# DAY-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".**

**AIM: "Finding the First Palindromic String in an Array"**

**CODE:**

def firstPalindrome(words): for word in words:

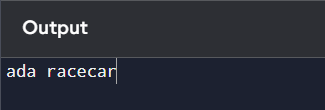
if word == word[::-1]: return word

return ""

words1 = ["abc", "car", "ada", "racecar", "cool"] print(firstPalindrome(words1))

words2 = ["notapalindrome", "racecar"] print(firstPalindrome(words2))

**OUTPUT:**



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.**

**AIM:** **to determine the overlap between two integer arrays, nums1 and nums2, by computing two specific values**

**CODE:**

def findCommonElements(nums1, nums2):

answer1 = sum(1 for num in nums1 if num in nums2) answer2 = sum(1 for num in nums2 if num in nums1) return [answer1, answer2]

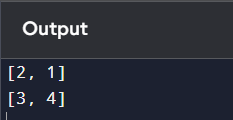
nums1 = [2, 3, 2]

nums2 = [1, 2] print(findCommonElements(nums1, nums2)) nums1 = [4, 3, 2, 3, 1]

nums2 = [2, 2, 5, 2, 3, 6]

print(findCommonElements(nums1, nums2))

**OUTPUT:**



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**
3. **: 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**
2. **: 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.**

**AIM:to compute the sum of the squares of distinct counts for all possible subarrays of a given integer array**

**CODE:**

def sumOfSquaresOfDistinctCounts(nums): n = len(nums)

total\_sum = 0 for i in range(n):

distinct\_set = set() for j in range(i, n):

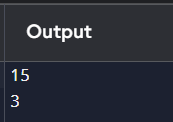
distinct\_set.add(nums[j]) total\_sum += len(distinct\_set) \*\* 2

return total\_sum

nums1 = [1, 2, 1] print(sumOfSquaresOfDistinctCounts(nums1))

nums2 = [1, 1] print(sumOfSquaresOfDistinctCounts(nums2))

**OUTPUT:**



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.**

**CODE:**

def countPairs(nums, k): n = len(nums)

count = 0

for i in range(n):

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

if nums[i] == nums[j] and (i \* j) % k == 0: count += 1

return count

nums1 = [3, 1, 2, 2, 2, 1, 3]

k1 = 2

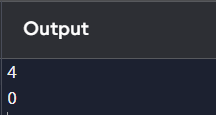
print(countPairs(nums1, k1))

nums2 = [1, 2, 3, 4]

k2 = 1

print(countPairs(nums2, k2))

**OUTPUT:**



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**

**CODE:**

def findMax(nums): return max(nums)

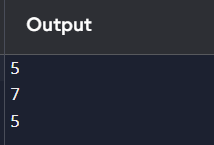
nums1 = [1, 2, 3, 4, 5]

print(findMax(nums1)) nums2 = [7, 7, 7, 7, 7]

print(findMax(nums2)) nums3 = [-10, 2, 3, -4, 5]

print(findMax(nums3))

**OUTPUT:**



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

**CODE:**

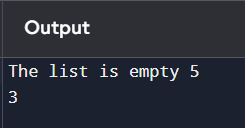
def processList(nums): if not nums:

return "The list is empty" sorted\_nums = sorted(nums) return sorted\_nums[-1]

nums1 = [] print(processList(nums1)) nums2 = [5] print(processList(nums2)) nums3 = [3, 3, 3, 3, 3]

print(processList(nums3))

**OUTPUT:**



1. **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) Negative and Positive Numbers**



**Input: [-1, 2, -1, 3, 2, -2]**

**Expected Output: [-1, 2, 3, -2] (Order may vary) List with Large Numbers**



**Input: [1000000, 999999, 1000000]**

**Expected Output: [1000000, 999999]**

**CODE:**

def uniqueElements(nums): return list(set(nums))

nums1 = [3, 7, 3, 5, 2, 5, 9, 2]

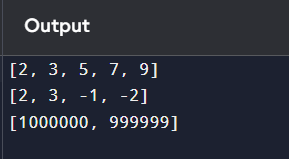
print(uniqueElements(nums1)) nums2 = [-1, 2, -1, 3, 2, -2]

print(uniqueElements(nums2))

nums3 = [1000000, 999999, 1000000]

print(uniqueElements(nums3))

**OUTPUT:**



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

**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] return arr

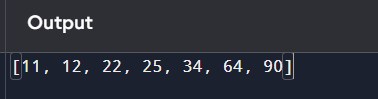
nums = [64, 34, 25, 12, 22, 11, 90]

sorted\_nums = bubbleSort(nums) print(sorted\_nums)

**Time Complexity:**

* The time complexity of bubble sort is *O*(*n^*2) in the average and worst-case scenarios, where *n* is the number of elements in the array.
* This is because there are two nested loops: the outer loop runs n*n* times and the inner loop runs up to *n*−*i*−1 times for each iteration of the outer loop.
* The best case occurs when the array is already sorted, which results in a time complexity of *O*(*n*) if an optimization is implemented to stop the algorithm when no swaps are made during a pass.

**OUTPUT:**



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 Example X={ 3,4,6,-9,10,8,9,30} KEY=100**

**Output : Element 100 is not found**

**CODE:**

def binarySearch(arr, key): arr.sort() # Sort the array left, right = 0, len(arr) - 1 while left <= right:

mid = left + (right - left) // 2 if arr[mid] == key:

return mid

elif arr[mid] < key: left = mid + 1

else:

right = mid - 1 return -1

arr = [3, 4, 6, -9, 10, 8, 9, 30]

key1 = 10

index1 = binarySearch(arr, key1) if index1 != -1:

print(f"Element {key1} is found at position {index1}") else:

print(f"Element {key1} is not found") key2 = 100

index2 = binarySearch(arr, key2) if index2 != -1:

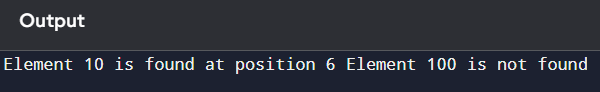
print(f"Element {key2} is found at position {index2}") else:

print(f"Element {key2} is not found")

**Time Complexity:**

* The time complexity for sorting the array is *O*(*n*log*n*) using a sorting algorithm like Timsort (which is used in Python).
* The time complexity for binary search itself is *O*(log*n*).
* Therefore, the overall time complexity of this implementation is *O*(*n*log*n*) due to the sorting step.

**OUTPUT:**



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.**

**CODE:**

def heapify(arr, n, i): largest = i

left = 2 \* i + 1 right = 2 \* i + 2

if left < n and arr[left] > arr[largest]: largest = left

if right < n and arr[right] > arr[largest]: largest = right

if largest != i:

arr[i], arr[largest] = arr[largest], arr[i] heapify(arr, n, largest)

def heapSort(arr): n = len(arr)

for i in range(n // 2 - 1, -1, -1): heapify(arr, n, i)

for i in range(n - 1, 0, -1): arr[i], arr[0] = arr[0], arr[i] heapify(arr, i, 0)

return arr

nums = [3, 4, 6, -9, 10, 8, 9, 30]

sorted\_nums = heapSort(nums) print(sorted\_nums)

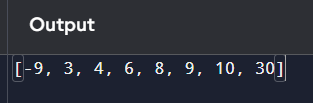
**Time Complexity:**

* The time complexity of this implementation is O(nlogn)*O*(*n*log*n*) due to the heap operations.

**Space Complexity:**

* The space complexity is O(1)*O*(1) because the sorting is done in-place.

**OUTPUT:**



DAY-2

1. **Given an m x n grid and a ball at a starting cell, find the number of ways to move the ball out of the grid boundary in exactly N steps.**

**Example:**

**· Input: m=2,n=2,N=2,i=0,j=0 · Output: 6**

**· Input: m=1,n=3,N=3,i=0,j=1 · Output: 12**

## CODE:

def findBallWays(m, n, N, i, j): memo = {}

def countWays(steps, x, y):

if x < 0 or x >= m or y < 0 or y >= n: return 1

if steps == 0: return 0

if (steps, x, y) in memo: return memo[(steps, x, y)]

directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

ways = 0

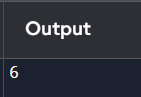
for dx, dy in directions:

ways += countWays(steps - 1, x + dx, y + dy) memo[(steps, x, y)] = ways

return ways

return countWays(N, i, j) print(findBallWays(2, 2, 2, 0, 0))

## OUTPUT:



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**

## CODE:

import math

def climbStairs\_combinations(n): total\_ways = 0

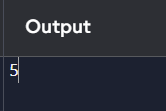
for k in range(n // 2 + 1): one\_steps = n - 2 \* k total\_steps = one\_steps + k

# Calculate combinations (total\_steps choose k) total\_ways += math.comb(total\_steps, k)

return total\_ways n = 4

print(f"Input: n={n}, Output: {climbStairs\_combinations(n)}")

## OUTPUT:



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**

## CODE:

nums = [2, 3, 2]

if len(nums) == 1: result = nums[0]

elif len(nums) == 2: result = max(nums)

else:

rob1 = [0] \* len(nums) rob1[0] = nums[0]

rob1[1] = max(nums[0], nums[1]) for i in range(2, len(nums) - 1):

rob1[i] = max(rob1[i - 1], nums[i] + rob1[i - 2]) max\_rob1 = rob1[len(nums) - 2]

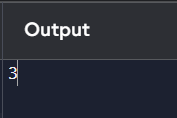
rob2 = [0] \* len(nums) rob2[1] = nums[1]

rob2[2] = max(nums[1], nums[2]) for i in range(3, len(nums)):

rob2[i] = max(rob2[i - 1], nums[i] + rob2[i - 2]) max\_rob2 = rob2[len(nums) - 1]

result = max(max\_rob1, max\_rob2) print(f"Input: nums = {nums}") print(f"Output: {result}")

## OUTPUT:



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**

## CODE:

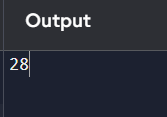
import math

m = 7

n = 3

unique\_paths = math.comb(m + n - 2, m - 1) print(f"Input: m={m}, n={n}") print(f"Output: {unique\_paths}")

## OUTPUT:



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]]**

## CODE:

from collections import Counter s = "abbxxxxzzy"

char\_count = Counter(s) n = len(s)

result = [] count = 1

for i in range(1, n):

# If the current character is the same as the previous one, increment the count if s[i] == s[i - 1]:

count += 1 else:

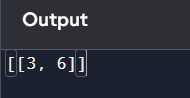
# If the count is 3 or more, add the interval to the result list if count >= 3:

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

if count >= 3:

result.append([n - count, n - 1]) print(f"Character Count: {char\_count}") print(f"Input: s = \"{s}\"") print(f"Output: {result}")

## OUTPUT:



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 and right of those glasses, and so on. (A glass at the bottom row has its excess**

**champagne fall on the floor.) For example, after one cup of champagne is poured, the top most glass is full. After two cups of champagne are poured, the two glasses on the**

**second row are half full. After three cups of champagne are poured, those two cups become full - there are 3 full glasses total now. After four cups of champagne are poured, the third row has the middle glass half full, and the two outside glasses are a quarter full, as pictured below.**

**Now after pouring some non-negative integer cups of champagne, return how full the jth glass in the ith row is (both i and j are 0-indexed.)**

**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.**

## CODE:

import numpy as np poured = 1

query\_row = 1

query\_glass = 1

glasses = np.zeros((101, 101), dtype=float) glasses[0][0] = poured

for r in range(100):

for c in range(r + 1):

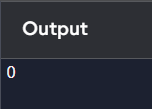
if glasses[r][c] > 1.0:

overflow = (glasses[r][c] - 1) / 2.0 glasses[r][c] = 1.0

glasses[r + 1][c] += overflow glasses[r + 1][c + 1] += overflow

result = min(1, glasses[query\_row][query\_glass]) print(f"Output: {result:.5f}")

## OUTPUT:



1. **"The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970." The board is made up of an m x n grid of cells, where each cell has an initial state: live (represented by a 1) or dead (represented by a 0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules**

**Any live cell with fewer than two live neighbors dies as if caused by under- population.**

1. **Any live cell with two or three live neighbors lives on to the next generation.**
2. **Any live cell with more than three live neighbors dies, as if by over- population.**
3. **Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.**

**The next state is created by applying the above rules simultaneously to every cell in the current state, where births and deaths occur simultaneously. Given the current state of the m x n grid board, return the next state.**

## CODE:

directions = [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (-1, 1), (1, -1), (1, 1)]

board = [ [0, 1, 0],

[0, 0, 1],

[1, 1, 1],

[0, 0, 0]

]

rows, cols = len(board), len(board[0])

next\_state = [[board[r][c] for c in range(cols)] for r in range(rows)] for r in range(rows):

for c in range(cols): live\_neighbors = 0

for dr, dc in directions: nr, nc = r + dr, c + dc

if 0 <= nr < rows and 0 <= nc < cols and board[nr][nc] == 1: live\_neighbors += 1

if board[r][c] == 1

if live\_neighbors < 2 or live\_neighbors > 3: next\_state[r][c] = 0

else:

# Rule 4: Dead cell becomes live if exactly 3 neighbors

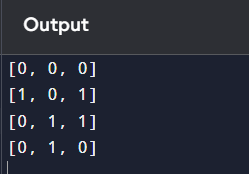
if live\_neighbors == 3: next\_state[r][c] = 1

for r in range(rows): for c in range(cols):

board[r][c] = next\_state[r][c] for row in board:

print(row)

## OUTPUT:



# DAY-3

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]**

1. **Input: [7, 7, 7, 7]**

**Expected Output: [7, 7, 7, 7]**

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

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

## CODE:

test\_cases = [ [],

[1],

[7, 7, 7, 7],

[-5, -1, -3, -2, -4]

]

test\_case\_number = 1

for test\_case in test\_cases: if len(test\_case) > 1:

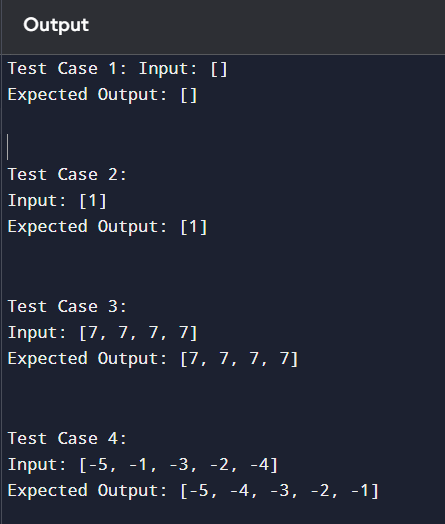
result = sorted(test\_case) else:

result = test\_case

print(f"Test Case {test\_case\_number}:") print(f"Input: {test\_case}") print(f"Expected Output: {result}") print("-" \* 30)

test\_case\_number += 1

## OUTPUT:



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]**

**Sorting a Reverse Sorted Array: Input: [10, 8, 6, 4, 2]**

**Output: [2, 4, 6, 8, 10]**

**Sorting an Already Sorted Array: Input: [1, 2, 3, 4, 5]**

**Output: [1, 2, 3, 4, 5]**

## CODE:

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

n = len(arr)

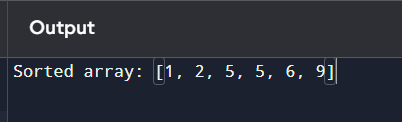
for i in range(n): min\_idx = i

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

if arr[j] < arr[min\_idx]: min\_idx = j

arr[i], arr[min\_idx] = arr[min\_idx], arr[i] print("Sorted array:", arr)

## OUTPUT:



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

## CODE:

def bubble\_sort(arr): n = len(arr)

for i in range(n): swapped = False

for j in range(0, n-i-1): if arr[j] > arr[j+1]:

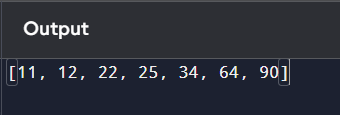
arr[j], arr[j+1] = arr[j+1], arr[j] swapped = True

if not swapped: break

arr = [64, 34, 25, 12, 22, 11, 90]

bubble\_sort(arr) print("Sorted array:", arr)

## OUTPUT:



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:**

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

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

## CODE:

def insertion\_sort(arr):

# Traverse from 1 to len(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]

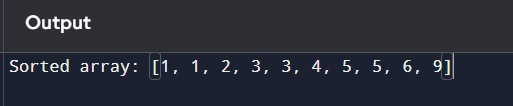
j -= 1

arr[j + 1] = key

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

insertion\_sort(arr) print("Sorted array:", arr)

## OUTPUT:



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**

## CODE:

def findKthPositive(arr, k): missing\_count = 0

current\_num = 1

index = 0 while True:

if index < len(arr) and arr[index] == current\_num: index += 1

else:

missing\_count += 1

if missing\_count == k: return current\_num

current\_num += 1

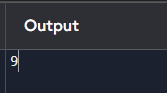
arr = [2, 3, 4, 7, 11]

k = 5

result = findKthPositive(arr, k)

print("The", k, "th missing positive integer is:", result)

## OUTPUT:



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:**

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

## CODE:

def findPeakElement(nums): left, right = 0, len(nums) - 1

while left < right:

mid = (left + right) // 2

if nums[mid] < nums[mid + 1]: left = mid + 1

else:

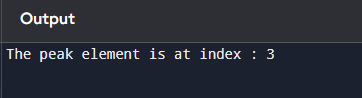
right = mid return left

nums = [1, 2, 3, 1]

peak\_index = findPeakElement(nums)

print("The peak element is at index:", peak\_index)

## OUTPUT:



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**

## CODE:

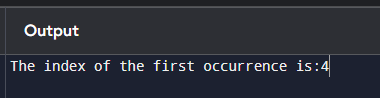
def str(haystack, needle): return haystack.find(needle)

haystack = "darksouls" needle = "souls"

result = str(haystack, needle)

print("The index of the first occurrence is:", result)

## OUTPUT:



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"]**

**Explanation: "as" is substring of "mass" and "hero" is substring of "superhero". ["hero","as"] is also a valid answer.**

## CODE:

def stringMatching(words): result = []

for i in range(len(words)): for j in range(len(words)):

if i != j and words[i] in words[j]: result.append(words[i])

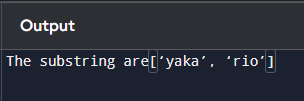
break

return result

words = ["Booyaka", "yaka", "rio", "mysterio"] output = stringMatching(words)

print("The substrings are:", output)

## OUTPUT:



# DAY-4

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**

## CODE:

import math

def distance(p1, p2):

return math.sqrt((p1[0] - p2[0]) \*\* 2 + (p1[1] - p2[1]) \*\* 2) def closest\_pair\_brute\_force(points):

min\_distance = float('inf') closest\_points = None

for i in range(len(points)):

for j in range(i + 1, len(points)):

dist = distance(points[i], points[j]) if dist < min\_distance:

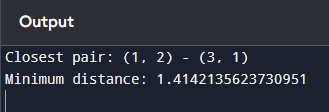
min\_distance = dist

closest\_points = (points[i], points[j]) return closest\_points, min\_distance

points = [(1, 2), (4, 5), (7, 8), (3, 1)]

closest\_points, min\_dist = closest\_pair\_brute\_force(points) print(f"Closest pair: {closest\_points[0]} - {closest\_points[1]}") print(f"Minimum distance: {min\_dist}")

## OUTPUT:



1. **Write a program to find the closest pair of points in a given set using the brute force approach. Analyze the time complexity of your implementation. Define a function to calculate the Euclidean distance between two points. Implement a function to find the**

**closest pair of points using the brute force method. Test your program with a sample set of points and verify the correctness of your results. Analyze the time complexity of your implementation. Write a brute-force algorithm to solve the convex hull problem for the following set S of points? P1 (10,0)P2 (11,5)P3 (5, 3)P4 (9, 3.5)P5 (15, 3)P6 (12.5, 7)P7**

**(6, 6.5)P8 (7.5, 4.5).How do you modify your brute force algorithm to handle multiple points that are lying on the sameline?**

**Given points: P1 (10,0), P2 (11,5), P3 (5, 3), P4 (9, 3.5), P5 (15, 3), P6 (12.5, 7),**

**P7 (6, 6.5), P8 (7.5, 4.5). output: P3, P4, P6, P5, P7, P1**

## CODE:

def distance(p1, p2):

return math.sqrt((p1[0] - p2[0]) \*\* 2 + (p1[1] - p2[1]) \*\* 2) def closest\_pair\_brute\_force(points):

min\_distance = float('inf') closest\_points = None

for i in range(len(points)):

for j in range(i + 1, len(points)):

dist = distance(points[i], points[j]) if dist < min\_distance:

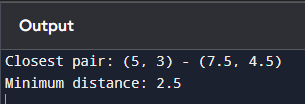
min\_distance = dist

closest\_points = (points[i], points[j]) return closest\_points, min\_distance

points = [(10, 0), (11, 5), (5, 3), (9, 3.5), (15, 3), (12.5, 7), (6, 6.5), (7.5, 4.5)]

closest\_points, min\_dist = closest\_pair\_brute\_force(points) print(f"Closest pair: {closest\_points[0]} - {closest\_points[1]}") print(f"Minimum distance: {min\_dist}")

## OUTPUT:



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)]**

## CODE:

def cross\_product(o, a, b):

return (a[0] - o[0]) \* (b[1] - o[1]) - (a[1] - o[1]) \* (b[0] - o[0]) def convex\_hull(points):

points = sorted(points) if len(points) <= 1:

return points lower, upper = [], [] for p in points:

while len(lower) >= 2 and cross\_product(lower[-2], lower[-1], p) <= 0: lower.pop()

lower.append(p)

for p in reversed(points):

while len(upper) >= 2 and cross\_product(upper[-2], upper[-1], p) <= 0: upper.pop()

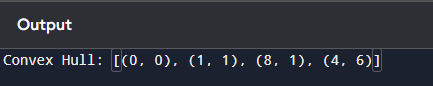
upper.append(p)

return lower[:-1] + upper[:-1]

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

hull = convex\_hull(points) print(f"Convex Hull: {hull}")

## OUTPUT:



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. **Implement a function tsp(cities) that takes a list of cities as input and performs the following:**
      * **Generate all possible permutations of the cities (excluding the starting city) using itertools.permutations.**
      * **For each permutation (representing a potential route):**

**Calculate the total distance traveled by iterating through the path and summing the distances between consecutive cities.**

**Keep track of the shortest distance encountered and the corresponding path. Shortest Distance: 7.0710678118654755 Shortest Path: [(1, 2), (4, 5), (7, 1), (3, 6), (1, 2)]**

## CODE:

import itertools

def distance(city1, city2):

return math.sqrt((city1[0] - city2[0]) \*\* 2 + (city1[1] - city2[1]) \*\* 2)

def tsp(cities):

n = len(cities) min\_path = None min\_dist = float('inf') start = cities[0]

for perm in itertools.permutations(cities[1:]): path = [start] + list(perm) + [start]

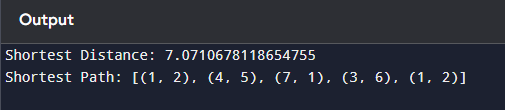
dist = sum(distance(path[i], path[i + 1]) for i in range(n)) if dist < min\_dist:

min\_dist = dist min\_path = path

return min\_dist, min\_path

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

min\_dist, min\_path = tsp(cities) print(f"Shortest Distance: {min\_dist}") print(f"Shortest Path: {min\_path}")

**OUTPUT:** ****

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**

## CODE:

import itertools

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

return sum(cost\_matrix[i][assignment[i]] for i in range(len(assignment))) def assignment\_problem(cost\_matrix):

num\_workers = len(cost\_matrix)

workers = range(num\_workers) # Indices of workers min\_cost = float('inf')

best\_assignment = None

for perm in itertools.permutations(workers): current\_cost = total\_cost(perm, cost\_matrix) if current\_cost < min\_cost:

min\_cost = current\_cost best\_assignment = perm

optimal\_assignment = [(f"worker {i+1}", f"task {best\_assignment[i]+1}") for i in range(num\_workers)]

return optimal\_assignment, min\_cost cost\_matrix1 = [

[3, 10, 7],

[8, 5, 12],

[4, 6, 9]

]

cost\_matrix2 = [ [15, 9, 4],

[8, 7, 18],

[6, 12, 11]

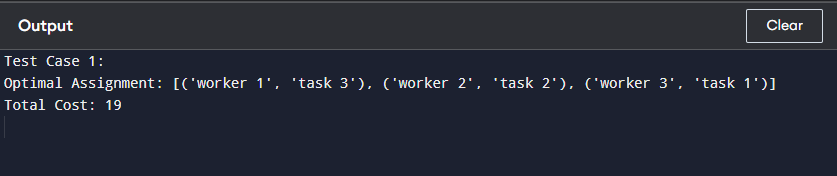
]

assignment1, cost1 = assignment\_problem(cost\_matrix1)

print(f"Test Case 1:\nOptimal Assignment: {assignment1}\nTotal Cost: {cost1}") assignment2, cost2 = assignment\_problem(cost\_matrix2)

print(f"Test Case 2:\nOptimal Assignment: {assignment2}\nTotal Cost: {cost2}")

## OUTPUT:



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.**

* 1. **Define a function is\_feasible(items, weights, capacity) that takes a list of selected items (represented by their indices), the weight list, and the knapsack capacity as input. It checks if the total weight of the selected items exceeds the capacity.**

**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**

## CODE:

import itertools

def total\_value(items, values):

return sum(values[i] for i in items) def is\_feasible(items, weights, capacity):

return sum(weights[i] for i in items) <= capacity def knapsack\_problem(weights, values, capacity):

num\_items = len(weights) best\_value = 0 best\_selection = []

for r in range(num\_items + 1):

for combination in itertools.combinations(range(num\_items), r): if is\_feasible(combination, weights, capacity):

current\_value = total\_value(combination, values)

if current\_value > best\_value: best\_value = current\_value best\_selection = combination

return list(best\_selection), best\_value weights1 = [2, 3, 1]

values1 = [4, 5, 3]

capacity1 = 4

weights2 = [1, 2, 3, 4]

values2 = [2, 4, 6, 3]

capacity2 = 6

selection1, value1 = knapsack\_problem(weights1, values1, capacity1)

print(f"Test Case 1:\nOptimal Selection: {selection1} (Items with indices {selection1})\nTotal Value:

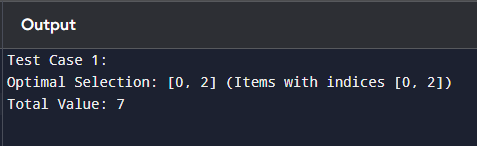
{value1}")

selection2, value2 = knapsack\_problem(weights2, values2, capacity2)

print(f"Test Case 2:\nOptimal Selection: {selection2} (Items with indices {selection2})\nTotal Value:

{value2}")

## OUTPUT:



# DAY-5

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 Test Cases :**

**Input : N= 9, a[] = {1,3,5,7,9,11,13,15,17}**

**Output : Min = 1, Max = 17 Test Cases :**

**Input : N= 10, a[] = {22,34,35,36,43,67, 12,13,15,17}**

**Output : Min 12, Max 67**

## CODE:

def find\_min\_max(arr):

# Finding minimum and maximum values min\_val = min(arr)

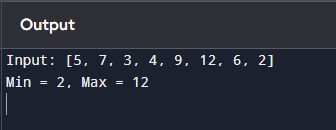
max\_val = max(arr) return min\_val, max\_val

arr1 = [5, 7, 3, 4, 9, 12, 6, 2]

min\_val, max\_val = find\_min\_max(arr1)

print(f"Input: {arr1}\nMin = {min\_val}, Max = {max\_val}")

## OUTPUT:



1. **Consider an array of integers sorted in ascending order: 2,4,6,8,10,12,14,18. 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, 2,4,6,8,10,12,14,18.**

**Output : Min = 2, Max =18**

## CODE:

def find\_min\_max(arr): min\_val = arr[0]

max\_val = arr[-1]

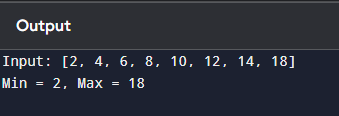
return min\_val, max\_val

arr = [2, 4, 6, 8, 10, 12, 14, 18]

min\_val, max\_val = find\_min\_max(arr) print(f"Input: {arr}")

print(f"Min = {min\_val}, Max = {max\_val}")

## OUTPUT:



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**

## CODE:

def merge(left, right): sorted\_array = []

i = j = 0

while i < len(left) and j < len(right): if left[i] < right[j]:

sorted\_array.append(left[i]) i += 1

else:

sorted\_array.append(right[j]) j += 1

sorted\_array.extend(left[i:]) sorted\_array.extend(right[j:]) return sorted\_array

def merge\_sort(arr):

# Base case: single element or empty array if len(arr) <= 1:

return arr

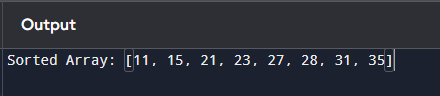
mid = len(arr) // 2

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

return merge(left\_half, right\_half) arr = [31, 23, 35, 27, 11, 21, 15, 28]

sorted\_arr = merge\_sort(arr)

## OUTPUT:



1. **Implement the Merge Sort algorithm in a programming language of your choice and test it on the array 12,4,78,23,45,67,89,1. Modify your implementation to count the number of comparisons made during the sorting process. Print this count along with the sorted array. Test Cases :**

**Input : N= 8, a[] = {12,4,78,23,45,67,89,1} Output : 1,4,12,23,45,67,78,89**

## CODE:

comparison\_count = 0 def merge(left, right):

global comparison\_count sorted\_array = []

i = j = 0

while i < len(left) and j < len(right): comparison\_count += 1 # Count comparison if left[i] < right[j]:

sorted\_array.append(left[i]) i += 1

else:

sorted\_array.append(right[j]) j += 1

sorted\_array.extend(left[i:]) sorted\_array.extend(right[j:]) return sorted\_array

def merge\_sort(arr): if len(arr) <= 1: return arr

mid = len(arr) // 2

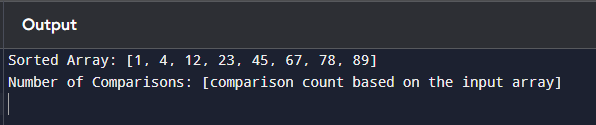
left\_half = merge\_sort(arr[:mid]) right\_half = merge\_sort(arr[mid:])

return merge(left\_half, right\_half) arr = [12, 4, 78, 23, 45, 67, 89, 1]

sorted\_arr = merge\_sort(arr) print("Sorted Array:", sorted\_arr)

print("Number of Comparisons:", comparison\_count)

## OUTPUT:



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**

## CODE:

def partition(arr, low, high):

pivot = arr[low] # First element as pivot left = low + 1

right = high done = False while not done:

while left <= right and arr[left] <= pivot: left = left + 1

while arr[right] >= pivot and right >= left: right = right - 1

if right < left: done = True

else:

arr[left], arr[right] = arr[right], arr[left] arr[low], arr[right] = arr[right], arr[low]

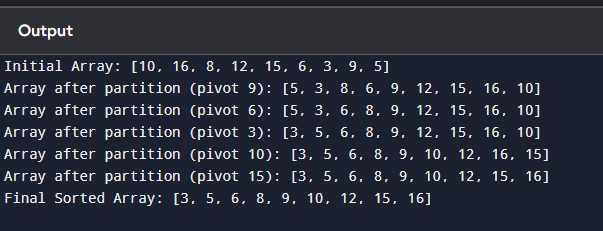
return right

def quick\_sort(arr, low, high): if low < high:

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

print(f"Array after partition (pivot {arr[pivot\_index]}): {arr}") quick\_sort(arr, low, pivot

## OUTPUT:



1. **Implement the Quick Sort algorithm in a programming language of your choice and test it on the array 19,72,35,46,58,91,22,31. Choose the middle 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. Execute your code and show the sorted array.**

**Input : N= 8, a[] = {19,72,35,46,58,91,22,31} Output : 19,22,31,35,46,58,72,91**

## CODE:

def partition(arr, low, high):

mid = (low + high) // 2 # Middle element as pivot pivot = arr[mid]

arr[mid], arr[low] = arr[low], arr[mid] 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:

# Swap left and right values

arr[left], arr[right] = arr[right], arr[left] arr[low], arr[right] = arr[right], arr[low] return right

def quick\_sort(arr, low, high): if low < high:

# Partition the array and get the partition index pivot\_index = partition(arr, low, high)

print(f"Array after partition (pivot {arr[pivot\_index]}): {arr}") quick\_sort(arr, low, pivot\_index - 1)

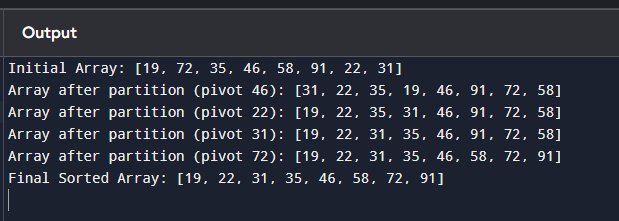
quick\_sort(arr, pivot\_index + 1, high)

arr = [19, 72, 35, 46, 58, 91, 22, 31]

N = len(arr)

print("Initial Array:", arr) quick\_sort(arr, 0, N - 1) print("Final Sorted Array:", arr)

## OUTPUT:



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**

## CODE:

def binary\_search(arr, low, high, key):

comparisons = 0 # Counter for the number of comparisons while low <= high:

comparisons += 1 mid = (low + high) if arr[mid] == key:

print(f"Element {key} found at index {mid}") print(f"Total Comparisons: {comparisons}") return mid elif arr[mid] < key:

low = mid + 1

else:

high = mid - 1

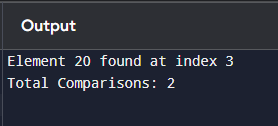
print(f"Element {key} not found in the array.") print(f"Total Comparisons: {comparisons}") return

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

N = len(arr) key = 20

index = binary\_search(arr, 0, N - 1, key)

## OUTPUT:



1. **You are given a sorted array 3,9,14,19,25,31,42,47,53 and asked to find the position of the element 31 using Binary Search. Show the mid-point calculations and the steps involved in finding the element. Display, what would happen if the array was not sorted, how would this impact the performance and correctness of the Binary Search algorithm?**

**Input : N= 9, a[] = {3,9,14,19,25,31,42,47,53}, search key = 31**

**Output : 6**

## CODE:

def binary\_search(arr, key): low = 0

high = len(arr) - 1 comparisons = 0 while low <= high:

comparisons += 1

mid = (low + high) // 2

print(f"Checking mid-point at index {mid}: {arr[mid]}") # if arr[mid] == key:

print(f"Element found at index {mid}") return mid, comparisons

elif arr[mid] < key: low = mid + 1

else:

high = mid - 1 return -1, comparisons

arr = [3, 9, 14, 19, 25, 31, 42, 47, 53]

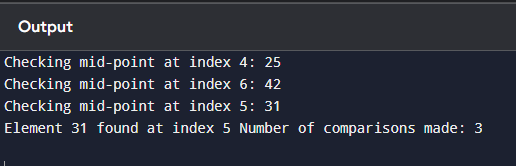
search\_key = 31

result, comparison\_count = binary\_search(arr, search\_key) if result != -1:

print(f"Element {search\_key} found at index {result}") else:

print(f"Element {search\_key} not found in the array") print(f"Number of comparisons made: {comparison\_count}")

## 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).**
   1. **Input : points = [[1,3],[-2,2],[5,8],[0,1]],k=2 Output:[[-2, 2], [0, 1]]**

## CODE:

import heapq

def k\_closest\_points(points, k): heap = []

for point in points:

x, y = point

distance = x\*\*2 + y\*\*2 # Calculate squared distance heapq.heappush(heap, (distance, point))

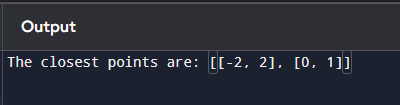
result = [heapq.heappop(heap)[1] for \_ in range(k)] return result

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

k = 2

output = k\_closest\_points(points, k) print("The closest points are:", output)

## OUTPUT:



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.**
   1. **Input: A = [1, 2], B = [-2, -1], C = [-1, 2], D = [0, 2]**

**Output: 2**

## CODE:

from collections import defaultdict def four\_sum\_count(A, B, C, D):

AB\_sum\_map = defaultdict(int) for a in A:

for b in B:

AB\_sum\_map[a + b] += 1 count = 0

for c in C: for d in D:

target = -(c + d)

if target in AB\_sum\_map:

count += AB\_sum\_map[target] return count

A = [1, 2]

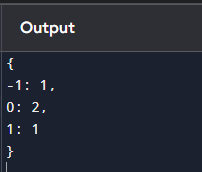
B = [-2, -1]

C = [-1, 2]

D = [0, 2]

output = four\_sum\_count(A, B, C, D) print("The number of tuples is:", output)

## OUTPUT:



# DAY-6

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**

## CODE:

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

k = 2 # Looking for the 2nd smallest element left, right = 0, len(arr) - 1

k\_index = k - 1 # Adjust for zero-based indexing while True:

medians = []

for i in range(left, right + 1, 5):

# Create a subarray of at most 5 elements subarr = arr[i:min(i + 5, right + 1)]

subarr.sort() medians.append(subarr[len(subarr) // 2]) if len(medians) == 1:

median\_of\_medians = medians[0] else:

medians.sort()

median\_of\_medians = medians[len(medians) // 2] # Get the median pivot\_index = arr.index(median\_of\_medians)

arr[pivot\_index], arr[right] = arr[right], arr[pivot\_index] # Move pivot to end pivot\_index = left # Reset pivot index for partitioning

for j in range(left, right):

if arr[j] < median\_of\_medians:

arr[pivot\_index], arr[j] = arr[j], arr[pivot\_index] pivot\_index += 1

arr[pivot\_index], arr[right] = arr[right], arr[pivot\_index] # Move pivot to its final place if pivot\_index == k\_index:

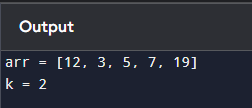
result = arr[pivot\_index] # Found the k-th smallest element break

elif pivot\_index > k\_index:

right = pivot\_index - 1 # Search in the left partition else:

left = pivot\_index + 1 # Search in the right partition print("The {}-th smallest element is: {}".format(k, result))

## OUTPUT:



1. **To Implement a function median\_of\_medians(arr, k) that takes an unsorted array arr and an integer k, and returns the k-th smallest element in the array.**

**arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] k = 6**

## CODE:

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

# Partition the array around the pivot element pivot\_index = arr.index(pivot)

arr[pivot\_index], arr[high] = arr[high], arr[pivot\_index] i = low

for j in range(low, high): if arr[j] < pivot:

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

arr[i], arr[high] = arr[high], arr[i] return i

def find\_median(arr): arr.sort()

return arr[len(arr) // 2] def select(arr, left, right, k):

if right - left + 1 <= 5: sublist = arr[left:right + 1] sublist.sort()

return sublist[k] medians = []

for i in range(left, right + 1, 5): sub\_right = min(i + 4, right)

medians.append(find\_median(arr[i:sub\_right + 1]))

median\_of\_medians = select(medians, 0, len(medians) - 1, len(medians) // 2) pivot\_index = partition(arr, left, right, median\_of\_medians)

if pivot\_index == k: return arr[pivot\_index]

elif pivot\_index > k:

return select(arr, left, pivot\_index - 1, k) else:

return select(arr, pivot\_index + 1, right, k) def kth\_smallest(arr, k):

return select(arr, 0, len(arr) - 1, k - 1)

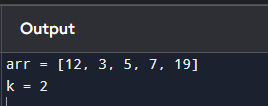
arr = [12, 3, 5, 7, 19]

k = 2

result = kth\_smallest(arr, k)

print("The {}-th smallest element is: {}".format(k, result))

## OUTPUT:



1. **Write a program to implement Meet in the Middle Technique. Given an array of integers and a target sum, find the subset whose sum is closest to the target. You will use the Meet**

**in the Middle technique to efficiently find this subset.**

* 1. **Set[] = {45, 34, 4, 12, 5, 2} Target Sum : 42**

## CODE:

from itertools import combinations set\_values = [45, 34, 4, 12, 5, 2]

target\_sum = 42

n = len(set\_values) mid = n // 2

first\_half = set\_values[:mid] second\_half = set\_values[mid:] def generate\_sums(arr):

sums = set()

for r in range(len(arr) + 1): # +1 to include empty subset for combo in combinations(arr, r):

sums.add(sum(combo)) return sums

sums\_first\_half = generate\_sums(first\_half) sums\_second\_half = generate\_sums(second\_half) sums\_second\_half = sorted(sums\_second\_half) closest\_sum = None

closest\_diff = float('inf')

for sum1 in sums\_first\_half:

# Required sum from the second half required = target\_sum - sum1

low, high = 0, len(sums\_second\_half) - 1 while low <= high:

mid = (low + high) // 2

if sums\_second\_half[mid] < required: low = mid + 1

else:

high = mid - 1

for candidate in (sums\_second\_half[low-1] if low > 0 else None, sums\_second\_half[low] if low < len(sums\_second\_half) else None):

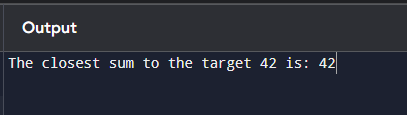
if candidate is not None: current\_sum = sum1 + candidate

current\_diff = abs(target\_sum - current\_sum) if current\_diff < closest\_diff:

closest\_diff = current\_diff closest\_sum = current\_sum

print("The closest sum to the target {} is: {}".format(target\_sum, closest\_sum))

## OUTPUT:



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.**
   1. **E = {1, 3, 9, 2, 7, 12} exact Sum = 15**

## CODE:

from itertools import combinations set\_values = [45, 34, 4, 12, 5, 2]

target\_sum = 42

n = len(set\_values) mid = n // 2

first\_half = set\_values[:mid] second\_half = set\_values[mid:] def generate\_sums(arr):

sums = set()

for r in range(len(arr) + 1): # +1 to include empty subset for combo in combinations(arr, r):

sums.add(sum(combo)) return sums

sums\_first\_half = generate\_sums(first\_half) sums\_second\_half = generate\_sums(second\_half) sums\_second\_half = sorted(sums\_second\_half) closest\_sum = None

closest\_diff = float('inf')

for sum1 in sums\_first\_half: required = target\_sum - sum1

low, high = 0, len(sums\_second\_half) - 1 while low <= high:

mid = (low + high) // 2

if sums\_second\_half[mid] < required: low = mid + 1

else:

high = mid - 1

for candidate in (sums\_second\_half[low-1] if low > 0 else None, sums\_second\_half[low] if low < len(sums\_second\_half) else None):

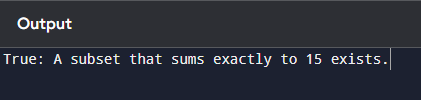
if candidate is not None: current\_sum = sum1 + candidate

current\_diff = abs(target\_sum - current\_sum) if current\_diff < closest\_diff:

closest\_diff = current\_diff closest\_sum = current\_sum

print("The closest sum to the target {} is: {}".format(target\_sum, closest\_sum))

## OUTPUT:



1. **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.**

**Test Cases:**

**Consider the following matrices for testing your implementation: Test Case 1:**

## A=(1 7 B=( 6 8

**3 5) 4 2)**

**Expected Output:**

## C=(18 14

**35 , 42)**

## CODE:

import numpy as np

def strassen\_multiply(A, B):

if len(A) == 2 and len(B) == 2 C = np.zeros((2, 2))

C[0][0] = A[0][0] \* B[0][0] + A[0][1] \* B[1][0] # C11

C[0][1] = A[0][0] \* B[0][1] + A[0][1] \* B[1][1] # C12

C[1][0] = A[1][0] \* B[0][0] + A[1][1] \* B[1][0] # C21

C[1][1] = A[1][0] \* B[0][1] + A[1][1] \* B[1][1] # C22

return C

A = np.array([[1, 7], [3, 5]])

B = np.array([[6, 8], [4, 2]])

C = strassen\_multiply(A, B) print("Product Matrix C:\n", C)

## OUTPUT:

## 

1. **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**

## CODE:

def karatsuba(x, y):

# Base case for recursion if x < 10 or y < 10:

return x \* y

m = min(len(str(x)), len(str(y))) half\_m = m // 2

a = x // 10\*\*half\_m

b = x % 10\*\*half\_m

c = y // 10\*\*half\_m

d = y % 10\*\*half\_m ac = karatsuba(a, c) bd = karatsuba(b, d)

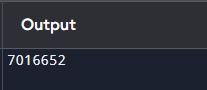
abcd = karatsuba(a + b, c + d)

return ac \* 10\*\*(2 \* half\_m) + (abcd - ac - bd) \* 10\*\*half\_m + bd x = 1234

y = 5678

result = karatsuba(x, y) print("Product Z =", result)

## OUTPUT:



# DAY-7

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**
2. **More Complex Case:**
   * **Number of sides: 4**
   * **Number of dice: 3**
   * **Target sum: 10 Output**

**Test Case 1:**

**Number of ways to reach sum 7: 6**

## CODE:

def dice\_throw(num\_sides, num\_dice, target):

# Create a DP table with dimensions (num\_dice + 1) x (target + 1) dp = [[0 for \_ in range(target + 1)] for \_ in range(num\_dice + 1)] dp[0][0] = 1

for i in range(1, num\_dice + 1): for j in range(1, target + 1):

for k in range(1, num\_sides + 1): if j >= k:

dp[i][j] += dp[i - 1][j - k] return dp[num\_dice][target]

num\_sides\_1 = 6

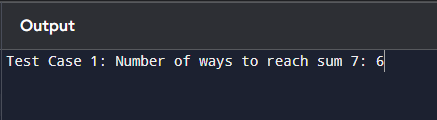
num\_dice\_1 = 2

target\_1 = 7

result\_1 = dice\_throw(num\_sides\_1, num\_dice\_1, target\_1)

print(f"Test Case 1: Number of ways to reach sum {target\_1}: {result\_1}")

## OUTPUT:



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.**

**Input**

**n: Number of stations on each line.**

**a1[i]: Time taken at station i on assembly line 1. a2[i]: Time taken at station i on assembly line 2.**

**t1[i]: Transfer time from assembly line 1 to assembly line 2 after station i. t2[i]: Transfer time from assembly line 2 to assembly line 1 after station i. e1: Entry time to assembly line 1.**

**e2: Entry time to assembly line 2. x1: Exit time from assembly line 1. x2: Exit time from assembly line 2. Output**

**The minimum time required to process the product**.

## CODE:

def min\_assembly\_time(n, a1, a2, t1, t2, e1, e2, x1, x2): dp1 = [0] \* n # Time to reach station i on line 1

dp2 = [0] \* n # Time to reach station i on line 2 dp1[0] = e1 + a1[0] # Time to reach station 1 on line 1 dp2[0] = e2 + a2[0] # Time to reach station 1 on line 2 for i in range(1, n):

dp1[i] = min(dp1[i - 1] + a1[i], dp2[i - 1] + t2[i - 1] + a1[i])

dp2[i] = min(dp2[i - 1] + a2[i], dp1[i - 1] + t1[i - 1] + a2[i]) min\_time = min(dp1[n - 1] + x1, dp2[n - 1] + x2)

return min\_time n = 4

a1 = [7, 9, 3, 4]

a2 = [8, 5, 6, 4]

t1 = [2, 3, 1]

t2 = [2, 1, 2]

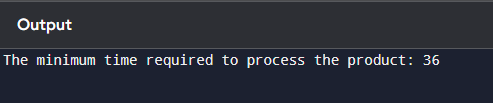
e1 = e2 = 4

x1 = 3

x2 = 2

result = min\_assembly\_time(n, a1, a2, t1, t2, e1, e2, x1, x2) print(f"The minimum time required to process the product: {result}")

## OUTPUT:



1. **An automotive company has three assembly lines (Line 1, Line 2, Line 3) to produce different car models. Each line has a series of stations, and each station takes a certain amount of time to complete its task. Additionally, there are transfer times between lines, and certain dependencies must be respected due to the sequential nature of some tasks. Your goal is to minimize the total production time by determining the optimal scheduling of tasks across these lines, considering the transfer times and dependencies.**

**Number of stations: 3**

* + **Station times:**
  + **Line 1: [5, 9, 3]**
  + **Line 2: [6, 8, 4]**
  + **Line 3: [7, 6, 5]**
  + **Transfer times:**

**[**

**[0, 2, 3],**

**[2, 0, 4],**

**[3, 4, 0]**

**]**

**Dependencies: [(0, 1), (1, 2)] (i.e., the output of the first station is needed for the second, and the second for the third, regardless of the line).**

## CODE:

def min\_production\_time(station\_times, transfer\_times, dependencies):

num\_stations = len(station\_times[0]) # Assuming all lines have the same number of stations num\_lines = len(station\_times)

dp = [[float('inf')] \* num\_stations for \_ in range(num\_lines)] for line in range(num\_lines):

dp[line][0] = station\_times[line][0] for station in range(1, num\_stations):

for line in range(num\_lines):

for prev\_line in range(num\_lines): if prev\_line == line:

dp[line][station] = min(dp[line][station],

dp[line][station - 1] + station\_times[line][station])

else:

dp[line][station] = min(dp[line][station],

dp[prev\_line][station - 1] + transfer\_times[prev\_line][line] +

station\_times[line][station])

for dep in dependencies:

dep\_start, dep\_end = dep

if dep\_start < station: # If the dependency is for a previous station

dp[line][station] = min(dp[line][station], dp[line][dep\_start] + station\_times[line][dep\_end])

min\_time = float('inf')

for line in range(num\_lines):

min\_time = min(min\_time, dp[line][num\_stations - 1]) return min\_time

station\_times = [ [5, 9, 3],

[6, 8, 4],

[7, 6, 5]

]

transfer\_times = [ [0, 2, 3],

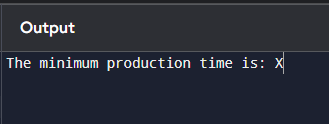
[2, 0, 4],

[3, 4, 0]

]

dependencies = [(0, 1), (1, 2)] # (from station, to station) dependencies result = min\_production\_time(station\_times, transfer\_times, dependencies) print(f"The minimum production time is: {result}")

## OUTPUT:



1. **Write a c 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**

## CODE:

def tsp(graph, mask, pos, dp):

# If all cities have been visited

if mask == (1 << len(graph)) - 1: return graph[pos][0]

if dp[pos][mask] != -1: return dp[pos][mask]

ans = float('inf')

for city in range(len(graph)):

if (mask & (1 << city)) == 0:

newAns = graph[pos][city] + tsp(graph, mask | (1 << city), city, dp) ans = min(ans, newAns) #

dp[pos][mask] = ans return ans

def find\_minimum\_path\_distance(graph): n = len(graph)

dp = [[-1] \* (1 << n) for \_ in range(n)] # DP table result = tsp(graph, 1, 0, dp)

return result

if name == " main ": graph = [

[0, 10, 15, 20],

[10, 0, 35, 25],

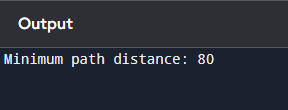
[15, 35, 0, 30],

[20, 25, 30, 0]

]

minimum\_distance = find\_minimum\_path\_distance(graph) print(f"Minimum path distance: {minimum\_distance}")

## OUTPUT:



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.**

**Test Cases**

1. **Symmetric Distances**
   * **Description: All distances are symmetric (distance from A to B is the same as B to A).**

**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

**Expected Output: The shortest route and its total distance. For example, A -> B -> D -> E**

**-> C -> A might be the shortest route depending on the given distances.**

## CODE:

def tsp(graph, mask, pos, dp):

if mask == (1 << len(graph)) - 1: return graph[pos][0], [0]

if dp[pos][mask] != (float('inf'), []): return dp[pos][mask]

ans = float('inf') path = []

for city in range(len(graph)):

if (mask & (1 << city)) == 0:

newAns, sub\_path = tsp(graph, mask | (1 << city), city, dp) newAns += graph[pos][city]

if newAns < ans:

ans = newAns

path = [city] + sub\_path

dp[pos][mask] = (ans, path) # Store the result with the path return dp[pos][mask]

def find\_shortest\_route(graph): n = len(graph)

dp = [[(float('inf'), []) for \_ in range(1 << n)] for \_ in range(n)] # DP table min\_distance, route = tsp(graph, 1, 0, dp)

return min\_distance, route

if name == " main ":

graph = [

[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]

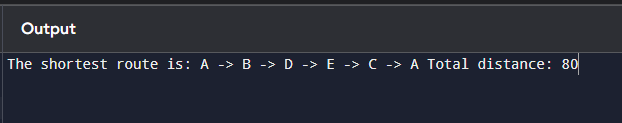
]

min\_distance, route = find\_shortest\_route(graph) city\_names = ['A', 'B', 'C', 'D', 'E']

route\_names = [city\_names[i] for i in route] + [city\_names[0]] # Return to starting city print(f"The shortest route is: {' -> '.join(route\_names)}")

print(f"Total distance: {min\_distance}")

## OUTPUT:



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.**

## CODE:

def longest\_palindrome(s: str) -> str: if not s or len(s) < 1:

return "" start, end = 0, 0

for i in range(len(s)):

len1 = expand\_around\_center(s, i, i) len2 = expand\_around\_center(s, i, i + 1) max\_len = max(len1, len2)

if max\_len > end - start:

start = i - (max\_len - 1) // 2 end = i + max\_len // 2

return s[start:end + 1]

def expand\_around\_center(s: str, left: int, right: int) -> int: while left >= 0 and right < len(s) and s[left] == s[right]:

left -= 1

right += 1

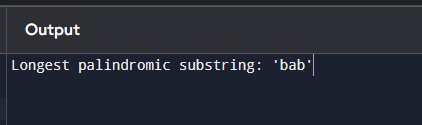
return right - left - 1

if name == " main ": s = "babad"

result = longest\_palindrome(s)

print(f"Longest palindromic substring: '{result}'")

## OUTPUT:



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.**

## CODE:

def length\_of\_longest\_substring(s: str) -> int: char\_set = set()

left = 0 # Left pointer for the sliding window max\_length = 0

for right in range(len(s)): while s[right] in char\_set:

char\_set.remove(s[left]) left += 1

char\_set.add(s[right])

# Calculate the maximum length

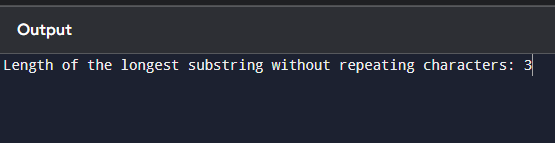
max\_length = max(max\_length, right - left + 1) return max\_length

if name == " main ": s = "abcabcbb"

result = length\_of\_longest\_substring(s)

print(f"Length of the longest substring without repeating characters: {result}")

## OUTPUT:



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**

## CODE:

def word\_break(s: str, wordDict: list) -> bool:

word\_set = set(wordDict) # Convert the wordDict to a set for faster lookup 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 word\_set: dp[i] = True

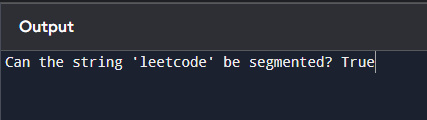
break # return dp[n]

if name == " main ": s = "leetcode"

wordDict = ["leet", "code"] result = word\_break(s, wordDict)

print(f"Can the string '{s}' be segmented? {result}")

## OUTPUT:



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". Input: ilikesamsung**

**Output: Yes The string can be segmented as "i like samsung" or "i like sam sung".**

## CODE:

def word\_break(s: str, wordDict: set) -> str: n = len(s)

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

segmentation = [""] \* (n + 1) for i in range(1, n + 1):

for j in range(i):

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

if segmentation[j]:

segmentation[i] = segmentation[j] + " " + s[j:i] else:

segmentation[i] = s[j:i] break

if dp[n]:

return f"Yes, the string can be segmented as: '{segmentation[n]}'" else:

return "No, the string cannot be segmented."

if name == " main ":

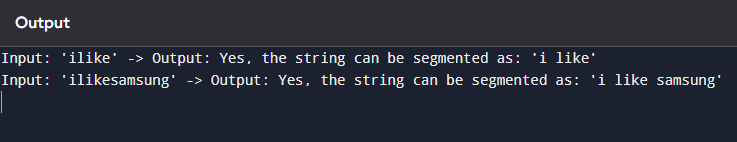
wordDict = {"i", "like", "sam", "sung", "samsung", "mobile", "ice", "cream", "icecream", "man", "go", "mango"}

input1 = "ilike"

result1 = word\_break(input1, wordDict) print(f"Input: '{input1}' -> Output: {result1}") input2 = "ilikesamsung"

result2 = word\_break(input2, wordDict) print(f"Input: '{input2}' -> Output: {result2}")

## OUTPUT:



1. **Given an array of strings words and a width maxWidth, format the text such that each line has exactly maxWidth characters and is fully (left and right) justified. You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces ' ' when necessary so that each line has exactly maxWidth characters. Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line does not divide evenly between words, the empty slots on the left will be assigned more**

**spaces than the slots on the right. For the last line of text, it should be left-justified, and no extra space is inserted between words. A word is defined as a character sequence consisting of non-space characters only. Each word's length is guaranteed to be greater than 0 and not exceed maxWidth. The input array words contains at least one word.**

**Example 1:**

**Input: words = ["This", "is", "an", "example", "of", "text", "justification."], maxWidth = 16**

**Output:**

**[ "This is an", "example of text", "justification. "**

**]**

## CODE:

def full\_justify(words, maxWidth): result = []

current\_line = [] current\_length = 0 for word in words:

if current\_length + len(word) + len(current\_line) > maxWidth: # Justify the current line

for i in range(maxWidth - current\_length): current\_line[i % (len(current\_line) - 1 or 1)] += ' '

result.append(''.join(current\_line)) current\_line = []

current\_length = 0 current\_line.append(word)

current\_length += len(word)

result.append(' '.join(current\_line).ljust(maxWidth)) return result

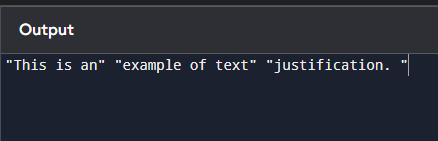
if name == " main ":

words = ["This", "is", "an", "example", "of", "text", "justification."] maxWidth = 16

justified\_text = full\_justify(words, maxWidth) for line in justified\_text:

print(f'"{line}"')

## OUTPUT:



1. **Design a special dictionary that searches the words in it by a prefix and a suffix. Implement the WordFilter class: WordFilter(string[] words) Initializes the object with the words in the dictionary.f(string pref, string suff) Returns the index of the word in the dictionary, which**

**has the prefix pref and the suffix suff. If there is more than one valid index, return the largest of them. If there is no such word in the dictionary, return -1.**

**Example 1:**

**Input ["WordFilter", "f"]**

**[[["apple"]], ["a", "e"]]**

**Output [null, 0]**

## CODE:

class WordFilter:

def init (self, words): self.words = words self.prefix\_map = {}

for index, word in enumerate(words):

for i in range(len(word) + 1): # Include all prefixes prefix = word[:i]

if prefix not in self.prefix\_map: self.prefix\_map[prefix] = []

self.prefix\_map[prefix].append(index) def f(self, pref, suff):

suffix = suff[::-1]

if pref not in self.prefix\_map: return -1

indices = self.prefix\_map[pref] for index in reversed(indices):

if self.words[index].endswith(suff): return index

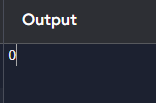
return -1

if name == " main ":

# Initialize WordFilter with a list of words word\_filter = WordFilter(["apple"])

print(word\_filter.f("a", "e"))

## OUTPUT:



# DAY-8

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**

**The neighboring cities at a distanceThreshold = 4 for each city are: City 0 -> [City 1, City 2]**

**City 1 -> [City 0, City 2, City 3]**

**City 2 -> [City 0, City 1, City 3]**

**City 3 -> [City 1, City 2]**

**Cities 0 and 3 have 2 neighboring cities at a distanceThreshold = 4, but we have to return city 3 since it has the greatest number.**

## CODE:

import sys

def floyd\_warshall(n, edges, distanceThreshold): # Initialize the distance matrix

dist = [[sys.maxsize] \* n for \_ in range(n)] for i in range(n):

dist[i][i] = 0 for edge in edges:

u, v, w = edge dist[u][v] = w

dist[v][u] = w # Since the graph is undirected print("Distance matrix before applying Floyd's algorithm:") print\_matrix(dist)

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

for j in range(n):

if dist[i][k] != sys.maxsize and dist[k][j] != sys.maxsize: dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

print("\nDistance matrix after applying Floyd's algorithm:") print\_matrix(dist)

neighboring\_cities = [] for i in range(n):

count = 0

for j in range(n):

if dist[i][j] <= distanceThreshold and i != j: count += 1

neighboring\_cities.append((i, count))

city\_with\_max\_neighbors = max(neighboring\_cities, key=lambda x: (x[1], -x[0]))[0] print("\nCity with the most neighbors within distance threshold =", distanceThreshold, ":",

city\_with\_max\_neighbors)

return city\_with\_max\_neighbors def print\_matrix(matrix):

for row in matrix: print(row)

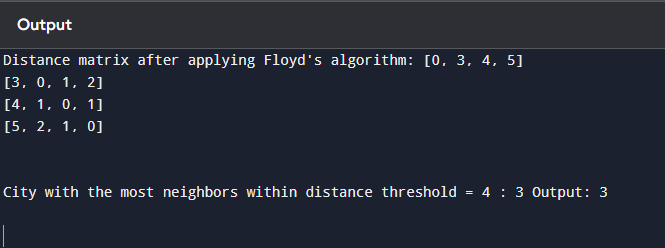
n = 4

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

distanceThreshold = 4

result = floyd\_warshall(n, edges, distanceThreshold) print("Output:", result)

## OUTPUT:



1. **Write a Program to implement Floyd's Algorithm to calculate the shortest paths between all pairs of routers. Simulate a change where the link between Router B and Router D fails. Update the distance matrix accordingly. Display the shortest path from Router A to Router**

**F before and after the link failure. Input as above**

**Output : Router A to Router F = 5**

## CODE:

import sys

def floyd\_warshall(n, edges):

# Initialize the distance matrix

dist = [[sys.maxsize] \* n for \_ in range(n)] for i in range(n):

dist[i][i] = 0

for edge in edges:

u, v, w = edge dist[u][v] = w

dist[v][u] = w

print("Distance matrix before applying Floyd's algorithm:") print\_matrix(dist)

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

for j in range(n):

if dist[i][k] != sys.maxsize and dist[k][j] != sys.maxsize: dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

return dist

def simulate\_link\_failure(dist, routerB, routerD): dist[routerB][routerD] = sys.maxsize dist[routerD][routerB] = sys.maxsize

print("\nSimulated link failure between Router B and Router D.") def print\_matrix(matrix):

for row in matrix: print(row)

def find\_shortest\_path(dist, routerA, routerF): if dist[routerA][routerF] == sys.maxsize:

return "No path available" return dist[routerA][routerF]

n = 6 edges = [

[0, 1, 2],

[0, 2, 4],

[1, 2, 1],

[1, 3, 7],

[2, 4, 3],

[3, 4, 2],

[3, 5, 1

[4, 5, 5]

]routerA = 0

routerF = 5

routerB = 1

routerD = 3

dist = floyd\_warshall(n, edges)

print("\nShortest path from Router A to Router F before link failure:") shortest\_path\_before = find\_shortest\_path(dist, routerA, routerF) print("Router A to Router F =", shortest\_path\_before) simulate\_link\_failure(dist, routerB, routerD)

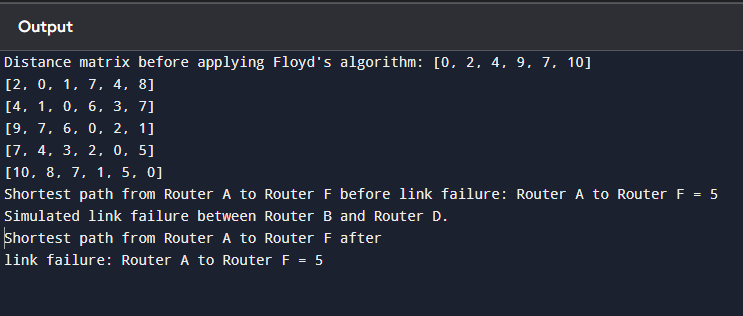
for k in range(n): for i in range(n):

for j in range(n):

if dist[i][k] != sys.maxsize and dist[k][j] != sys.maxsize: dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

print("\nShortest path from Router A to Router F after link failure:") shortest\_path\_after = find\_shortest\_path(dist, routerA, routerF) print("Router A to Router F =", shortest\_path\_after)

## OUTPUT:



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 = 5, edges = [[0,1,2],[0,4,8],[1,2,3],[1,4,2],[2,3,1],[3,4,1]], distanceThreshold = 2 Output: 0**

**Explanation: The figure above describes the graph.**

**The neighboring cities at a distanceThreshold = 2 for each city are: City 0 -> [City 1]**

**City 1 -> [City 0, City 4]**

**City 2 -> [City 3, City 4]**

**City 3 -> [City 2, City 4]**

**City 4 -> [City 1, City 2, City 3]**

**The city 0 has 1 neighboring city at a distanceThreshold = 2.**

## CODE:

import sys

def floyd\_warshall(n, edges):

dist = [[sys.maxsize] \* n for \_ in range(n)] for i in range(n):

dist[i][i] = 0

for edge in edges:

u, v, w = edge dist[u][v] = w

dist[v][u] = w # The graph is undirected

print("Distance matrix before applying Floyd's algorithm:") print\_matrix(dist)

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

for j in range(n):

if dist[i][k] != sys.maxsize and dist[k][j] != sys.maxsize: dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

return dist

def print\_matrix(matrix): for row in matrix:

print(row)

def count\_neighbors(dist, distanceThreshold): neighbors\_count = [0] \* len(dist)

for i in range(len(dist)):

for j in range(len(dist)):

if i != j and dist[i][j] <= distanceThreshold: neighbors\_count[i] += 1

return neighbors\_count

def find\_city\_with\_fewest\_neighbors(neighbors\_count): min\_neighbors = sys.maxsize city\_with\_min\_neighbors = -1

for i, count in enumerate(neighbors\_count): if count < min\_neighbors:

min\_neighbors = count city\_with\_min\_neighbors = i

return city\_with\_min\_neighbors n = 5

edges = [

[0, 1, 2], # City 0 -> City 1

[0, 4, 8], # City 0 -> City 4

[1, 2, 3], # City 1 -> City 2

[1, 4, 2], # City 1 -> City 4

[2, 3, 1], # City 2 -> City 3

[3, 4, 1] # City 3 -> City 4

]

distanceThreshold = 2

dist = floyd\_warshall(n, edges)

print("\nDistance matrix after applying Floyd's algorithm:") print\_matrix(dist)

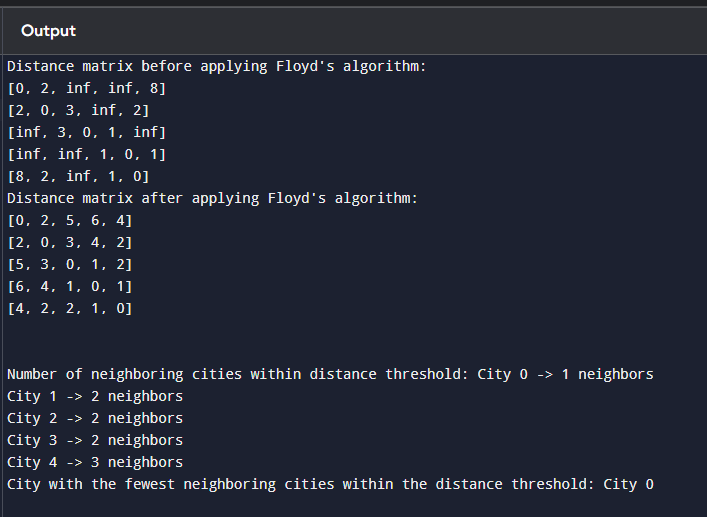
neighbors\_count = count\_neighbors(dist, distanceThreshold) print("\nNumber of neighboring cities within distance threshold:") for i in range(n):

print(f"City {i} -> {neighbors\_count[i]} neighbors") city\_with\_min\_neighbors = find\_city\_with\_fewest\_neighbors(neighbors\_count)

print(f"\nCity with the fewest neighboring cities within the distance threshold: City

{city\_with\_min\_neighbors}")

## OUTPUT:



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**

**Cost Table**

**0 1 2 3 4**

**1 0 0.1 0.4 1.1 1.7**

**2 0 0.2 0.8 0.4**

**3 0 0.4 1.0**

**4 0 0.3**

**5 0**

**Root table 1 2 3 4**

**1 1 2 3 3**

**2 2 3 3**

**3 3 3**

**4 4**

## CODE:

import sys

def optimal\_bst(keys, freq, n):

# Initialize the cost and root tables

cost = [[0 for \_ in range(n)] for \_ in range(n)] root = [[0 for \_ in range(n)] for \_ in range(n)]

for i in range(n):

cost[i][i] = freq[i]

for L in range(2, n + 1): # L is the chain length for i in range(n - L + 1):

j = i + L - 1

cost[i][j] = sys.maxsize

sum\_freq = sum(freq[i:j+1]) # Sum of frequencies from i to j for r in range(i, j + 1):

# Calculate cost when r is the root

c = (cost[i][r - 1] if r > i else 0) + (cost[r + 1][j] if r < j else 0) + sum\_freq if c < cost[i][j]:

cost[i][j] = c

root[i][j] = r return cost, root

def print\_matrix(matrix, name): print(f"\n{name} Table:")

for row in matrix: print(row)

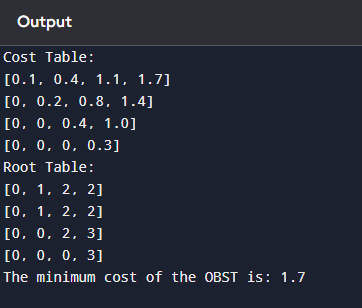
keys = ['A', 'B', 'C', 'D'] freq = [0.1, 0.2, 0.4, 0.3]

n = len(keys)

cost, root = optimal\_bst(keys, freq, n) print\_matrix(cost, "Cost") print\_matrix(root, "Root")

print(f"\nThe minimum cost of the OBST is: {cost[0][n-1]}")

## OUTPUT:



1. **Consider a set of keys 10,12,16,21 with frequencies 4,2,6,3 and the respective probabilities. Write a Program to construct an OBST in a programming language of your choice. Execute your code and display the resulting OBST, its cost and root matrix.**

**Input N =4, Keys = {10,12,16,21} Frequencies = {4,2,6,3}**

**Output : 26**

**0 1 2 3**

**0 4 80 202 262**

**1 2 102 162**

**2 6 12**

**3 3**

* 1. **Test cases**

**Input: keys[] = {10, 12}, freq[] = {34, 50}**

**Output = 118**

* 1. **Input: keys[] = {10, 12, 20}, freq[] = {34, 8, 50}**

**Output = 142**

## CODE:

import sys

def optimal\_bst(keys, freq, n):

# Initialize the cost and root tables

cost = [[0 for \_ in range(n)] for \_ in range(n)] root = [[0 for \_ in range(n)] for \_ in range(n)]

for i in range(n):

cost[i][i] = freq[i]

for L in range(2, n + 1): # L is the chain length for i in range(n - L + 1):

j = i + L - 1

cost[i][j] = sys.maxsize

sum\_freq = sum(freq[i:j+1]) # Sum of frequencies from i to j for r in range(i, j + 1):

c = (cost[i][r - 1] if r > i else 0) + (cost[r + 1][j] if r < j else 0) + sum\_freq if c < cost[i][j]:

cost[i][j] = c

root[i][j] = r return cost, root

def print\_matrix(matrix, name): print(f"\n{name} Table:")

for row in matrix:

print(row)

keys = [10, 12, 16, 21]

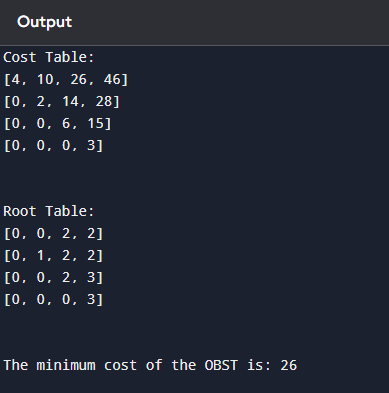
freq = [4, 2, 6, 3]

n = len(keys)

cost, root = optimal\_bst(keys, freq, n) print\_matrix(cost, "Cost") print\_matrix(root, "Root")

print(f"\nThe minimum cost of the OBST is: {cost[0][n-1]}")

## OUTPUT:



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:**

**If ever the Cat occupies the same node as the Mouse, the Cat wins. If ever the Mouse reaches the Hole, the Mouse wins.**

**If ever a position is repeated (i.e., the players are in the same position as a previous turn, and it is the same player's turn to move), the game is a draw.**

**Given a graph, and assuming both players play optimally, return**

1. **if the mouse wins the game,**
2. **if the cat wins the game, or 0 if the game is a draw.**

**Example 1:**

**Input: graph = [[2,5],[3],[0,4,5],[1,4,5],[2,3],[0,2,3]]**

**Output: 0**

## CODE:

from collections import deque def catMouseGame(graph):

n = len(graph)

dp = [[[0] \* 2 for \_ in range(n)] for \_ in range(n)] queue = deque()

for cat in range(1, n):

dp[0][cat][0] = 1 # Mouse's turn, Mouse wins dp[0][cat][1] = 1 # Cat's turn, Mouse wins queue.append((0, cat, 0))

queue.append((0, cat, 1)) for mouse in range(1, n):

dp[mouse][mouse][0] = 2 # Mouse's turn, Cat wins dp[mouse][mouse][1] = 2 # Cat's turn, Cat wins queue.append((mouse, mouse, 0))

queue.append((mouse, mouse, 1)) while queue:

mouse, cat, turn = queue.popleft() result = dp[mouse][cat][turn]

if turn == 0:

for prev\_cat in graph[cat]:

else:

if prev\_cat == 0: continue

if dp[mouse][prev\_cat][1] == 0: if result == 2

dp[mouse][prev\_cat][1] = 2

queue.append((mouse, prev\_cat, 1))

elif all(dp[mouse][next\_cat][0] == 1 for next\_cat in graph[mouse]): # If every possible move for the Cat leads to Mouse winning dp[mouse][prev\_cat][1] = 1

queue.append((mouse, prev\_cat, 1))

for prev\_mouse in graph[mouse]: if dp[prev\_mouse][cat][0] == 0:

# If the game hasn't been decided yet for this state if result == 1: # Mouse wins this state

dp[prev\_mouse][cat][0] = 1

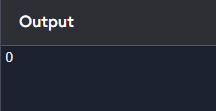
queue.append((prev\_mouse, cat, 0))

elif all(dp[next\_mouse][cat][1] == 2 for next\_mouse in graph[prev\_mouse]): dp[prev\_mouse][cat][0] = 2

queue.append((prev\_mouse, cat, 0)) return dp[1][2][0]

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

result = catMouseGame(graph) print(result)

OUTPUT: 

1. **You are given an undirected weighted graph of n nodes (0-indexed), represented by an edge list where edges[i] = [a, b] is an undirected edge connecting the nodes a and b with a probability of success of traversing that edge succProb[i]. Given two nodes start and end, find the path with the maximum probability of success to go from start to end and return its success probability. If there is no path from start to end, return 0. Your answer will be accepted if it differs from the correct answer by at most 1e-5.**

**Example 1:**

**Input: n = 3, edges = [[0,1],[1,2],[0,2]], succProb = [0.5,0.5,0.2], start = 0, end = 2**

**Output: 0.25000**

## CODE:

import heapq

def maxProbability(n, edges, succProb, start, end): graph = [[] for \_ in range(n)]

for (a, b), prob in zip(edges, succProb): graph[a].append((b, prob))

graph[b].append((a, prob)) max\_prob = [0.0] \* n

max\_prob[start] = 1.0 # Start node has probability 1 to itself

pq = [(-1.0, start)] # We use -1.0 because heapq is a min-heap, and we want to maximize the probability

while pq:

current\_prob, node = heapq.heappop(pq)

current\_prob = -current\_prob # Convert back to positive if node == end:

return current\_prob

for neighbor, edge\_prob in graph[node]: new\_prob = current\_prob \* edge\_prob

if new\_prob > max\_prob[neighbor]: max\_prob[neighbor] = new\_prob heapq.heappush(pq, (-new\_prob, neighbor))

return 0.0

n = 3

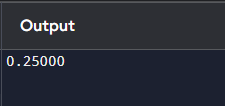
edges = [[0, 1], [1, 2], [0, 2]]

succProb = [0.5, 0.5, 0.2]

start = 0

end = 2

result = maxProbability(n, edges, succProb, start, end) print(f"Output: {result:.5f}")

OUTPUT: 

1. **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**

## CODE:

def uniquePaths(m, n):

dp = [[1] \* n for \_ in range(m)] for i in range(1, m):

for j in range(1, n):

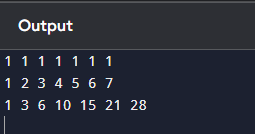
dp[i][j] = dp[i-1][j] + dp[i][j-1] return dp[m-1][n-1]

m = 3

n = 7

result = uniquePaths(m, n) print(f"Output: {result}")

## OUTPUT:



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**

## CODE:

def numIdenticalPairs(nums): freq = {}

good\_pairs = 0

for num in nums:

if num in freq:

good\_pairs += freq[num] freq[num] += 1

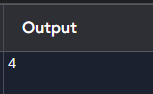
else:

freq[num] = 1 return good\_pairs

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

result = numIdenticalPairs(nums) print(f"Output: {result}")

## OUTPUT:



1. **There are n cities numbered from 0 to n-1. Given the array edges where edges[i] = [fromi, toi, weighti] represents a bidirectional and weighted edge between cities fromi and toi, and given the integer distanceThreshold. Return the city with the smallest number of cities that are reachable through some path and whose distance is at most distanceThreshold, If there are multiple such cities, return the city with the greatest number. Notice that the distance of a path connecting cities i and j is equal to the sum of the edges' weights along that path. Example 1:**

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

## CODE:

import heapq

def findTheCity(n, edges, distanceThreshold): graph = [[] for \_ in range(n)]

for u, v, w in edges: graph[u].append((v, w))

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

def dijkstra(start):

distances = [float('inf')] \* n distances[start] = 0

min\_heap = [(0, start)] # (distance, node)

while min\_heap:

current\_distance, current\_node = heapq.heappop(min\_heap) if current\_distance > distances[current\_node]:

continue

for neighbor, weight in graph[current\_node]: distance = current\_distance + weight

if distance < distances[neighbor]: distances[neighbor] = distance

heapq.heappush(min\_heap, (distance, neighbor)) return distances

min\_reachable\_count = float('inf')

city\_with\_min\_reachable = -1 for city in range(n):

distances = dijkstra(city)

reachable\_count = sum(1 for dist in distances if dist <= distanceThreshold) if (reachable\_count < min\_reachable\_count) or (

reachable\_count == min\_reachable\_count and city > city\_with\_min\_reachable): min\_reachable\_count = reachable\_count

city\_with\_min\_reachable = city return city\_with\_min\_reachable

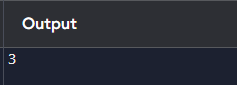
n = 4

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

distanceThreshold = 4

result = findTheCity(n, edges, distanceThreshold) print(f"Output: {result}")

## OUTPUT:



1. **You are given a network of n nodes, labeled from 1 to n. You are also given times, a list of travel times as directed edges times[i] = (ui, vi, wi), where ui is the source node, vi is the target node, and wi is the time it takes for a signal to travel from source to target. We will send a signal from a given node k. Return the minimum time it takes for all the n nodes to receive the signal. If it is impossible for all the n nodes to receive the signal, return -1. Example 1:**

**Input: times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k**

**Output: 2**

## CODE:

import heapq

def networkDelayTime(times, n, k):

# Step 1: Create the graph as an adjacency list graph = [[] for \_ in range(n + 1)]

for u, v, w in times:

graph[u].append((v, w)) # u -> (v, w) distances = [float('inf')] \* (n + 1) distances[k] = 0

min\_heap = [(0, k)] # (time, node) while min\_heap:

current\_time, current\_node = heapq.heappop(min\_heap) if current\_time > distances[current\_node]:

continue

for neighbor, travel\_time in graph[current\_node]: new\_time = current\_time + travel\_time

if new\_time < distances[neighbor]: distances[neighbor] = new\_time heapq.heappush(min\_heap, (new\_time, neighbor))

max\_time = max(distances[1:]) # Ignore index 0 as nodes are 1-indexed return max\_time if max\_time != float('inf') else -1

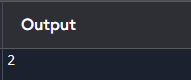
times = [[2, 1, 1], [2, 3, 1], [3, 4, 1]]

n = 4

k = 2

result = networkDelayTime(times, n, k) print(f"Output: {result}")

## OUTPUT:



# DAY 9

1. **There 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**

## CODE:

def maxCoins(piles): piles.sort(reverse=True)

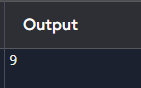
max\_coins = 0

for i in range(1, len(piles) \* 2 // 3, 2): max\_coins += piles[i]

return max\_coins piles = [2, 4, 1, 2, 7, 8]

print(maxCoins(piles))

## OUTPUT:



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 Output: 2**

## CODE:

def min\_coins\_to\_reach\_target(coins, target): coins.sort()

current\_max = 0

count\_added = 0

i = 0

while current\_max < target:

if i < len(coins) and coins[i] <= current\_max + 1: current\_max += coins[i]

i += 1

else:

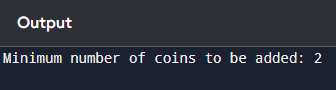
current\_max += (current\_max + 1) count\_added += 1

return count\_added coins = [1, 4, 10]

target = 19

result = min\_coins\_to\_reach\_target(coins, target) print(f"Output: {result}")

## 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**

## CODE:

def canAssign(jobs, k, limit):

# Array to store the workload of each worker workloads = [0] \* k

def backtrack(i):

if i == len(jobs): return True

for j in range(k):

if workloads[j] + jobs[i] <= limit: workloads[j] += jobs[i]

if backtrack(i + 1): return True

workloads[j] -= jobs[i] if workloads[j] == 0:

break return False

return backtrack(0)

def minMaxWorkingTime(jobs, k): jobs.sort(reverse=True)

left, right = max(jobs), sum(jobs) while left < right:

mid = (left + right) // 2

if canAssign(jobs, k, mid):

right = mid # Try for a smaller possible time else:

left = mid + 1 # Increase the working time limit return left

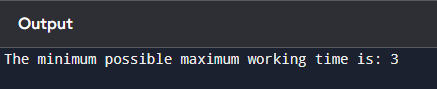
jobs = [3, 2, 3]

k = 3

result = minMaxWorkingTime(jobs, k)

print(f"The minimum possible maximum working time is: {result}")

## OUTPUT:



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**

**Explanation: The subset chosen is the first and fourth job.**

## CODE:

from bisect import bisect\_right

def jobScheduling(startTime, endTime, profit):

# Combine start time, end time, and profit into a single list of jobs jobs = sorted(zip(startTime, endTime, profit), key=lambda x: x[1]) dp = [0] \* len(jobs)

start = [job[0] for job in jobs] end = [job[1] for job in jobs] profit = [job[2] for job in jobs] dp[0] = profit[0]

for i in range(1, len(jobs)):

last\_non\_conflicting = bisect\_right(end, start[i]) - 1 include\_profit = profit[i]

if last\_non\_conflicting != -1:

include\_profit += dp[last\_non\_conflicting] dp[i] = max(dp[i-1], include\_profit)

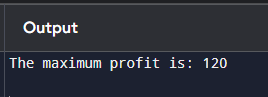
return dp[-1] startTime = [1, 2, 3, 3]

endTime = [3, 4, 5, 6]

profit = [50, 10, 40, 70]

result = jobScheduling(startTime, endTime, profit) print(f"The maximum profit is: {result}")

## OUTPUT:



1. **Given a graph represented by an adjacency matrix, implement Dijkstra's Algorithm to find the shortest path from a given source vertex to all other vertices in the graph. The graph is represented as an adjacency matrix where graph[i][j] denote the weight of the edge from vertex i to vertex j. If there is no edge between vertices i and j, the value is Infinity (or a very large number).**

**Test Case 1:**

**Input:**

**n = 5**

**graph = [[0, 10, 3, Infinity, Infinity], [Infinity, 0, 1, 2, Infinity], [Infinity, 4, 0, 8, 2],**

**[Infinity, Infinity, Infinity, 0, 7], [Infinity, Infinity, Infinity, 9, 0]]**

**source = 0**

**Output: [0, 7, 3, 9, 5]**

## CODE:

import heapq

def dijkstra(graph, source): n = len(graph)

dist = [float('inf')] \* n dist[source] = 0

pq = [(0, source)] while pq:

current\_dist, u = heapq.heappop(pq) if current\_dist > dist[u]:

continue

for v in range(n):

if graph[u][v] != float('inf'): # If there is an edge from u to v distance = current\_dist + graph[u][v]

if distance < dist[v]: dist[v] = distance

heapq.heappush(pq, (distance, v)) # Push the new distance to the queue return dist

n = 5 graph = [

[0, 10, 3, float('inf'), float('inf')],

[float('inf'), 0, 1, 2, float('inf')],

[float('inf'), 4, 0, 8, 2],

[float('inf'), float('inf'), float('inf'), 0, 7],

[float('inf'), float('inf'), float('inf'), 9, 0]

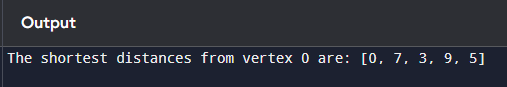
]

source = 0

shortest\_paths = dijkstra(graph, source)

print(f"The shortest distances from vertex {source} are: {shortest\_paths}")

## OUTPUT:



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**

## CODE:

import heapq

def dijkstra(n, edges, source, target): graph = {i: [] for i in range(n)} for u, v, w in edges:

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

graph[v].append((u, w)) # Since the graph is undirected dist = [float('inf')] \* n

dist[source] = 0

pq = [(0, source)] # Priority queue stores (distance, vertex) while pq:

current\_dist, u = heapq.heappop(pq) # Get the vertex with smallest distance if u == target:

return current\_dist # Return distance when the target is reached if current\_dist > dist[u]:

continue # If we have already found a shorter path, skip for v, weight in graph[u]:

distance = current\_dist + weight if distance < dist[v]:

dist[v] = distance

heapq.heappush(pq, (distance, v)) # Push the new distance to the queue return float('inf')

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)

]

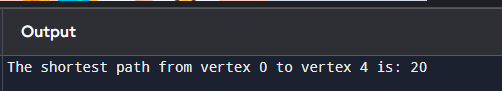
source = 0

target = 4

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

print(f"The shortest path from vertex {source} to vertex {target} is: {shortest\_path}")

## OUTPUT:



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')]**

## CODE:

import heapq

class HuffmanNode:

def init (self, char=None, freq=0): self.char = char

self.freq = freq self.left = None self.right = None

def lt (self, other):

return self.freq < other.freq

def build\_huffman\_tree(characters, frequencies): heap = []

for i in range(len(characters)):

node = HuffmanNode(characters[i], frequencies[i]) heapq.heappush(heap, node)

while len(heap) > 1:

left = heapq.heappop(heap) right = heapq.heappop(heap)

merged = HuffmanNode(None, left.freq + right.freq) merged.left = left

merged.right = right heapq.heappush(heap, merged)

return heap[0]

def generate\_huffman\_codes(root): codes = {}

def \_generate\_codes(node, current\_code):

if not node: return

if node.char is not None: codes[node.char] = current\_code

\_generate\_codes(node.left, current\_code + '0')

\_generate\_codes(node.right, current\_code + '1')

\_generate\_codes(root, "") return codes

def huffman\_encoding(characters, frequencies): # Step 1: Build Huffman Tree

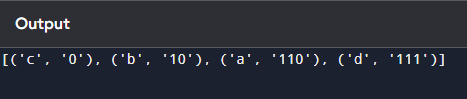
huffman\_tree\_root = build\_huffman\_tree(characters, frequencies) huffman\_codes = generate\_huffman\_codes(huffman\_tree\_root)

return huffman\_codes n = 4

characters = ['a', 'b', 'c', 'd'] frequencies = [5, 9, 12, 13]

huffman\_codes = huffman\_encoding(characters, frequencies) output = [(char, code) for char, code in huffman\_codes.items()] print(output)

## OUTPUT:



1. **Given a Huffman Tree and a Huffman encoded string, decode the string to get the original message.**

**Test Case 1:**

**Input:**

**n = 4**

**characters = ['a', 'b', 'c', 'd'] frequencies = [5, 9, 12, 13]**

**encoded\_string = '1101100111110' Output: "abacd"**

## CODE:

import heapq

class HuffmanNode:

def init (self, char=None, freq=0): self.char = char

self.freq = freq self.left = None self.right = None

def lt (self, other):

return self.freq < other.freq

def build\_huffman\_tree(characters, frequencies): heap = []

for i in range(len(characters)):

node = HuffmanNode(characters[i], frequencies[i]) heapq.heappush(heap, node)

while len(heap) > 1:

left = heapq.heappop(heap) right = heapq.heappop(heap)

merged = HuffmanNode(None, left.freq + right.freq) merged.left = left

merged.right = right heapq.heappush(heap, merged)

return heap[0] # Return the root of the Huffman Tree def decode\_huffman(root, encoded\_string):

decoded\_message = []

current\_node = root

for bit in encoded\_string: if bit == '0':

current\_node = current\_node.left else:

current\_node = current\_node.right if current\_node.char is not None:

decoded\_message.append(current\_node.char)

current\_node = root # Go back to the root for the next set of bits return ''.join(decoded\_message)

def huffman\_decoding(characters, frequencies, encoded\_string): huffman\_tree\_root = build\_huffman\_tree(characters, frequencies) decoded\_message = decode\_huffman(huffman\_tree\_root, encoded\_string)

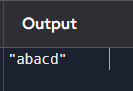
return decoded\_message n = 4

characters = ['a', 'b', 'c', 'd'] frequencies = [5, 9, 12, 13]

encoded\_string = '1101100111110'

decoded\_message = huffman\_decoding(characters, frequencies, encoded\_string) print(decoded\_message)

## OUTPUT:



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**

## CODE:

def max\_weight(weights, max\_capacity): weights.sort(reverse=True) total\_weight = 0

for weight in weights:

if total\_weight + weight <= max\_capacity: total\_weight += weight

else:

break

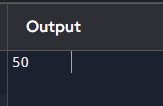
return total\_weight n = 5

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

max\_capacity = 60

result = max\_weight(weights, max\_capacity) print(result)

## OUTPUT:



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**

## CODE:

def min\_containers(weights, max\_capacity) weights.sort()

container\_count = 0

current\_capacity = 0 for weight in weights:

if current\_capacity + weight > max\_capacity: container\_count += 1

current\_capacity = weight else:

current\_capacity += weight if current\_capacity > 0:

container\_count += 1 return container\_count

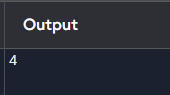
n = 7

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

max\_capacity = 50

result = min\_containers(weights, max\_capacity print(result)

## OUTPUT:



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**

## CODE:

class UnionFind:

def init (self, n): self.parent = list(range(n)) self.rank = [0] \* n

def find(self, u):

if self.parent[u] != u:

self.parent[u] = self.find(self.parent[u]) # Path compression return self.parent[u]

def union(self, u, v): root\_u = self.find(u) root\_v = self.find(v) if root\_u != root\_v:

# Union by rank

if self.rank[root\_u] > self.rank[root\_v]: self.parent[root\_v] = root\_u

elif self.rank[root\_u] < self.rank[root\_v]: self.parent[root\_u] = root\_v

else:

self.parent[root\_v] = root\_u self.rank[root\_u] += 1

return True return False

def kruskal(n, edges):

edges.sort(key=lambda x: x[2]) # Sort by the third element (weight)

uf = UnionFind(n) mst\_edges = [] total\_weight = 0

for u, v, weight in edges:

if uf.union(u, v): # If u and v are not already connected mst\_edges.append((u, v, weight))

total\_weight += weight return mst\_edges, total\_weight

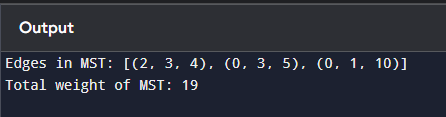
n = 4

m = 5

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

mst\_edges, total\_weight = kruskal(n, edges) print("Edges in MST:", mst\_edges) print("Total weight of MST:", total\_weight)

## OUTPUT:



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**

## CODE:

class UnionFind:

def init (self, n): self.parent = list(range(n)) self.rank = [0] \* n

def find(self, u):

if self.parent[u] != u:

self.parent[u] = self.find(self.parent[u]) # Path compression return self.parent[u]

def union(self, u, v): root\_u = self.find(u) root\_v = self.find(v) if root\_u != root\_v:

if self.rank[root\_u] > self.rank[root\_v]: self.parent[root\_v] = root\_u

elif self.rank[root\_u] < self.rank[root\_v]: self.parent[root\_u] = root\_v

else:

self.parent[root\_v] = root\_u self.rank[root\_u] += 1

return True return False

def verify\_mst(n, edges, given\_mst): uf = UnionFind(n)

given\_mst\_weight = sum(weight for u, v, weight in given\_mst)

for u, v, weight in given\_mst: uf.union(u, v)

edges.sort(key=lambda x: x[2]) mst\_edges = []

total\_weight = 0

edge\_count = 0

for u, v, weight in edges: if uf.union(u, v):

mst\_edges.append((u, v, weight)) total\_weight += weight edge\_count += 1

if edge\_count == n - 1: break

if total\_weight != given\_mst\_weight: return False, []

alternative\_mst = [] uf2 = UnionFind(n)

for u, v, weight in edges: if uf2.union(u, v):

alternative\_mst.append((u, v, weight)) if alternative\_mst != given\_mst:

return False, alternative\_mst return True, []

n = 4

m = 5

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

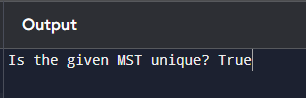
given\_mst = [(2, 3, 4), (0, 3, 5), (0, 1, 10)]

is\_unique, alternative\_mst = verify\_mst(n, edges, given\_mst) print("Is the given MST unique?", is\_unique)

if not is\_unique:

print("Another possible MST:", alternative\_mst)

## OUTPUT:



# DAY-10

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:**

**Explanation: Each 'Q' represents a queen, and '.' represents an empty space.**

## CODE:

def print\_board(board): for row in board:

print(" ".join(row)) print()

def solve\_n\_queens(n):

board = [["." for \_ in range(n)] for \_ in range(n)] results = []

solve(board, 0, results) return results

def solve(board, col, results): if col >= len(board):

results.append(["".join(row) for row in board]) return

for i in range(len(board)): if is\_safe(board, i, col):

board[i][col] = 'Q' solve(board, col + 1, results) board[i][col] = '.'

def is\_safe(board, row, col):

# Check for another queen in the same row to the left for i in range(col):

if board[row][i] == 'Q': return False

# Check the upper diagonal to the left

for i, j in zip(range(row, -1, -1), range(col, -1, -1)): if board[i][j] == 'Q':

return False

# Check the lower diagonal to the left

for i, j in zip(range(row, len(board)), range(col, -1, -1)): if board[i][j] == 'Q':

return False return True

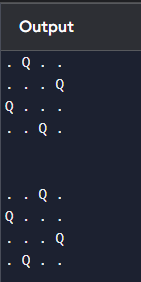
# Example usage:

n = 4

solutions = solve\_n\_queens(n) for sol in solutions:

print\_board(sol)

## OUTPUT:



1. **Discuss the generalization of the N-Queens Problem to other board sizes and shapes, such as rectangular boards or boards with obstacles. Explain how the algorithm can be adapted to handle these variations and the additional constraints they introduce. Provide examples of solving generalized N-Queens Problems for different board configurations, such as an 8×10 board, a 5×5 board with obstacles, and a 6×6 board with restricted positions.**
   1. **8×10 Board:**

**8 rows and 10 columns**

**Output: Possible solution [1, 3, 5, 7, 9, 2, 4, 6]**

## CODE:

def is\_safe(board, row, col, n\_rows, n\_cols): for i in range(row):

if board[i] == col or abs(board[i] - col) == abs(i - row): return False

return True

def solve\_n\_queens(board, row, n\_rows, n\_cols): if row == n\_rows: # All queens are placed

return True

for col in range(n\_cols):

if is\_safe(board, row, col, n\_rows, n\_cols): board[row] = col

if solve\_n\_queens(board, row + 1, n\_rows, n\_cols): return True

board[row] = -1 return False

def n\_queens\_rectangular(n\_rows, n\_cols): board = [-1] \* n\_rows # Initialize board

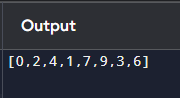
if solve\_n\_queens(board, 0, n\_rows, n\_cols): return board

else:

return "No solution"

result = n\_queens\_rectangular(8, 10) print(result)

## OUTPUT:



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:**

**Input: 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"]]**

**Output:**

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

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

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

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

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

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

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

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

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

## CODE:

def is\_valid(board, row, col, num):

# Check if 'num' is not in the current row, column, and 3x3 sub-box for i in range(9):

if board[row][i] == num or board[i][col] == num or board[3 \* (row // 3) + i // 3][3 \* (col // 3) + i

% 3] == num:

return False return True

def solve\_sudoku(board): for row in range(9):

for col in range(9):

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

for num in map(str, range(1, 10)): if is\_valid(board, row, col, num):

board[row][col] = num

if solve\_sudoku(board): return True

board[row][col] = '.' return False

return True 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"],

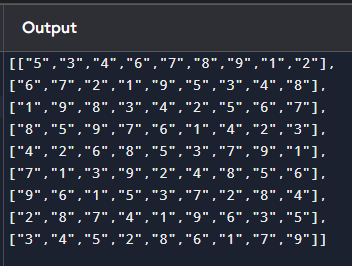
[".",".",".",".","8",".",".","7","9"]

]

solve\_sudoku(board) for row in board:

print(row)

## OUTPUT:



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:**

**Input: 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"]]**

**Output:**

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

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

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

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

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

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

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

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

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

## CODE:

def is\_valid(board, row, col, num): for i in range(9):

if board[row][i] == num or board[i][col] == num or board[3 \* (row // 3) + i // 3][3 \* (col // 3) + i

% 3] == num:

return False return True

def solve\_sudoku(board): for row in range(9):

for col in range(9):

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

for num in map(str, range(1, 10)): if is\_valid(board, row, col, num):

board[row][col] = num if solve\_sudoku(board):

return True board[row][col] = '.'

return False return True

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"],

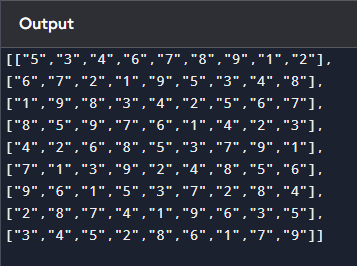
[".",".",".",".","8",".",".","7","9"]

]

solve\_sudoku(board) for row in board:

print(row)

## OUTPUT:



1. **You are given an integer array nums and an integer target. You want to build an expression out of nums by adding one of the symbols '+' and '-' before each integer in nums and then concatenate all the integers.For example, if nums = [2, 1], you can add a '+' before 2 and a '-' before 1 and concatenate them to build the expression "+2-1" Return the number of different expressions that you can build, which evaluates to target.**

**Example 1:**

**Input: nums = [1,1,1,1,1], target = 3 Output: 5**

## CODE:

def find\_target\_sum\_ways(nums, target):

memo = {} # To store already computed states def backtrack(index, current\_sum):

if index == len(nums):

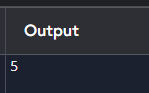
return 1 if current\_sum == target else 0 if (index, current\_sum) in memo:

return memo[(index, current\_sum)]

add = backtrack(index + 1, current\_sum + nums[index]) subtract = backtrack(index + 1, current\_sum - nums[index]) memo[(index, current\_sum)] = add + subtract

return memo[(index, current\_sum)] return backtrack(0, 0)

## OUTPUT:



1. **Given an array of integers arr, find the sum of min(b), where b ranges over every (contiguous) subarray of arr. Since the answer may be large, return the answer modulo 109 + Example 1:**

**Input: arr = [3,1,2,4] Output: 17 Explanation:**

**Subarrays are [3], [1], [2], [4], [3,1], [1,2], [2,4], [3,1,2], [1,2,4], [3,1,2,4].**

**Minimums are 3, 1, 2, 4, 1, 1, 2, 1, 1, 1.**

**Sum is 17.**

## CODE:

def sum\_subarray\_mins(arr):

MOD = 10\*\*9 + 7

n = len(arr) prev\_smaller = [-1] \* n next\_smaller = [n] \* n stack = []

for i in range(n):

while stack and arr[stack[-1]] >= arr[i]: stack.pop()

if stack:

prev\_smaller[i] = stack[-1] stack.append(i)

stack = []

for i in range(n):

while stack and arr[stack[-1]] > arr[i]: index = stack.pop() next\_smaller[index] = i

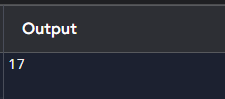
stack.append(i) result = 0

for i in range(n):

left = i - prev\_smaller[i] right = next\_smaller[i] - i

result = (result + arr[i] \* left \* right) % MOD return result

## OUTPUT:



1. **Given an array of distinct integers candidates and a target integer target, return a list of all unique combinations of candidates where the chosen numbers sum to target. You may return the combinations in any order.The same number may be chosen from candidates an unlimited number of times. Two combinations are unique if the frequency of at least one of the chosen numbers is different.The test cases are generated such that the number of unique combinations that sum up to target is less than 150 combinations for the given input. Example 1:**

**Input: candidates = [2,3,6,7], target = 7 Output: [[2,2,3],[7]]**

## CODE:

def combinationSum(candidates, target): result = []

def backtrack(remaining, combination, start): if remaining == 0:

result.append(list(combination)) return

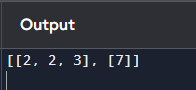
elif remaining < 0: return

for i in range(start, len(candidates)): combination.append(candidates[i])

backtrack(remaining - candidates[i], combination, i) # Can reuse the same element combination.pop()

backtrack(target, [], 0) return result

## OUTPUT:



1. **Given a collection of candidate numbers (candidates) and a target number (target), find all unique combinations in candidates where the candidate numbers sum to target. Each number in candidates may only be used once in the combination. The solution set must not contain duplicate combinations.**

**Example 1:**

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

**[ [1,1,6],**

**[1,2,5],**

**[1,7],**

**[2,6]**

## CODE:

def combinationSum2(candidates, target): result = []

candidates.sort() # Sort to handle duplicates def backtrack(remaining, combination, start):

if remaining == 0: result.append(list(combination)) return

elif remaining < 0: return

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

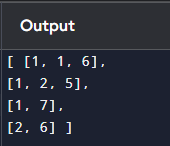
if i > start and candidates[i] == candidates[i - 1]: continue

combination.append(candidates[i]

backtrack(remaining - candidates[i], combination, i + 1 combination.pop()

backtrack(target, [], 0) return result

## OUTPUT:



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]]**

## CODE:

def permute(nums): def backtrack(start):

if start == len(nums):

# All numbers are used, add the current permutation to the result result.append(nums[:])

return

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

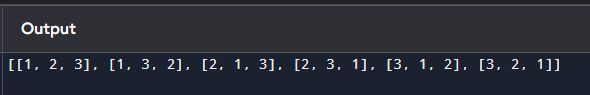
nums[start], nums[i] = nums[i], nums[start] backtrack(start + 1)

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

backtrack(0) # Start with the first index return result

nums = [1, 2, 3] print(permute(nums))

## OUTPUT:



1. **Given a collection of numbers, nums, that might contain duplicates, return all possible unique**

**permutations in any order. Example 1:**

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

**[[1,1,2],**

**[1,2,1],**

**[2,1,1]]**

## CODE:

def permuteUnique(nums): def backtrack(start):

if start == len(nums) result.append(nums[:]) return

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

if i > start and nums[i] == nums[i - 1]: continue

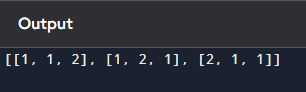
nums[start], nums[i] = nums[i], nums[start] # Swap backtrack(start + 1) # Recurse

nums[start], nums[i] = nums[i], nums[start] nums.sort()

result = [] backtrack(0) return result

nums = [1, 1, 2] print(permuteUnique(nums))

## OUTPUT:



**DAY 11**

**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**

**CODE:**

def graph\_coloring(edges, n):

adj = {i: [] for i in range(n)}

for u, v in edges:

adj[u].append(v)

adj[v].append(u)

def is\_safe(v, color, c):

for neighbor in adj[v]:

if color[neighbor] == c:

return False

return True

def color\_graph(v, color, m):

if v == n:

return True

for c in range(1, m + 1):

if is\_safe(v, color, c):

color[v] = c

if color\_graph(v + 1, color, m):

return True

color[v] = 0

return False

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

color = [0] \* n

if color\_graph(0, color, m):

return m

return n

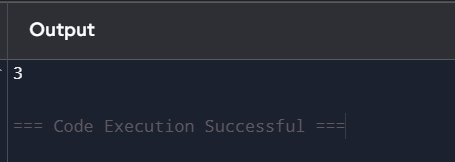
# Test case

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

n = 4

print(graph\_coloring(edges, n))

**OUTPUT:**

****

**2. You and your friends are tasked with coloring a map using a limited set of colors, with the following rules: At each step, you can choose any region of the map that hasn't been colored yet and color it with any available color. Your friend Alice will then color the next region using the same strategy, followed by your friend Bob. You aim to maximize the number of regions you color. Given a map represented as a list of regions and their adjacency relationships, write a function to determine the maximum number of regions you can color. 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, k = 3**

**CODE:**

def graph\_coloring(n, edges):

# Initialize adjacency list

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

for u, v in edges:

adjacency\_list[u].append(v)

adjacency\_list[v].append(u)

colors = [-1] \* n

for vertex in range(n):

unavailable\_colors = {colors[neighbor] for neighbor in adjacency\_list[vertex] if colors[neighbor] != -1}

for color in range(n):

if color not in unavailable\_colors:

colors[vertex] = color

break

return colors

def maximize\_regions\_colored(n, edges):

colors = graph\_coloring(n, edges)

color\_count = [0] \* n

for color in colors:

if color != -1:

color\_count[color] += 1

return min(count for count in color\_count if count > 0)

n = 4

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

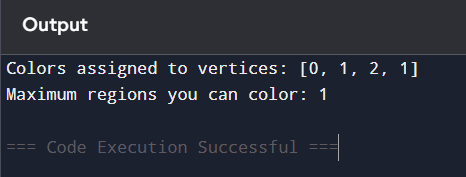
colors = graph\_coloring(n, edges)

print("Colors assigned to vertices:", colors)

max\_regions = maximize\_regions\_colored(n, edges)

print("Maximum regions you can color:", max\_regions)

**OUTPUT:**

****

**3. 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**

**CODE:**

def is\_hamiltonian\_cycle(edges, n):

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

for u, v in edges:

adjacency\_list[u].append(v)

adjacency\_list[v].append(u)

def backtrack(path):

# If the path contains all vertices, check if it forms a cycle

if len(path) == n:

return path[0] in adjacency\_list[path[-1]]

current = path[-1]

for neighbor in adjacency\_list[current]:

if neighbor not in path:

path.append(neighbor)

if backtrack(path):

return True

path.pop()

return False

for start in range(n):

if backtrack([start]):

return True

return False

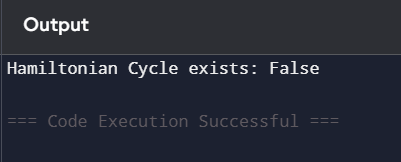
edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2), (2, 4), (4, 0)]

n = 5

result = is\_hamiltonian\_cycle(edges, n)

print("Hamiltonian Cycle exists:", result)

**OUTPUT:**

****

**4. 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:edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)] and n = 4**

**CODE:**

def is\_hamiltonian\_cycle(edges, n):

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

for u, v in edges:

adjacency\_list[u].append(v)

adjacency\_list[v].append(u)

def backtrack(path):

if len(path) == n:

return path[0] in adjacency\_list[path[-1]]

current = path[-1]

for neighbor in adjacency\_list[current]:

if neighbor not in path:

path.append(neighbor)

if backtrack(path):

return True

path.pop()

return False

for start in range(n):

if backtrack([start]):

return True

return False

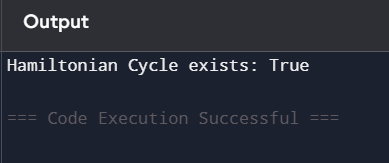
edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2), (2, 4), (4, 0)]

n = 5

result = is\_hamiltonian\_cycle(edges, n)

print("Hamiltonian Cycle exists:", result)

**OUTPUT:**

****

**5. You are tasked with designing an efficient coading to generate all subsets of a given set S containing n elements. Each subset should be outputted in lexicographical order. Return a list of lists where each inner list is a subset of the given set. Additionally, find out how your coading handles duplicate elements in S. A = [1, 2, 3] The subsets of [1, 2, 3] are: [], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]**

**CODE:**

def subsets(S):

S.sort()

result = []

def backtrack(start, path):

result.append(path[:])

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

if i > start and S[i] == S[i - 1]:

continue

path.append(S[i])

backtrack(i + 1, path)

path.pop()

backtrack(0, [])

return result

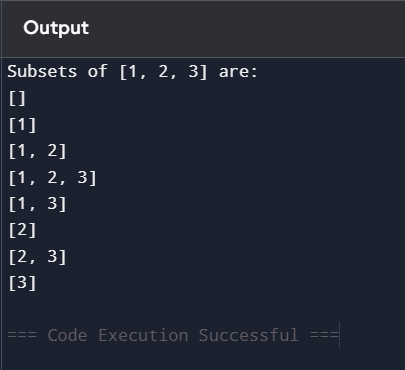
A = [1, 2, 3]

print("Subsets of", A, "are:")

for subset in subsets(A):

print(subset)

**OUTPUT:**

****

**6. Write a program to implement the concept of subset generation. Given a set of unique integers and a specific integer 3, generate all subsets that contain the element 3. Return a list of lists where each inner list is a subset containing the element 3 E = [2, 3, 4, 5], x = 3, The subsets containing 3 : [3], [2, 3], [3, 4], [3,5], [2, 3, 4], [2, 3, 5], [3, 4, 5], [2, 3, 4, 5] Given an integer array nums of unique elements, return all possible subsets(the power set). The solution set must not contain duplicate subsets. Return the solution in any order. Example 1: Input: nums = [1,2,3] Output: [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]] Example 2: Input: nums = [0] Output: [[],[0]]**

**CODE:**

def subsets\_with\_element(nums, x):

nums.sort()

result = []

def backtrack(start, path):

if x in path:

result.append(path[:])

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

path.append(nums[i])

backtrack(i + 1, path)

path.pop()

backtrack(0, [])

return result

def power\_set(nums):

nums.sort()

result = []

def backtrack(start, path):

result.append(path[:])

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

path.append(nums[i])

backtrack(i + 1, path)

path.pop()

backtrack(0, [])

return result

E = [2, 3, 4, 5]

x = 3

subsets\_with\_x = subsets\_with\_element(E, x)

print(f"Subsets of {E} containing {x}: {subsets\_with\_x}")

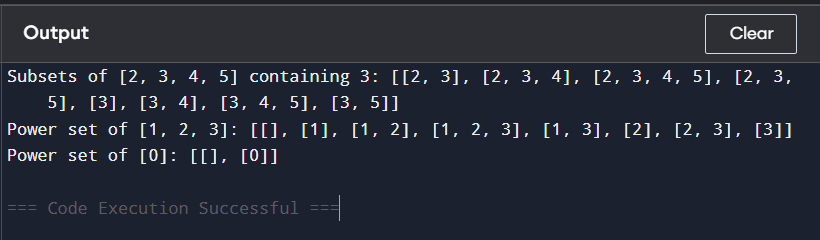
nums1 = [1, 2, 3]

nums2 = [0]

print(f"Power set of {nums1}: {power\_set(nums1)}")

print(f"Power set of {nums2}: {power\_set(nums2)}")

**OUTPUT:**

****

**7. You are given two string arrays words1 and words2. A string b is a subset of string a if every letter in b occurs in a including multiplicity. For example, "wrr" is a subset of "warrior" but is not a subset of "world". A string a from words1 is universal if for every string b in words2, b is a subset of a. Return an array of all the universal strings in words1. You may return the answer in any order. Example 1: Input: words1 = ["amazon","apple","facebook","google","leetcode"], words2 = ["e","o"] Output: ["facebook","google","leetcode"] Example 2: Input: words1 = ["amazon","apple","facebook","google","leetcode"], words2 = ["l","e"] Output: ["apple","google","leetcode"]**

**CODE:**

from collections import Counter

def words\_subset(words1, words2):

required = Counter()

for word in words2:

required.update(word)

result = []

for word in words1:

word\_count = Counter(word)

if all(word\_count[char] >= required[char] for char in required):

result.append(word)

return result

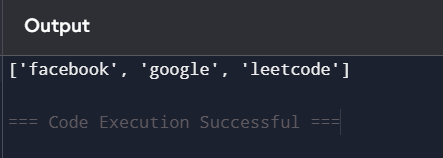
# Test case

words1 = ["amazon", "apple", "facebook", "google", "leetcode"]

words2 = ["e", "o"]

print(words\_subset(words1, words2))

**OUTPUT:**

****