**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 arr = [12, 3, 5, 7, 4, 19, 26] k = 3 Expected Output:5 arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] k = 6 Expected Output:6**

**Code :**

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

pivot\_value = arr[pivot]

arr[pivot], arr[high] = arr[high], arr[pivot]

store\_index = low

for i in range(low, high):

if arr[i] < pivot\_value:

arr[store\_index], arr[i] = arr[i], arr[store\_index]

store\_index += 1

arr[store\_index], arr[high] = arr[high], arr[store\_index]

return store\_index

def select\_pivot(arr, low, high):

if high - low + 1 <= 5:

return sorted(arr[low:high+1])[(high-low)//2]

sublists = [arr[i:i+5] for i in range(low, high+1, 5)]

medians = [sorted(sublist)[len(sublist)//2] for sublist in sublists]

return select\_pivot(medians, 0, len(medians)-1)

def kth\_smallest(arr, low, high, k):

while low <= high:

pivot = select\_pivot(arr, low, high)

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

if pivot\_index == k:

return arr[pivot\_index]

elif pivot\_index < k:

low = pivot\_index + 1

else:

high = pivot\_index - 1

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

return kth\_smallest(arr, 0, len(arr)-1, k-1)

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

k = 2

print("The smallest element is {}".format(k, find\_kth\_smallest(arr, k)))

**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.**

**Code :**

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

pivot\_value = arr[pivot\_index]

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

store\_index = low

for i in range(low, high):

if arr[i] < pivot\_value:

arr[store\_index], arr[i] = arr[i], arr[store\_index]

store\_index += 1

arr[store\_index], arr[high] = arr[high], arr[store\_index]

return store\_index

def select\_pivot(arr, low, high):

if high - low + 1 <= 5:

return sorted(arr[low:high+1])[len(arr[low:high+1])//2]

medians = []

for i in range(low, high+1, 5):

sublist = sorted(arr[i:i+5])

medians.append(sublist[len(sublist)//2])

return select\_pivot(medians, 0, len(medians)-1)

def kth\_smallest(arr, low, high, k):

while low <= high:

pivot\_index = arr.index(select\_pivot(arr, low, high))

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

if pivot\_index == k:

return arr[pivot\_index]

elif pivot\_index < k:

low = pivot\_index + 1

else:

high = pivot\_index - 1

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

return kth\_smallest(arr, 0, len(arr)-1, k-1)

# Example usage

arr1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

k1 = 6

print("The smallest element is {}".format(k1, median\_of\_medians(arr1, k1)))

**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} Target Sum : 42 b) Set[]= {1, 3, 2, 7, 4, 6} Target sum = 10:**

**Code :**

from itertools import combinations

def get\_all\_sums(arr):

sums = []

for i in range(len(arr) + 1):

for comb in combinations(arr, i):

sums.append(sum(comb))

return sums

def meet\_in\_the\_middle(arr, target\_sum):

n = len(arr)

left\_half = arr[:n // 2]

right\_half = arr[n // 2:]

left\_sums = get\_all\_sums(left\_half)

right\_sums = get\_all\_sums(right\_half)

right\_sums.sort()

closest\_sum = float('inf')

closest\_pair = (0, 0)

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]

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

closest\_sum = current\_sum

closest\_pair = (left\_sum, right\_sums[mid])

if current\_sum < target\_sum:

low = mid + 1

else:

high = mid - 1

return closest\_sum

# Example usage

set1 = [45, 34, 4, 12, 5, 2]

target\_sum1 = 42

print("Closest sum to {} is {}".format(target\_sum1, meet\_in\_the\_middle(set1, target\_sum1)))

set2 = [1, 3, 2, 7, 4, 6]

target\_sum2 = 10

print("Closest sum to {} is {}".format(target\_sum2, meet\_in\_the\_middle(set2, target\_sum2)))

**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.**

**Code :**

from itertools import combinations

def get\_all\_sums(arr):

sums = []

for i in range(len(arr) + 1):

for comb in combinations(arr, i):

sums.append(sum(comb))

return sums

def meet\_in\_the\_middle(arr, exact\_sum):

n = len(arr)

left\_half = arr[:n // 2]

right\_half = arr[n // 2:]

left\_sums = get\_all\_sums(left\_half)

right\_sums = get\_all\_sums(right\_half)

right\_sums\_set = set(right\_sums)

for left\_sum in left\_sums:

if (exact\_sum - left\_sum) in right\_sums\_set:

return True

return False

arr1 = [1, 3, 9, 2, 7, 12]

exact\_sum1 = 15

print("Subset with exact sum {} exists: {}".format(exact\_sum1, meet\_in\_the\_middle(arr1, exact\_sum1)))

**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. 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).**

**Code :**

import numpy as np

def strassen\_multiply(A, B):

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

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

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

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)

c11 = p5 + p4 - p2 + p6

c12 = p1 + p2

c21 = p3 + p4

c22 = p1 + p5 - p3 - p7

return np.array([[c11, c12], [c21, c22]])

else:

raise ValueError("Strassen's algorithm is typically used for matrices larger than 2x2.")

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

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

C = strassen\_multiply(A, B)

print("Resultant matrix C:")

print(C)

**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**

**Code :**

def karatsuba(x, y):

if x < 10 or y < 10:

return x \* y

n = max(len(str(x)), len(str(y)))

n\_half = n // 2

high1, low1 = divmod(x, 10\*\*n\_half)

high2, low2 = divmod(y, 10\*\*n\_half)

z0 = karatsuba(low1, low2)

z1 = karatsuba((low1 + high1), (low2 + high2))

z2 = karatsuba(high1, high2)

return (z2 \* 10\*\*(2 \* n\_half)) + ((z1 - z2 - z0) \* 10\*\*n\_half) + z0

x = 1234

y = 5678

z = karatsuba(x, y)

print(f"The product of {x} and {y} is {z}")