**79.Finding the maximum and minimum**

**PROGRAM:**

import time

def find\_max\_min(lst):

start\_time = time.time() # Start time measurement

maximum = max(lst)

minimum = min(lst)

end\_time = time.time() # End time measurement elapsed\_time = end\_time - start\_time

return maximum, minimum, elapsed\_time

# Example usage

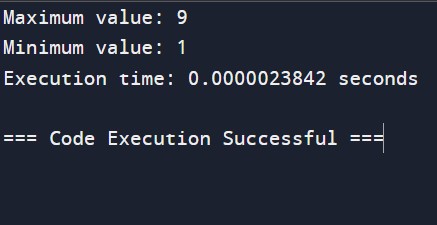
example\_list = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]

max\_value, min\_value, execution\_time = find\_max\_min(example\_list)

print(f"Maximum value: {max\_value}") print(f"Minimum value: {min\_value}")

print(f"Execution time: {execution\_time:.10f} seconds")

OUTPUT:-



TIME COMPLEXITY:-O(n)

**80. Merge sort**

**PROGRAM:**

:- import time

def merge\_sort(arr): if len(arr) > 1:

mid = len(arr) // 2 # Finding the mid of the array left\_half = arr[:mid] # Dividing the array elements into 2 halves right\_half = arr[mid:]

merge\_sort(left\_half) # Sorting the first half

merge\_sort(right\_half) # Sorting the second half

1. = j = k = 0

# Copy data to temp arrays L[] and R[] while i < len(left\_half) and j < len(right\_half): if left\_half[i] < right\_half[j]:

arr[k] = left\_half[i]

* 1. += 1 else: arr[k] = right\_half[j]
  2. += 1 k += 1

# Checking if any element was left while i < len(left\_half): arr[k] = left\_half[i]

i += 1

k += 1

while j < len(right\_half): arr[k] = right\_half[j]

1. += 1
2. += 1

def find\_merge\_sort\_time(arr):

start\_time = time.time() # Start time measurement

merge\_sort(arr) # Perform merge sort

end\_time = time.time() # End time measurement

elapsed\_time = end\_time - start\_time

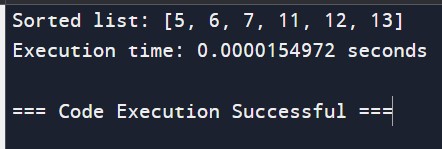
return elapsed\_time

# Example usage example\_list = [12, 11, 13, 5, 6, 7]

execution\_time = find\_merge\_sort\_time(example\_list) print(f"Sorted list: {example\_list}")

print(f"Execution time: {execution\_time:.10f} seconds")

**OUTPUT**:-



**TIME COMPLEXITY**:-O(n log n)

**81. Quick sort**

**PROGRAM:**

import time

def quick\_sort(arr): if len(arr) <= 1: return arr else:

pivot = arr[len(arr) // 2] left = [x for x in arr if x < pivot] middle = [x for x in arr if x == pivot] right = [x for x in arr if x > pivot]

return quick\_sort(left) + middle + quick\_sort(right)

def find\_quick\_sort\_time(arr):

start\_time = time.time() # Start time measurement

sorted\_arr = quick\_sort(arr) # Perform quick sort

end\_time = time.time() # End time measurement elapsed\_time = end\_time - start\_time

return sorted\_arr, elapsed\_time

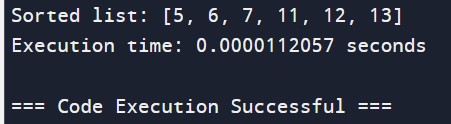
# Example usage example\_list = [12, 11, 13, 5, 6, 7]

sorted\_list, execution\_time = find\_quick\_sort\_time(example\_list)

print(f"Sorted list: {sorted\_list}")

print(f"Execution time: {execution\_time:.10f} seconds")

**OUTPUT**:-



**TIME COMPLEXITY**:-O(n log n)

**82. Binary search**

**PROGRAM:**

:- import time

def binary\_search(arr, x): left, right = 0, len(arr) - 1 while left <= right:

mid = left + (right - left) // 2 if arr[mid] == x: return mid elif arr[mid] < x: left = mid + 1 else: right = mid - 1 return -1

def find\_binary\_search\_time(arr, x):

start\_time = time.time() # Start time measurement

index = binary\_search(arr, x) # Perform binary search

end\_time = time.time() # End time measurement elapsed\_time = end\_time - start\_time

return index, elapsed\_time

# Example usage

example\_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

search\_element = 7

index, execution\_time = find\_binary\_search\_time(example\_list, search\_element)

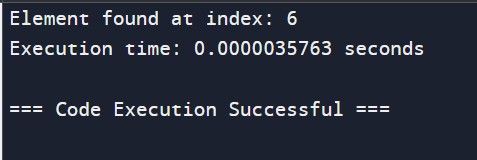
if index != -1:

print(f"Element found at index: {index}") else:

print("Element not found")

print(f"Execution time: {execution\_time:.10f} seconds")

**OUTPUT**:-



**TIME COMPLEXITY**:-O( log n)

**83. Strassens matrix multiplication**

**PROGRAM:**

import numpy as np import time

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

n = len(A)

if n == 1:

return A \* B

# Splitting matrices into quadrants

mid = n // 2 A11 = A[:mid, :mid]

A12 = A[:mid, mid:]

A21 = A[mid:, :mid]

A22 = A[mid:, mid:]

B11 = B[:mid, :mid]

B12 = B[:mid, mid:]

B21 = B[mid:, :mid]

B22 = B[mid:, mid:]

# Computing the 7 products using Strassen's formulas

M1 = strassen\_matrix\_multiplication(A11 + A22, B11 + B22)

M2 = strassen\_matrix\_multiplication(A21 + A22, B11)

M3 = strassen\_matrix\_multiplication(A11, B12 - B22)

M4 = strassen\_matrix\_multiplication(A22, B21 - B11)

M5 = strassen\_matrix\_multiplication(A11 + A12, B22)

M6 = strassen\_matrix\_multiplication(A21 - A11, B11 + B12)

M7 = strassen\_matrix\_multiplication(A12 - A22, B21 + B22)

# Combining the results to get the final quadrants

C11 = M1 + M4 - M5 + M7

C12 = M3 + M5

C21 = M2 + M4

C22 = M1 - M2 + M3 + M6

# Combining the quadrants into a single matrix

C = np.vstack((np.hstack((C11, C12)), np.hstack((C21, C22))))

return C

def find\_strassen\_time(A, B):

start\_time = time.time() # Start time measurement

C = strassen\_matrix\_multiplication(A, B) # Perform Strassen's matrix multiplication

end\_time = time.time() # End time measurement elapsed\_time = end\_time - start\_time

return C, elapsed\_time

# Example usage

n = 4 # Size of the matrix (must be a power of 2)

1. = np.random.randint(0, 10, (n, n))
2. = np.random.randint(0, 10, (n, n))

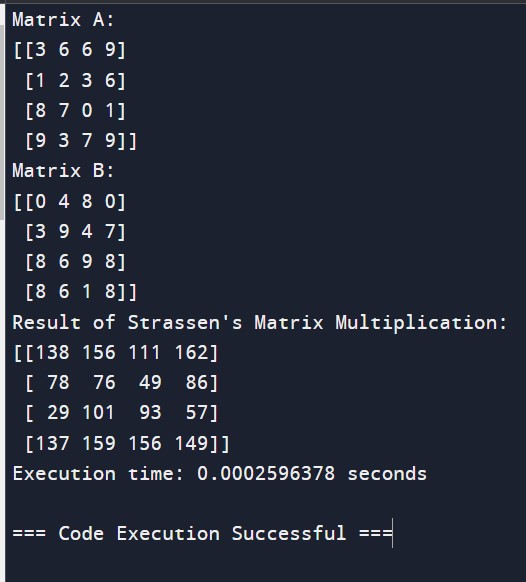
result, execution\_time = find\_strassen\_time(A, B)

print("Matrix A:") print(A) print("Matrix B:") print(B)

print("Result of Strassen's Matrix Multiplication:") print(result)

print(f"Execution time: {execution\_time:.10f} seconds")

**OUTPUT:**-



TIME COMPLEXITY:-O(n3)

**84. Karatsuba algorithm for multiplication**

**PROGRAM:**

:- import time

def karatsuba(x, y): # Base case for recursion 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 the digit sequences about the middle high1, low1 = divmod(x, 10\*\*m)

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

# Perform 3 multiplications z0 = karatsuba(low1, low2) z1 = karatsuba((low1 + high1), (low2 + high2)) z2 = karatsuba(high1, high2)

# Combine the results

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

def find\_karatsuba\_time(x, y):

start\_time = time.time() # Start time measurement

result = karatsuba(x, y) # Perform Karatsuba multiplication

end\_time = time.time() # End time measurement

elapsed\_time = end\_time - start\_time

return result, elapsed\_time

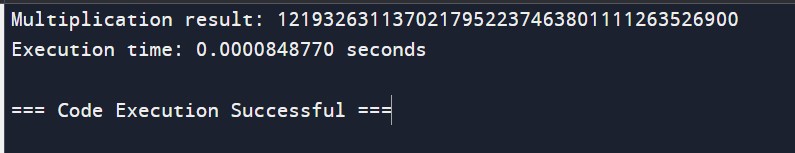
# Example usage x = 12345678901234567890 y = 98765432109876543210

result, execution\_time = find\_karatsuba\_time(x, y)

print(f"Multiplication result: {result}")

print(f"Execution time: {execution\_time:.10f} seconds")

**OUTPUT**:-



TIME COMPLEXITY:- O(n^log\_2{3}) ≈ O(n^1.585)

**85. Closest pair of points using divide and conquer**

**PROGRAM:**

:- import math import time

def distance(point1, point2):

return math.sqrt((point1[0] - point2[0])\*\*2 + (point1[1] - point2[1])\*\*2)

def closest\_pair\_dc(points): def closest\_pair\_rec(points\_sorted\_x, points\_sorted\_y):

n = len(points\_sorted\_x) if n <= 3:

return brute\_force\_closest\_pair(points\_sorted\_x)

mid = n // 2

mid\_point = points\_sorted\_x[mid]

left\_x = points\_sorted\_x[:mid]

right\_x = points\_sorted\_x[mid:]

midpoint = points\_sorted\_x[mid][0] left\_y = list(filter(lambda x: x[0] <= midpoint, points\_sorted\_y)) right\_y = list(filter(lambda x: x[0] > midpoint, points\_sorted\_y))

(p1\_left, p2\_left, dist\_left) = closest\_pair\_rec(left\_x, left\_y)

(p1\_right, p2\_right, dist\_right) = closest\_pair\_rec(right\_x, right\_y)

if dist\_left < dist\_right: min\_dist = dist\_left min\_pair = (p1\_left, p2\_left) else:

min\_dist = dist\_right

min\_pair = (p1\_right, p2\_right)

(p3, p4, dist\_split) = closest\_split\_pair(points\_sorted\_x, points\_sorted\_y, min\_dist, min\_pair)

if min\_dist <= dist\_split:

return min\_pair[0], min\_pair[1], min\_dist else:

return p3, p4, dist\_split

def brute\_force\_closest\_pair(points):

min\_dist = float('inf') p1, p2 = None, None n = len(points) for i in range(n): for j in range(i + 1, n): d = distance(points[i], points[j]) if d < min\_dist: min\_dist = d

p1, p2 = points[i], points[j] return p1, p2, min\_dist

def closest\_split\_pair(points\_sorted\_x, points\_sorted\_y, delta, best\_pair): n = len(points\_sorted\_x)

mid\_x = points\_sorted\_x[n // 2][0]

sy = [p for p in points\_sorted\_y if mid\_x - delta <= p[0] <= mid\_x + delta]

best = delta ln\_sy = len(sy) for i in range(ln\_sy - 1): for j in range(i + 1, min(i + 7, ln\_sy)):

p, q = sy[i], sy[j] dst = distance(p, q) if dst < best: best\_pair = (p, q)

best = dst

return best\_pair[0], best\_pair[1], best

points\_sorted\_x = sorted(points, key=lambda x: x[0]) points\_sorted\_y = sorted(points, key=lambda x: x[1])

return closest\_pair\_rec(points\_sorted\_x, points\_sorted\_y)

def find\_closest\_pair\_time(points):

start\_time = time.time() # Start time measurement

result = closest\_pair\_dc(points) # Perform closest pair of points

end\_time = time.time() # End time measurement elapsed\_time = end\_time - start\_time

return result, elapsed\_time

# Example usage

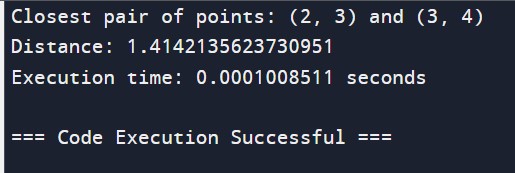
points = [(2, 3), (12, 30), (40, 50), (5, 1), (12, 10), (3, 4)]

result, execution\_time = find\_closest\_pair\_time(points)

print(f"Closest pair of points: {result[0]} and {result[1]}") print(f"Distance: {result[2]}")

print(f"Execution time: {execution\_time:.10f} seconds")

**OUTPUT**:-



TIME COMPLEXITY:- *O*(*n*log*n*)

**86. Median of medians**

**PROGRAM:**

import time

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(arr, low, high, k): if low == high: return arr[low]

pivot\_index = median\_of\_medians(arr, low, high)

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

if k == pivot\_index: return arr[k] elif k < pivot\_index: return select(arr, low, pivot\_index - 1, k) else:

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

def median\_of\_medians(arr, low, high):

n = high - low + 1 if n < 10:

return partition5(arr, low, high)

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

sub\_right = i + 4 if sub\_right > high: sub\_right = high median5 = partition5(arr, i, sub\_right)

medians.append(arr[median5])

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

def partition5(arr, low, high): sublist = arr[low:high + 1] sublist.sort() mid = (len(sublist) - 1) // 2 median = sublist[mid]

median\_index = arr.index(median, low, high + 1) return median\_index

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

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

def find\_median\_of\_medians\_time(arr):

start\_time = time.time() # Start time measurement

n = len(arr) if n % 2 == 1:

median = find\_kth\_smallest(arr, n // 2 + 1) else:

left = find\_kth\_smallest(arr, n // 2) right = find\_kth\_smallest(arr, n // 2 + 1) median = (left + right) / 2

end\_time = time.time() # End time measurement elapsed\_time = end\_time - start\_time

return median, elapsed\_time

# Example usage

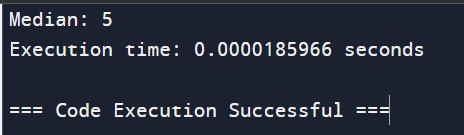
arr = [3, 2, 9, 1, 7, 6, 8, 5, 4]

median, execution\_time = find\_median\_of\_medians\_time(arr)

print(f"Median: {median}")

print(f"Execution time: {execution\_time:.10f} seconds")

**OUTPUT**:-



**TIME COMPLEXITY**:-o(n)

**87. Meet in middle technique**

**PROGRAM:**

import itertools import time

def meet\_in\_the\_middle(numbers, target): # Divide the list into two halves mid = len(numbers) // 2 left\_half = numbers[:mid]

right\_half = numbers[mid:]

# Generate all subset sums for each half

left\_sums = set(sum(subset) for i in range(len(left\_half) + 1) for subset in itertools.combinations(left\_half, i))

right\_sums = set(sum(subset) for i in range(len(right\_half) + 1) for subset in itertools.combinations(right\_half, i))

# Check if there is a combination of sums from left and right halves that equals the target for l\_sum in left\_sums: if (target - l\_sum) in right\_sums:

return True

return False

def find\_meet\_in\_the\_middle\_time(numbers, target): start\_time = time.time() # Start time measurement

result = meet\_in\_the\_middle(numbers, target) # Perform the meet-in-the-middle technique

end\_time = time.time() # End time measurement elapsed\_time = end\_time - start\_time

return result, elapsed\_time

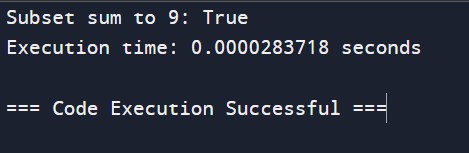
# Example usage numbers = [3, 34, 4, 12, 5, 2] target = 9

result, execution\_time = find\_meet\_in\_the\_middle\_time(numbers, target)

print(f"Subset sum to {target}: {result}")

print(f"Execution time: {execution\_time:.10f} seconds")

**OUTPUT:**-



**TIME COMPLEXITY**:- O(2^(n/2) \* n)