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.**

**Code:**

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

def partition(arr, l, r, pivot\_idx):

pivot\_value = arr[pivot\_idx]

arr[pivot\_idx], arr[r] = arr[r], arr[pivot\_idx]

store\_idx = l

for i in range(l, r):

if arr[i] < pivot\_value:

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

store\_idx += 1

arr[store\_idx], arr[r] = arr[r], arr[store\_idx]

return store\_idx

def select(arr, l, r, k):

if l == r:

return arr[l]

pivot\_idx = l

while True:

pivot\_idx = partition(arr, l, r, pivot\_idx)

if pivot\_idx == k:

return arr[k]

elif k < pivot\_idx:

r = pivot\_idx - 1

else:

l = pivot\_idx + 1

n = len(arr)

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

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

if len(medians) <= 5:

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

else:

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

pivot\_idx = arr.index(pivot)

arr[pivot\_idx], arr[n - 1] = arr[n - 1], arr[pivot\_idx]

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

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

k = 3

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

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

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.**

**Code:**

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

def partition(arr, l, r, pivot\_idx):

pivot\_value = arr[pivot\_idx]

arr[pivot\_idx], arr[r] = arr[r], arr[pivot\_idx]

store\_idx = l

for i in range(l, r):

if arr[i] < pivot\_value:

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

store\_idx += 1

arr[r], arr[store\_idx] = arr[store\_idx], arr[r]

return store\_idx

def select(arr, l, r, k):

if l == r:

return arr[l]

pivot\_idx = l

while True:

pivot\_idx = partition(arr, l, r, pivot\_idx)

if pivot\_idx == k:

return arr[k]

elif k < pivot\_idx:

r = pivot\_idx - 1

else:

l = pivot\_idx + 1

n = len(arr)

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

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

if len(medians) <= 5:

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

else:

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

pivot\_idx = arr.index(pivot)

arr[pivot\_idx], arr[n - 1] = arr[n - 1], arr[pivot\_idx]

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

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.**

**Code:**

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

n = len(arr)

half = n // 2

left\_half = arr[:half]

right\_half = arr[half:]

left\_sums = {0}

right\_sums = {0}

for i in range(1, 1 << len(left\_half)):

left\_sums.add(sum(left\_half[j] for j in range(len(left\_half)) if i & (1 << j)))

for i in range(1, 1 << len(right\_half)):

right\_sums.add(sum(right\_half[j] for j in range(len(right\_half)) if i & (1 << j)))

right\_sums = sorted(right\_sums)

closest\_sum = float('inf')

closest\_subset = None

for left\_sum in left\_sums:

idx = bisect\_left(right\_sums, target - left\_sum)

if idx < len(right\_sums):

total = left\_sum + right\_sums[idx]

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

closest\_sum = total

closest\_subset = (left\_sum, right\_sums[idx])

return closest\_subset

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**

**Code:**

def is\_subset\_sum(arr, n, E):

half = n // 2

for i in range(1 << half):

s = 0

for j in range(half):

if i & (1 << j):

s += arr[j]

if s == E:

return True

for i in range(1 << (n - half)):

s = 0

for j in range(n - half):

if i & (1 << j):

s += arr[half + j]

if s == E:

return True

return False

arr = [3, 34, 4, 12, 5, 2]

E = 9

n = len(arr)

if is\_subset\_sum(arr, n, E):

print("Subset with sum E exists")

else:

print("No subset with sum E")

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**

**Code:**

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

if len(A) == 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)

C = [[p5 + p4 - p2 + p6, p1 + p2], [p3 + p4, p1 + p5 - p3 - p7]]

return C

else:

return "Input matrices are not 2x2"

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

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

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

print(C)

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

**Code:**

def karatsuba(x, y):

if x < 10 or y < 10:

return x \* y

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

m2 = m // 2

high1, low1 = divmod(x, 10\*\*m2)

high2, low2 = divmod(y, 10\*\*m2)

z0 = karatsuba(low1, low2)

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

z2 = karatsuba(high1, high2)

return (z2 \* 10\*\*(2\*m2)) + ((z1 - z2 - z0) \* 10\*\*m2) + z0

X = 1234

Y = 5678

Z = karatsuba(X, Y)

print(Z)