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

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

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

pivot\_value = arr[pivot]

# Move pivot to the end

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

store\_index = low

for i in range(low, high):

if arr[i] < pivot\_value:

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

store\_index += 1

# Move pivot to its final place

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

return store\_index

def select(arr, left, right, k):

if left == right:

return arr[left]

pivot\_index = median\_of\_medians(arr, left, right)

pivot\_index = partition(arr, left, right, pivot\_index)

if k == pivot\_index:

return arr[k]

elif k < pivot\_index:

return select(arr, left, pivot\_index - 1, k)

else:

return select(arr, pivot\_index + 1, right, k)

def median\_of\_medians(arr, left, right):

# Divide arr into sublists of five elements each

if right - left + 1 <= 5:

return partition5(arr, left, right)

# Array of medians

medians = []

for i in range(left, right + 1, 5):

sub\_right = min(i + 4, right)

median5 = partition5(arr, i, sub\_right)

medians.append(arr[median5])

# Get the median of the medians recursively

return select(medians, 0, len(medians) - 1, len(medians) // 2)

def partition5(arr, left, right):

# Sort the list of five elements

i = left

while i <= right:

j = i

while j > left and arr[j - 1] > arr[j]:

arr[j], arr[j - 1] = arr[j - 1], arr[j]

j -= 1

i += 1

return (left + right) // 2

# Wrapper function

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

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

# Test cases

arr1 = [12, 3, 5, 7, 19]

k1 = 2

print("Expected Output:", 5)

print("Output:", find\_kth\_smallest(arr1, k1))

arr2 = [12, 3, 5, 7, 4, 19, 26]

k2 = 3

print("Expected Output:", 5)

print("Output:", find\_kth\_smallest(arr2, k2))

arr3 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

k3 = 6

print("Expected Output:", 6)

print("Output:", find\_kth\_smallest(arr3, k3))

Expected Output: 5

Output: 5

Expected Output: 5

Output: 5

Expected Output: 6

Output: 6

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.

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.

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

# Move pivot to end and partition the array around it

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

pivot\_value = arr[high]

store\_index = low

for i in range(low, high):

if arr[i] < pivot\_value:

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

store\_index += 1

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

return store\_index

def select(arr, low, high, k):

# Select the k-th smallest element in arr[low..high]

while True:

if low == high:

return arr[low]

pivot\_index = median\_of\_medians(arr[low:high+1])

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

if k == pivot\_index:

return arr[k]

elif k < pivot\_index:

high = pivot\_index - 1

else:

low = pivot\_index + 1

def median\_of\_medians(arr):

if len(arr) <= 5:

return sorted(arr)[len(arr) // 2]

medians = []

for i in range(0, len(arr), 5):

chunk = arr[i:i + 5]

medians.append(sorted(chunk)[len(chunk) // 2])

return median\_of\_medians(medians)

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

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

# Example usage:

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

k1 = 6

print("k-th smallest element:", find\_kth\_smallest(arr1, k1)) # Output should be 6

arr2 = [23, 17, 31, 44, 55, 21, 20, 18, 19, 27]

k2 = 5

print("k-th smallest element:", find\_kth\_smallest(arr2, k2)) # Output should be 20

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.
   1. Set[] = {45, 34, 4, 12, 5, 2} Target Sum : 42

* 1. Set[]= {1, 3, 2, 7, 4, 6} Target sum = 10:

from itertools import combinations

def generate\_subset\_sums(arr):

"""Generate all possible subset sums of the given array."""

subset\_sums = set()

n = len(arr)

# Generate all possible combinations of all sizes

for r in range(n + 1):

for combo in combinations(arr, r):

subset\_sums.add(sum(combo))

return sorted(subset\_sums)

def closest\_subset\_sum(arr, target):

"""Find the subset sum closest to the target using the Meet in the Middle technique."""

# Split array into two halves

mid = len(arr) // 2

set\_a = arr[:mid]

set\_b = arr[mid:]

# Generate all subset sums for each half

sums\_a = generate\_subset\_sums(set\_a)

sums\_b = generate\_subset\_sums(set\_b)

# Initialize variables to track the closest sum

closest\_sum = float('inf')

closest\_diff = float('inf')

# Sort sums\_b to perform binary search

sums\_b.sort()

# For each sum in sums\_a, find the best complement in sums\_b

for sum\_a in sums\_a:

left, right = 0, len(sums\_b) - 1

while left <= right:

mid = (left + right) // 2

sum\_b = sums\_b[mid]

current\_sum = sum\_a + sum\_b

current\_diff = abs(current\_sum - target)

# Update the closest sum if current diff is smaller

if current\_diff < closest\_diff:

closest\_diff = current\_diff

closest\_sum = current\_sum

# Move the search range

if current\_sum < target:

left = mid + 1

else:

right = mid - 1

return closest\_sum

# Test cases

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

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

target1 = 42

target2 = 10

# Finding closest sums

closest\_sum1 = closest\_subset\_sum(set1, target1)

closest\_sum2 = closest\_subset\_sum(set2, target2)

print(f"Closest subset sum to {target1} from set1: {closest\_sum1}")

print(f"Closest subset sum to {target2} from set2: {closest\_sum2}")

Closest subset sum to 42 from set1: [closest sum value]

Closest subset sum to 10 from set2: [closest sum value]

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.
   1. E = {1, 3, 9, 2, 7, 12} exact Sum = 15
   2. E = {3, 34, 4, 12, 5, 2} exact Sum = 15

from itertools import chain, combinations

def all\_subset\_sums(arr):

"""

Generate all possible subset sums of an array.

"""

return {sum(subset) for subset in chain.from\_iterable(combinations(arr, r) for r in range(len(arr) + 1))}

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

"""

Meet in the Middle technique to find if there exists a subset with the exact sum target\_sum.

"""

# Split the array into two halves

mid = len(arr) // 2

left, right = arr[:mid], arr[mid:]

# Generate all possible subset sums for each half

left\_sums = all\_subset\_sums(left)

right\_sums = all\_subset\_sums(right)

# Check if any combination of sums from left\_sums and right\_sums equals target\_sum

for left\_sum in left\_sums:

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

return True

return False

# Test cases

E1 = [1, 3, 9, 2, 7, 12]

target\_sum1 = 15

E2 = [3, 34, 4, 12, 5, 2]

target\_sum2 = 15

# Results

print("Test case 1:", meet\_in\_the\_middle(E1, target\_sum1)) # Expected output: True

print("Test case 2:", meet\_in\_the\_middle(E2, target\_sum2)) # Expected output: True

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.

**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)

def strassen\_2x2(A, B):

# Extract elements from matrix A

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

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

# Extract elements from matrix B

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

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

# Calculate the seven products

M1 = (a + d) \* (e + h)

M2 = (c + d) \* e

M3 = a \* (f - h)

M4 = d \* (g - e)

M5 = (a + b) \* h

M6 = (c - a) \* (e + f)

M7 = (b - d) \* (g + h)

# Compute the entries of the resulting matrix C

C11 = M1 + M4 - M5 + M7

C12 = M3 + M5

C21 = M2 + M4

C22 = M1 - M2 + M3 + M6

# Construct the resulting matrix C

C = [[C11, C12], [C21, C22]]

return C

# Test Case 1

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

B = [[6, 8], [4, 2]]

# Expected Output: [[18, 14], [62, 66]]

C = strassen\_2x2(A, B)

print("Resulting Matrix C:")

for row in C:

print(row)

1. 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

def karatsuba(x, y):

# Base case for recursion: if either number is small, multiply directly

if x < 10 or y < 10:

return x \* y

# Calculate the size of the numbers

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

m = n // 2

# Split x and y into two parts

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

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

# Recursively calculate three products

z0 = karatsuba(low1, low2) # bd

z1 = karatsuba((low1 + high1), (low2 + high2)) # (a+b)(c+d)

z2 = karatsuba(high1, high2) # ac

# Combine the results according to Karatsuba's formula

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

# Test case

x = 1234

y = 5678

z = karatsuba(x, y)

print("Resulting Product Z:", z)