**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

**Program:-**

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

k = 4  # Find the 4th smallest element (0-indexed)

# Step 1: Base case for small arrays

if len(arr) < 10:

    result = sorted(arr)[k]

else:

    while True:

        # Step 2: Divide into groups of 5

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

        # Step 3: Find medians of each group

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

        # Step 4: Recursively find the median of medians

        if len(medians) <= 5:

            pivot = sorted(medians)[len(medians) // 2]

        else:

            # Using the median of medians logic

            arr\_copy = medians

            while True:

                if len(arr\_copy) < 10:

                    pivot = sorted(arr\_copy)[len(arr\_copy) // 2]

                    break

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

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

                arr\_copy = medians\_copy

        # Step 5: Partition the array

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

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

        pivot\_count = arr.count(pivot)

        # Step 6: Determine the k-th smallest element

        if k < len(low):

            arr = low  # Search in the lower partition

        elif k < len(low) + pivot\_count:

            result = pivot  # Pivot is the k-th smallest

            break

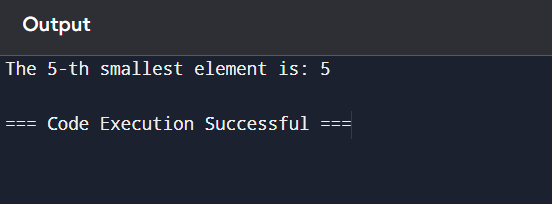
        else:

            arr = high  # Search in the upper partition

            k -= len(low) + pivot\_count  # Adjust k for the upper partition

print(f"The {k + 1}-th smallest element is: {result}")

**output:-**



2.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

arr = [23, 17, 31, 44, 55, 21, 20, 18, 19, 27] k = 5

Output: An integer representing the k-th smallest element in the array.

**Program:-**

def median\_of\_medians(arr, k):

    # Base case for small arrays

    if len(arr) < 10:

        return sorted(arr)[k]  # Directly sort and return k-th element

    # Step 1: Divide into groups of 5

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

    # Step 2: Find medians of each group

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

    # Step 3: Recursively find the median of medians

    pivot = median\_of\_medians(medians, len(medians) // 2)

    # Step 4: Partition the original array

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

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

    pivot\_count = arr.count(pivot)

    # Step 5: Determine the k-th smallest element

    if k < len(low):

        return median\_of\_medians(low, k)

    elif k < len(low) + pivot\_count:

        return pivot  # pivot is the k-th smallest

    else:

        return median\_of\_medians(high, k - len(low) - pivot\_count)

# Example usage

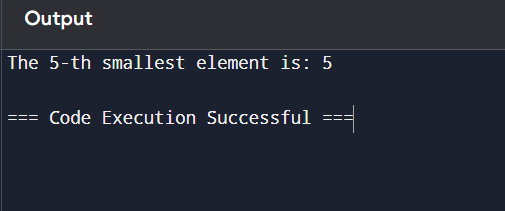
arr = [3, 1, 2, 5, 4, 6, 7, 9, 8]

k = 4  # Find the 4th smallest element (0-indexed)

result = median\_of\_medians(arr, k)

print(f"The {k + 1}-th smallest element is: {result}")

**output:-**



3.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.

a) Set[] = {45, 34, 4, 12, 5, 2}

b) Set[]= {1, 3, 2, 7, 4, 6}

Target Sum : 42 Target sum = 10:

**Program:-**

from itertools import combinations

arr = [2, 3, 5, 7, 11]

target = 12

mid = len(arr) // 2

left\_part = arr[:mid]

right\_part = arr[mid:]

left\_sums = set()

for i in range(len(left\_part) + 1):

    for combo in combinations(left\_part, i):

        left\_sums.add(sum(combo))

left\_sums = sorted(left\_sums)  # Sort the left subset sums

right\_sums = set()

for i in range(len(right\_part) + 1):

    for combo in combinations(right\_part, i):

        right\_sums.add(sum(combo))

right\_sums = sorted(right\_sums)  # Sort the right subset sums

# Step 4: Find the closest sum to the target

closest\_sum = float('inf')

closest\_diff = float('inf')

# Step 5: Use two pointers to find the closest sum

for left\_sum in left\_sums:

    low, high = 0, len(right\_sums) - 1

    while low <= high:

        mid = (low + high) // 2

        current\_sum = left\_sum + right\_sums[mid]

        diff = abs(current\_sum - target)

        if diff < closest\_diff:

            closest\_diff = diff

            closest\_sum = current\_sum

        # Move pointers based on comparison

        if current\_sum < target:

            low = mid + 1

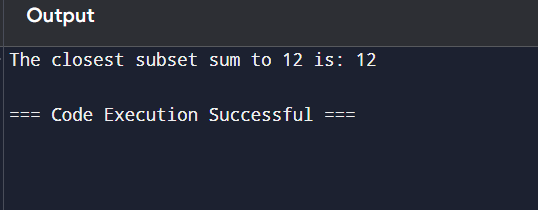
        else:

            high = mid - 1

# Output the closest subset sum

print(f"The closest subset sum to {target} is: {closest\_sum}")

**output:-**



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

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

b) E = {3, 34, 4, 12, 5, 2} exact Sum = 15

**program:-**

from itertools import combinations

# Input array and target sum

arr = [2, 3, 5, 7, 11, 13, 17, 19, 23]  # Example large array

E = 28  # Target sum

# Step 1: Split the array into two halves

mid = len(arr) // 2

left\_part = arr[:mid]

right\_part = arr[mid:]

# Step 2: Get all possible subset sums for the left half

left\_sums = set()

for i in range(len(left\_part) + 1):

    for combo in combinations(left\_part, i):

        left\_sums.add(sum(combo))

# Step 3: Get all possible subset sums for the right half

right\_sums = set()

for i in range(len(right\_part) + 1):

    for combo in combinations(right\_part, i):

        right\_sums.add(sum(combo))

# Step 4: Check if any combination of left and right sums equals E

found = False

for left\_sum in left\_sums:

    if (E - left\_sum) in right\_sums:

        found = True

        break

# Output the result

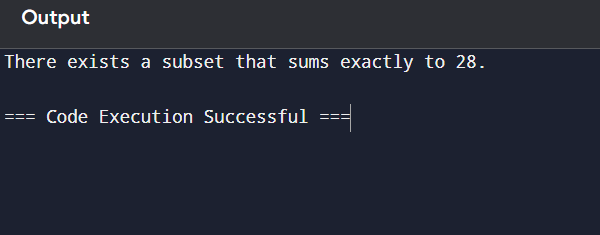
if found:

    print(f"There exists a subset that sums exactly to {E}.")

else:

    print(f"No subset sums exactly to {E}.")

**output:-**



5.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.

**Program:-**

import numpy as np

def strassen\_matrix\_multiplication(A, B):

    # Ensure the matrices are 2x2

    assert A.shape == (2, 2) and B.shape == (2, 2), "Both matrices must be 2x2."

    # Extracting elements of the matrices

    A11, A12 = A[0, 0], A[0, 1]

    A21, A22 = A[1, 0], A[1, 1]

    B11, B12 = B[0, 0], B[0, 1]

    B21, B22 = B[1, 0], B[1, 1]

    # Calculate the 7 products

    M1 = (A11 + A22) \* (B11 + B22)

    M2 = (A21 + A22) \* B11

    M3 = A11 \* (B12 - B22)

    M4 = A22 \* (B21 - B11)

    M5 = (A11 + A12) \* B22

    M6 = (A21 - A11) \* (B11 + B12)

    M7 = (A12 - A22) \* (B21 + B22)

    # Calculate the resulting matrix elements

    C11 = M1 + M4 - M5 + M7

    C12 = M3 + M5

    C21 = M2 + M4

    C22 = M1 - M2 + M3 + M6

    # Combine results into a 2x2 matrix

    C = np.array([[C11, C12],

                  [C21, C22]])

    return C

# Define matrices A and B

A = np.array([[1, 7],

              [3, 5]])

B = np.array([[1, 3],

              [7, 5]])

# Perform Strassen's multiplication

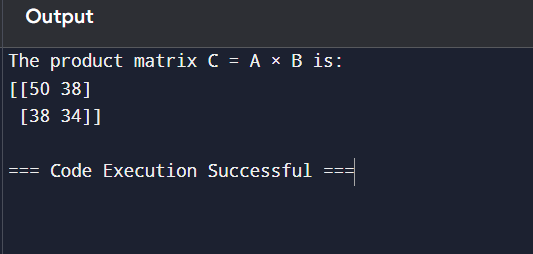
C = strassen\_matrix\_multiplication(A, B)

# Print the result

print("The product matrix C = A × B is:")

print(C)

**output:-**



6.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

**Program:-**

def karatsuba(x, y):

    # Base case for recursion

    if x < 10 or y < 10:

        return x \* y

    # Calculate the size of the numbers

    num1\_str = str(x)

    num2\_str = str(y)

    n = max(len(num1\_str), len(num2\_str))

    half\_n = n // 2

    # Split the digit sequences at the half point

    high1, low1 = x // 10\*\*half\_n, x % 10\*\*half\_n

    high2, low2 = y // 10\*\*half\_n, y % 10\*\*half\_n

    # 3 recursive calls

    z0 = karatsuba(low1, low2)        # Low parts

    z1 = karatsuba((low1 + high1), (low2 + high2))  # Cross terms

    z2 = karatsuba(high1, high2)      # High parts

    # Combine the results

    return (z2 \* 10\*\*(2 \* half\_n)) + ((z1 - z2 - z0) \* 10\*\*half\_n) + z0

# Test Case 1

x = 1234

y = 5678

z = karatsuba(x, y)

# Print the result

print(f"z = {x} × {y} = {z}")

**output:-**

