41. **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: o Generate all possible permutations of the cities (excluding the starting city) using itertools.permutations. o 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. o Return the minimum distance and the shortest path (including the starting city at the beginning and end).**

```
import itertools
import math


# Function to calculate Euclidean distance between two cities
def distance(city1, city2):
    # city1 and city2 are tuples representing coordinates (x, y)
    return math.sqrt((city1[0] - city2[0]) ** 2 + (city1[1] - city2[1]) ** 2)


# TSP function to find the shortest path using exhaustive search
def tsp(cities):
    # Assume the first city is the starting city
    start_city = cities[0]
    cities_to_visit = cities[1:]  # Exclude the start city from permutations

    min_distance = float('inf')  # Initialize min distance to a large value
    best_path = []  # This will store the best path

    # Generate all permutations of the remaining cities
    for perm in itertools.permutations(cities_to_visit):
        # Create a full route by including the start city at the beginning and end
        route = [start_city] + list(perm) + [start_city]

        # Calculate the total distance for this route
        total_distance = 0
        for i in range(len(route) - 1):
            total_distance += distance(route[i], route[i+1])

        # Check if this route is shorter than the previously found routes
        if total_distance < min_distance:
            min_distance = total_distance
            best_path = route

    # Return the shortest distance and the corresponding path
    return min_distance, best_path

# Test case
cities = [(0, 0), (1, 2), (3, 4), (5, 6)]  # List of cities (x, y) coordinates
min_distance, best_path = tsp(cities)

print("Minimum Distance:", min_distance)
print("Best Path:", best_path)
```

42. **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**

```
def is_hamiltonian_cycle(n, edges):
    # Step 1: Build the adjacency matrix or adjacency list from the edge list
    adj_list = {i: [] for i in range(n)}
    for u, v in edges:
        adj_list[u].append(v)
        adj_list[v].append(u)

    # Step 2: Create a list to keep track of visited vertices
    visited = [False] * n

    # Step 3: Function to check if there is a Hamiltonian cycle using backtracking
    def backtrack(curr_vertex, count, start_vertex):
        # If we've visited all vertices, check if we can return to the starting vertex
        if count == n:
            # If there is an edge back to the starting vertex, we have a cycle
            if start_vertex in adj_list[curr_vertex]:
                return True
            return False

        # Try to visit every vertex from the current vertex
        for next_vertex in adj_list[curr_vertex]:
            if not visited[next_vertex]:
                visited[next_vertex] = True
                if backtrack(next_vertex, count + 1, start_vertex):
                    return True
                visited[next_vertex] = False

        return False

    # Step 4: Start the backtracking from vertex 0
    visited[0] = True
    return backtrack(0, 1, 0)

# Example Input:
edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)]
n = 4

# Call the function and print the result
print(is_hamiltonian_cycle(n, edges))  # Output: True or False
```

43. **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.

```
#include <stdio.h>
#include <limits.h>

#define N 5  // Number of cities (A, B, C, D, E)
#define INF INT_MAX

// Distance matrix for the cities (symmetric)
int dist[N][N] = {
    {0, 10, 15, 20, 25},  // A
    {10, 0, 35, 25, 30},  // B
    {15, 35, 0, 30, 20},  // C
    {20, 25, 30, 0, 15},  // D
    {25, 30, 20, 15, 0}   // E
};

// DP table to store the minimum distance for subsets of cities
int dp[1 << N][N];

// Function to solve the TSP using Dynamic Programming with Bitmasking
int tsp(int mask, int pos) {
    // If all cities have been visited, return the distance to the starting city (A)
    if (mask == (1 << N) - 1) {
        return dist[pos][0];
    }

    // If the result is already calculated, return it
    if (dp[mask][pos] != -1) {
        return dp[mask][pos];
    }
    int ans = INF;
    // Try to go to all unvisited cities
    for (int city = 0; city < N; city++) {
        if ((mask & (1 << city)) == 0) {
            int newAns = dist[pos][city] + tsp(mask | (1 << city), city);
            ans = (ans < newAns) ? ans : newAns;
        }
    }

    // Store the result and return it
    dp[mask][pos] = ans;
    return ans;
}

int main() {
    // Initialize the DP table with -1 (unvisited)
    for (int i = 0; i < (1 << N); i++) {
        for (int j = 0; j < N; j++) {
            dp[i][j] = -1;
        }
    }

    // Solve the TSP starting from city A (index 0)
    int min_cost = tsp(1, 0);  // Mask 1 means only city A is visited, starting at city A

    printf("The shortest route and its total distance: %d\n", min_cost);

    return 0;
}
```

44. **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 a) Test cases Input: keys[] = {10, 12}, freq[] = {34, 50} Output = 118 b) Input: keys[] = {10, 12, 20}, freq[] = {34, 8, 50} Output = 142**

```
import sys


def optimal_bst(keys, freq, n):

    # cost[i][j] will store the minimum cost of the binary search tree for keys[i..j]

    cost = [[0 for x in range(n+1)] for y in range(n+1)]

    # root[i][j] will store the index of the root of the optimal binary search tree for keys[i..j]

    root = [[0 for x in range(n+1)] for y in range(n+1)]


    # Initialize the cost for a single key (i == j)

    for i in range(n):

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

        root[i][i] = i


    # Calculate cost[i][j] for all lengths greater than 1

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

        for i in range(n-length+1):

            j = i + length - 1

            cost[i][j] = sys.maxsize  # Initialize with a large number

            # Try all roots in the range [i..j]

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

                # Calculate the cost of having 'r' as the root

                left_cost = cost[i][r-1] if r > i else 0

                right_cost = cost[r+1][j] if r < j else 0

                total_cost = left_cost + right_cost + sum(freq[i:j+1])

                # If the total cost is less, update cost and root

                if total_cost < cost[i][j]:

                    cost[i][j] = total_cost
                    root[i][j] = r

    # Print the cost and root matrix
    print("\nCost Table:")
    for row in cost:
        print(row)

    print("\nRoot Table:")
    for row in root:
        print(row)

    return cost[0][n-1]  # The cost of the optimal binary search tree

# Input for test case
keys = [10, 12, 16, 21]
freq = [4, 2, 6, 3]
n = len(keys)

# Call the function and display the output
optimal_cost = optimal_bst(keys, freq, n)
print(f"\nOptimal BST Cost: {optimal_cost}")
```

45. **Given an array arr of positive integers sorted in a strictly increasing order, and an integer k. return the kth positive integer that is missing from this array. Example 1: Input: arr = [2,3,4,7,11], k = 5 Output: 9 Explanation: The missing positive integers are [1,5,6,8,9,10,12,13,...]. The 5th missing positive integer is 9. Example 2: Input: arr = [1,2,3,4], k = 2 Output: 6 Explanation: The missing positive integers are [5,6,7,...]. The 2nd missing positive integer is 6.**

```
def findKthPositive(arr, k):
    # Expected number starts at 1
    missing_count = 0
    current = 1
    idx = 0
    n = len(arr)
    while missing_count < k:

        # If current number is not in arr, it's a missing number

        if idx < n and arr[idx] == current:

            idx += 1

        else:

            missing_count += 1

            if missing_count == k:

                return current

        current += 1


    return current  # This line will be never hit because the loop exits when the kth missing is found


# Test cases

print(findKthPositive([2, 3, 4, 7, 11], 5))  # Output: 9

print(findKthPositive([1, 2, 3, 4], 2))      # Output: 6
```

46. **. 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 62 66)**

```
import numpy as np


def strassen_matrix_multiplication(A, B):

    # Base case for 2x2 matrices

    if len(A) == 2 and len(A[0]) == 2:

        a, b = A[0][0], A[0][1]

        c, d = A[1][0], A[1][1]

        e, f = B[0][0], B[0][1]

        g, h = B[1][0], B[1][1]


        # Calculate intermediate products

        P1 = a * (f - h)
        P2 = (a + b) * h
        P3 = (c + d) * e
        P4 = d * (g - e)
        P5 = (a + d) * (e + h)
        P6 = (b - d) * (g + h)
        P7 = (a - c) * (e + f)

        # Calculate final sub-matrices
        C11 = P5 + P4 - P2 + P6
        C12 = P1 + P2
        C21 = P3 + P4
        C22 = P1 + P5 - P3 - P7

        # Combine the results into a single matrix
        C = np.array([[C11, C12], [C21, C22]])
        return C

def main():
    A = np.array([[1, 7], [3, 5]])
    B = np.array([[6, 8], [4, 2]])

    # Multiply matrices using Strassen's algorithm
    C = strassen_matrix_multiplication(A, B)

    print("Product Matrix C:")
    print(C)

# Execute the main function
if __name__ == "__main__":
    main()
```

47. **In a string S of lowercase letters, these letters form consecutive groups of the same character. For example, a string like s = "abbxxxxzyy" has the groups "a", "bb", "xxxx", "z", and "yy". A group is identified by an interval [start, end], where start and end denote the start and end indices (inclusive) of the group. In the above example, "xxxx" has the interval [3,6]. A group is considered large if it has 3 or more characters. Return the intervals of every large group sorted in increasing order by start index. Example 1: Input: s = "abbxxxxzzy" Output: [[3,6]] Explanation: "xxxx" is the only large group with start index 3 and end index 6. Example 2: Input: s = "abc" Output: [] Explanation: We have groups "a", "b", and "c", none of which are large groups.**

```
def largeGroupPositions(s):
    result = []
    start = 0
    n = len(s)

    # Iterate through the string
    for i in range(1, n + 1):
        # If we're at the end of the string or the current character is different from the previous
        if i == n or s[i] != s[i - 1]:
            # Check if the group length is at least 3
            if i - start >= 3:
                result.append([start, i - 1])
            # Start a new group
            start = i

    return result


# Test Cases
print(largeGroupPositions("abbxxxxzzy"))  # Output: [[3, 6]]
print(largeGroupPositions("abc"))         # Output: []
```

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

```
def find_max(nums):
    # Initialize max_val to the smallest possible value
    max_val = float('-inf')
    # Traverse the array to find the maximum value
    for num in nums:
        if num > max_val:
            max_val = num

    return max_val

# Test Cases
print(find_max([1, 2, 3, 4, 5]))  # Expected Output: 5
print(find_max([7, 7, 7, 7, 7]))  # Expected Output: 7
print(find_max([-10, 2, 3, -4, 5]))  # Expected Output: 5
```

48. **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.**

```
import itertools

# Function to calculate the total cost of a given assignment
def total_cost(assignment, cost_matrix):
    cost = 0
    for worker, task in assignment:
        cost += cost_matrix[worker][task]
    return cost

# Function to solve the assignment problem using exhaustive search
def assignment_problem(cost_matrix):
    num_workers = len(cost_matrix)
    num_tasks = len(cost_matrix[0])

    # Generate all permutations of workers to tasks (assign workers to tasks)
    worker_indices = list(range(num_workers))
    task_indices = list(range(num_tasks))

    # Generate all possible assignments (permutations of worker-task pairings)
    min_cost = float('inf')
    best_assignment = []

    for perm in itertools.permutations(task_indices):
        # Create an assignment where each worker is assigned to the task in the perm
        assignment = [(worker, perm[worker]) for worker in worker_indices]

        # Calculate the total cost of this assignment
        current_cost = total_cost(assignment, cost_matrix)

        # Check if this assignment has a lower cost than the previous best
        if current_cost < min_cost:
            min_cost = current_cost
            best_assignment = assignment

    # Return the optimal assignment and the corresponding cost
    return best_assignment, min_cost

# Test cases
cost_matrix_1 = [
    [3, 10, 7],
    [8, 5, 12],
    [4, 6, 9]
]

cost_matrix_2 = [
    [15, 9, 4],
    [8, 7, 18],
    [6, 12, 11]
]

# Solve the assignment problem for both test cases
best_assignment_1, min_cost_1 = assignment_problem(cost_matrix_1)
best_assignment_2, min_cost_2 = assignment_problem(cost_matrix_2)

# Print the results
print("Test Case 1:")
print("Optimal Assignment:", [(f"worker {worker+1}", f"task {task+1}") for worker, task in best_assignment_1])
print("Total Cost:", min_cost_1)

print("\nTest Case 2:")
print("Optimal Assignment:", [(f"worker {worker+1}", f"task {task+1}") for worker, task in best_assignment_2])
print("Total Cost:", min_cost_2)
```

49. **You are given an integer array nums and an integer target. You want to build an expression out of nums by adding one of the symbols '+' and '-' before each integer in nums and then concatenate all the integers.For example, if nums = [2, 1], you can add a '+' before 2 and a '-' before 1 and concatenate them to build the expression "+2-1" Return the number of different expressions that you can build, which evaluates to target. Example 1: Input: nums = [1,1,1,1,1], target = 3 Output: 5 Explanation: There are 5 ways to assign symbols to make the sum of nums be target 3. -1 + 1 + 1 + 1 + 1 = 3 +1 - 1 + 1 + 1 + 1 = 3 +1 + 1 - 1 + 1 + 1 = 3 +1 + 1 + 1 - 1 + 1 = 3 +1 + 1 + 1 + 1 - 1 = 3**

```
def findTargetSumWays(nums, target):
    total_sum = sum(nums)

    # If the target is impossible, return 0
    if (total_sum + target) % 2 != 0 or total_sum < target:
        return 0

    S_pos = (total_sum + target) // 2

    # DP array to store the number of ways to achieve each sum up to S_pos
    dp = [0] * (S_pos + 1)
    dp[0] = 1  # There's one way to make sum 0 (by taking no elements)

    # Iterate through each number in nums
    for num in nums:
        # Traverse backwards to prevent overwriting results for the same iteration
        for i in range(S_pos, num - 1, -1):
            dp[i] += dp[i - num]

    # The answer is the number of ways to achieve sum S_pos
    return dp[S_pos]

# Test case
nums = [1, 1, 1, 1, 1]
target = 3
print(findTargetSumWays(nums, target))  # Output: 5
```

50. **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]**

```
import heapq

def dijkstra(n, graph, source):
    # Step 1: Initialize the distances and the visited set
    INF = float('inf')  # Use infinity for no path
    dist = [INF] * n  # Distance array
    dist[source] = 0  # Distance from source to itself is 0
    visited = [False] * n  # Visited array

    # Priority queue: stores pairs of (distance, vertex)
    pq = [(0, source)]  # Start with the source vertex
    while pq:
        # Step 2: Get the vertex with the minimum distance from the priority queue
        current_dist, u = heapq.heappop(pq)

        # Skip if already visited
        if visited[u]:
            continue

        # Mark the vertex as visited
        visited[u] = True

        # Step 3: Update the distances for neighbors of the current vertex
        for v in range(n):
            if not visited[v] and graph[u][v] != INF:
                new_dist = current_dist + graph[u][v]
                if new_dist < dist[v]:
                    dist[v] = new_dist
                    heapq.heappush(pq, (new_dist, v))

    # Step 4: Return the distance array
    return dist

# Test case
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


# Run Dijkstra's algorithm

result = dijkstra(n, graph, source)

print(result)  # Output: [0, 7, 3, 9, 5]
```

51. **Given a string s, return the longest palindromic substring in S. Example 1: Input: s = "babad" Output: "bab" Explanation: "aba" is also a valid answer. Example 2: Input: s = "cbbd" Output: "bb" Constraints: ● 1 <= s.length <= 1000 ● s consist of only digits and English letters.**

```
def longest_palindromic_substring(s: str) -> str:

    def expand_around_center(left: int, right: int) -> str:

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

            left -= 1

            right += 1

        # Return the palindromic substring

        return s[left + 1:right]


    longest = ""

    for i in range(len(s)):

        # Odd length palindrome

        substr1 = expand_around_center(i, i)

        # Even length palindrome

        substr2 = expand_around_center(i, i + 1)


        # Update longest if necessary

        if len(substr1) > len(longest):

            longest = substr1

        if len(substr2) > len(longest):

            longest = substr2


    return longest
```

```
# Example usage
print(longest_palindromic_substring("babad"))  # Output: "bab" or "aba"
print(longest_palindromic_substring("cbbd"))   # Output: "bb"
```

52. **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]]**

```
from typing import List


def subsets_with_element(nums: List[int], x: int) -> List[List[int]]:
    """
    Generate all subsets that must include the element x.
    """
    # Step 1: Generate subsets of nums excluding the element x
    def backtrack(start, path):
        # Add the current subset to the result (it will include x by default)
        result.append(path + [x])

        # Add all combinations starting from 'start' index.
        for i in range(start, len(nums)):
            backtrack(i + 1, path + [nums[i]])

    result = []
    # Remove element x from the list, since we want subsets containing x
    nums = [num for num in nums if num != x]

    # Step 2: Use backtracking to generate subsets from the remaining elements
    backtrack(0, [])

    return result
```

```
# Example 1: Input: nums = [2, 3, 4, 5], x = 3
nums = [2, 3, 4, 5]
x = 3
print("Subsets containing 3:", subsets_with_element(nums, x))


def generate_power_set(nums: List[int]) -> List[List[int]]:
    """
    Generate the power set of nums.
    """
    def backtrack(start, path):
        result.append(path)
        for i in range(start, len(nums)):
            backtrack(i + 1, path + [nums[i]])

    result = []
    backtrack(0, [])
    return result


# Example 2: Input: nums = [1, 2, 3]
nums2 = [1, 2, 3]
print("Power set:", generate_power_set(nums2))
```

53. **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. 2. 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.**

```
import itertools


# Function to calculate the total value of selected items
def total_value(items, values):
    return sum(values[i] for i in items)

# Function to check if the total weight of selected items is feasible (does not exceed capacity)
def is_feasible(items, weights, capacity):
    total_weight = sum(weights[i] for i in items)
    return total_weight <= capacity

# Function to solve the 0-1 Knapsack problem using exhaustive search
def knapsack_problem(weights, values, capacity):
    num_items = len(weights)

    # List to track the best solution
    best_value = 0
    best_selection = []

    # Generate all possible combinations of items (represented by their indices)
    for r in range(1, num_items + 1):  # Start from 1 to avoid the empty set
        for subset in itertools.combinations(range(num_items), r):
            # Check if the combination of items is feasible (weight <= capacity)
            if is_feasible(subset, weights, capacity):
                # Calculate the total value of this subset
                current_value = total_value(subset, values)

                # Track the best solution
                if current_value > best_value:
                    best_value = current_value
                    best_selection = subset

    return list(best_selection), best_value

# Test Case 1
```

```
weights_1 = [2, 3, 1]
values_1 = [4, 5, 3]
capacity_1 = 4

# Test Case 2
weights_2 = [1, 2, 3, 4]
values_2 = [2, 4, 6, 3]
capacity_2 = 6

# Solve the 0-1 Knapsack Problem for both test cases
best_selection_1, best_value_1 = knapsack_problem(weights_1, values_1, capacity_1)
best_selection_2, best_value_2 = knapsack_problem(weights_2, values_2, capacity_2)

# Print the results
print("Test Case 1:")
print("Optimal Selection:", best_selection_1)
print("Total Value:", best_value_1)

print("\nTest Case 2:")
print("Optimal Selection:", best_selection_2)
print("Total Value:", best_value_2)
```

54. **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 Test cases Input : N= 6, a[] = {10,20,30,40,50,60}, search key = 50 Output : 5 Input : N= 7, a[] = {21,32,40,54,65,76,87}, search key = 32 Output : 2**

```
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1
    comparisons = 0  # Initialize a variable to count comparisons
    while low <= high:
        mid = (low + high) // 2
        comparisons += 1  # Increment the comparison count

        # Check if target is present at mid
        if arr[mid] == target:
            return mid, comparisons
        # If target is smaller than mid, search the left half
        elif arr[mid] > target:
            high = mid - 1
        # If target is greater than mid, search the right half
        else:
            low = mid + 1

    return -1, comparisons  # Return -1 if the element is not found

# Test Case 1
arr1 = [5, 10, 15, 20, 25, 30, 35, 40, 45]
target1 = 20
index, comparisons1 = binary_search(arr1, target1)
print(f"Index of {target1}: {index}, Comparisons: {comparisons1}")

# Test Case 2
arr2 = [10, 20, 30, 40, 50, 60]
target2 = 50
index, comparisons2 = binary_search(arr2, target2)
print(f"Index of {target2}: {index}, Comparisons: {comparisons2}")

# Test Case 3
arr3 = [21, 32, 40, 54, 65, 76, 87]
target3 = 32
```

```
index, comparisons3 = binary_search(arr3, target3)

print(f"Index of {target3}: {index}, Comparisons: {comparisons3}")
```

55. **Given a string s and a dictionary of strings wordDict, return true if s can be segmented into a space-separated sequence of one or more dictionary words. Note that the same word in the dictionary may be reused multiple times in the segmentation. Example 1: Input: s = "leetcode", wordDict = ["leet","code"] Output: true Explanation: Return true because "leetcode" can be segmented as "leet code". Example 2: Input: s = "applepenapple", wordDict = ["apple","pen"] Output: true Explanation: Return true because "applepenapple" can be segmented as "apple pen apple". Note that you are allowed to reuse a dictionary word. Example 3: Input: s = "catsandog", wordDict = ["cats","dog","sand","and","cat"] Output: false**

```
def word_break(s: str, wordDict: list[str]) -> bool:

    word_set = set(wordDict)  # Convert list to set for faster lookup

    dp = [False] * (len(s) + 1)

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


    for i in range(1, len(s) + 1):

        for j in range(i):

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

                dp[i] = True

                break  # No need to check further once dp[i] is True


    return dp[len(s)]


# Example usage

print(word_break("leetcode", ["leet", "code"]))  # Output: True

print(word_break("applepenapple", ["apple", "pen"]))  # Output: True

print(word_break("catsandog", ["cats", "dog", "sand", "and", "cat"]))  # Output: False
```

56. **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. Example 2: Input: words = ["leetcode","et","code"] Output: ["et","code"] Explanation: "et", "code" are substring of "leetcode". Example 3: Input: words = ["blue","green","bu"] Output: [] Explanation: No string of words is substring of another string.**

```
def stringMatching(words):

    result = []
    # Iterate through each word in the list
    for i in range(len(words)):
        # Check if this word is a substring of any other word
        for j in range(len(words)):
            if i != j and words[i] in words[j]:
                result.append(words[i])
                break  # No need to check further if we already found a match
    return result


# Test cases
print(stringMatching(["mass", "as", "hero", "superhero"]))  # Output: ["as", "hero"]
print(stringMatching(["leetcode", "et", "code"]))  # Output: ["et", "code"]
print(stringMatching(["blue", "green", "bu"]))  # Output: []
```

57. **There is a robot on an m x n grid. The robot is initially located at the top-left corner (i.e., grid[0][0]). The robot tries to move to the bottom-right corner (i.e., grid[m - 1][n - 1]). The robot can only move either down or right at any point in time. Given the two integers m and n, return the number of possible unique paths that the robot can take to reach the bottom-right corner. The test cases are generated so that the answer will be less than or equal to 2 * 10 9. Example 1: START FINISH Input: m = 3, n = 7 Output: 28 Example 2: Input: m = 3, n = 2 Output: 3 Explanation: From the top-left corner, there are a total of 3 ways to reach the bottom-right corner: 1. Right -> Down -> Down 2. Down -> Down -> Right 3. Down -> Right -> Down**

```
def uniquePaths(m, n):
    # We need to calculate C(m+n-2, m-1) or C(m+n-2, n-1)

    # The number of paths is the combination of (m+n-2) choose (m-1) or (n-1)
    # We will use a more efficient method to compute the combination

    # We use the smaller of (m-1) or (n-1) for efficiency
    k = min(m - 1, n - 1)

    result = 1
    for i in range(k):
        result *= (m + n - 2 - i)  # Multiply by the top of the fraction
        result //= (i + 1)        # Divide by the bottom of the fraction


    return result


# Test cases

print(uniquePaths(3, 7))  # Output: 28

print(uniquePaths(3, 2))  # Output: 3
```

58. **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 Test cases Input : N= 7, a[] = {13,19,24,29,35,41,42}, search key = 42 Output : 7 Test cases Input : N= 6, a[] = {20,40,60,80,100,120}, search key = 60 Output : 3**

```
def binary_search(arr, target):

    low = 0

    high = len(arr) - 1

    comparisons = 0


    while low <= high:

        mid = (low + high) // 2

        comparisons += 1


        # Show midpoint calculation and comparisons

        print(f"Low: {low}, High: {high}, Mid: {mid}, Comparisons: {comparisons}")


        if arr[mid] == target:

            return mid, comparisons

        elif arr[mid] < target:

            low = mid + 1

        else:

            high = mid - 1
    return -1, comparisons  # Element not found


# Test the binary search function with the given array and target
arr = [3, 9, 14, 19, 25, 31, 42, 47, 53]
target = 31


# Perform Binary Search
index, comparisons = binary_search(arr, target)
if index != -1:
    print(f"Element {target} found at index {index}. Total comparisons: {comparisons}")
else:
    print(f"Element {target} not found. Total comparisons: {comparisons}")
```

59. **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 Test cases Input : N= 7, a[] = {13,19,24,29,35,41,42}, search key = 42 Output : 7 Test cases Input : N= 6, a[] = {20,40,60,80,100,120}, search key = 60 Output : 3**

```
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1
    comparisons = 0


    while low <= high:
        mid = (low + high) // 2
        comparisons += 1


        # Show midpoint calculation and comparisons
        print(f"Low: {low}, High: {high}, Mid: {mid}, Comparisons: {comparisons}")


        if arr[mid] == target:
            return mid, comparisons
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1

    return -1, comparisons  # Element not found

# Test the binary search function with the given array and target
arr = [3, 9, 14, 19, 25, 31, 42, 47, 53]
target = 31

# Perform Binary Search
index, comparisons = binary_search(arr, target)
if index != -1:
    print(f"Element {target} found at index {index}. Total comparisons: {comparisons}")
else:
    print(f"Element {target} not found. Total comparisons: {comparisons}")
```

60. **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**

```
def max_weight_in_container(weights, max_capacity):
    # Sort the weights in descending order to prioritize heavier items
    weights.sort(reverse=True)

    total_weight = 0  # This will store the total weight of items added to the container

    # Iterate over the sorted weights and add items until the container is full
    for weight in weights:
        if total_weight + weight <= max_capacity:
            total_weight += weight  # Add the current item
        else:
            break  # Stop if the next item doesn't fit

    return total_weight

# Test Case 1
n = 5
weights = [10, 20, 30, 40, 50]
max_capacity = 60

# Run the function
result = max_weight_in_container(weights, max_capacity)

# Output the result
print(result)
```