n))

* 1. You are given an undirected graph represented by a list of edges and the number of vertices n. Your task is to determine if there exists a Hamiltonian cycle in the graph. A Hamiltonian cycle is a cycle that visits each vertex exactly once and returns to the starting vertex. Write a function that takes the list of edges and the number of vertices as input and returns true if there exists a Hamiltonian cycle in the graph, otherwise return false. Example: Given edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2), (2, 4), (4, 0)] and n = 5

def h\_cycle(edges, n):

adj\_list = [[] for \_ in range(n)]

for u, v in edges:

adj\_list[u].append(v)

adj\_list[v].append(u)

def d(path, vertex):

return vertex not in path[:-1] and (len(path) == n and path[-1] in adj\_list[vertex] or vertex in adj\_list[path[-1]])

def h\_cycle(path):

if len(path) == n:

return True

return any(h\_cycle(path + [vertex]) for vertex in range(n) if d(path, vertex))

return any(h\_cycle([start]) for start in range(n))

edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2), (2, 4), (4, 0)]

n = 5

print(h\_cycle(edges, n))

def selectionSort(array, size):

for ind in range(size):

min\_index = ind

for j in range(ind + 1, size):

if array[j] < array[min\_index]:

min\_index = j

(array[ind], array[min\_index]) = (array[min\_index], array[ind])

arr = [2,7,1,4]

size = len(arr)

selectionSort(arr, size)

print('sorted :')

print(arr)

def bubblesort(arr):

n = len(arr)

for i in range(n):

for j in range(0, n - i - 1):

if arr[j] > arr[j + 1]:

arr[j], arr[j + 1] = arr[j + 1], arr[j]

arr = [5, 1, 4, 2, 8]

bubblesort(arr)

print("Sorted array is:", arr)

def merge\_sort(arr):

if len(arr) <= 1:

return arr

mid = len(arr) // 2

left = merge\_sort(arr[:mid])

right = merge\_sort(arr[mid:])

return merge(left, right)

def merge(left, right):

result = []

while left and right:

if left[0] <= right[0]:

result.append(left.pop(0))

else:

result.append(right.pop(0))

return result + left + right

input\_values = [5, 2, 7, 1, 9, 3]

sorted\_array = merge\_sort(input\_values)

print("Sorted array:", sorted\_array)

def quick\_sort(arr):

if len(arr) <= 1:

return arr

pivot = arr[len(arr) // 2]

left = [x for x in arr if x < pivot]

middle = [x for x in arr if x == pivot]

right = [x for x in arr if x > pivot]

return quick\_sort(left) + middle + quick\_sort(right)

input\_values = [5, 2, 7, 1, 9, 3]

print("Original array:", input\_values)

sorted\_array = quick\_sort(input\_values)

print("Sorted array:", sorted\_array)

closest pair

points = [(2, 3), (12, 30), (40, 50), (5, 1), (12, 10), (3, 4)]

points.sort()

n = len(points)

min\_distance = float('inf')

for i in range(n):

for j in range(i+1, n):

d = ((points[i][0] - points[j][0])\*\*2 + (points[i][1] - points[j][1])\*\*2)\*\*0.5

if d < min\_distance:

min\_distance = d

min\_point1 = points[i]

min\_point2 = points[j]

print("Closest pair:", min\_point1, min\_point2)

print("Distance:", min\_distance)

TODAYS TEST:

COIN CHANGE PROBLEM:

def count(S, m, n):

table = [0] \* (n + 1)

table[0] = 1

for i in range(0, m):

for j in range(S[i], n + 1):

table[j] += table[j - S[i]]

return table[n]

arr = [1, 2, 3]

m = len(arr)

n = 6

x = count(arr, m, n)

print (x)

KNAPSACK LOADING

def knapsack(weights, values, capacity):

n = len(weights)

dp = [0] \* (capacity + 1)

for i in range(n):

for w in range(capacity, weights[i] - 1, -1):

dp[w] = max(dp[w], dp[w - weights[i]] + values[i])

return dp[capacity]

weights = [2, 3, 4, 5]

values = [3, 4, 5, 6]

capacity = 5

max\_value = knapsack(weights, values, capacity)

print("Maximum value:", max\_value)

JOB SEQUENCING

def job\_sequencing(jobs):

jobs.sort(key=lambda x: x[1], reverse=True)

n = len(jobs)

result = [-1] \* n

slot = [False] \* n

for job in jobs:

for j in range(min(n, job[2]) - 1, -1, -1):

if not slot[j]:

slot[j] = True

result[j] = job[0]

break

total\_profit = sum([job[1] for i, job in enumerate(jobs) if result.count(job[0]) > 0])

job\_sequence = [result[i] for i in range(n) if result[i] != -1]