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() # Sort medians to find the overall median

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:

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

k = 2

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

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:

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

k = 2

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.

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:

The closest sum to the target 42 is: 42

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.

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:

True: A subset that sums exactly to 15 exists.

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

35 , 42)

CODE:

import numpy as np

def strassen\_multiply(A, B):

if len(A) == 2 and len(B) == 2:

# C11 = A11 \* B11 + A12 \* B21

# C12 = A11 \* B12 + A12 \* B22

# C21 = A21 \* B11 + A22 \* B21

# C22 = A21 \* B12 + A22 \* B22

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:

Product Matrix C:

[[34. 62.]

[38. 46.]]

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):

# 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:

7016652