1.a. . Given an unsorted array 10,16,8,12,15,6,3,9,5 Write a program to perform Quick Sort. Choose the first element as the pivot and partition the array accordingly. Show the array after this partition. Recursively apply Quick Sort on the sub-arrays formed. Display the array after each recursive call until the entire array is sorted.

Program:

def partition(arr, low, high):

pivot = arr[low]

i = low + 1

j = high

while True:

while i <= j and arr[i] <= pivot:

i += 1

while i <= j and arr[j] > pivot:

j -= 1

if i <= j:

arr[i], arr[j] = arr[j], arr[i]

else:

break

arr[low], arr[j] = arr[j], arr[low]

return j

def quick\_sort(arr, low, high):

if low < high:

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

print(arr)

quick\_sort(arr, low, pivot\_index - 1)

quick\_sort(arr, pivot\_index + 1, high)

# Input

N = 9

a = [10, 16, 8, 12, 15, 6, 3, 9, 5]

# Perform Quick Sort

quick\_sort(a, 0, N - 1)

1.b. Implement the Quick Sort algorithm in a programming language of your choice and test it on the array 19,72,35,46,58,91,22,31. Choose the middle element as the pivot and partition the array accordingly. Show the array after this partition. Recursively apply Quick Sort on the sub-arrays formed. Display the array after each recursive call until the entire array is sorted. Execute your code and show the sorted array.

Program:

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)

# Test the Quick Sort algorithm on the array [19, 72, 35, 46, 58, 91, 22, 31]

array = [19, 72, 35, 46, 58, 91, 22, 31]

sorted\_array = quick\_sort(array)

print(sorted\_array)

2.a. **Binary Search**

**1.** Implement the Binary Search algorithm in a programming language of your choice and test it on the array 5,10,15,20,25,30,35,40,45 to find the position of the element 20.

Execute your code and provide the index of the element 20. Modify your implementation to count the number of comparisons made during the search process. Print this count along with the result

Program:

def binary\_search(arr, x):

low = 0

high = len(arr) - 1

mid = 0

count = 0

while low <= high:

mid = (high + low) // 2

count += 1

if arr[mid] < x:

low = mid + 1

elif arr[mid] > x:

high = mid - 1

else:

return mid, count

return -1, count

# Test the binary search algorithm on the array [5, 10, 15, 20, 25, 30, 35, 40, 45] to find the position of 20

arr = [5, 10, 15, 20, 25, 30, 35, 40, 45]

x = 20

result, comparisons = binary\_search(arr, x)

if result != -1:

print(f"Element found at index {result} with {comparisons} comparisons.")

else:

print("Element not found.")

2.b. You are given a sorted array 3,9,14,19,25,31,42,47,53 and asked to find the position of the element 31 using Binary Search. Show the mid-point calculations and the steps involved in finding the element. Display, what would happen if the array was not sorted, how would this impact the performance and correctness of the Binary Search algorithm?

Program:

def binary\_search(arr, x):

low = 0

high = len(arr) - 1

mid = 0

while low <= high:

mid = (high + low) // 2

if arr[mid] < x:

low = mid + 1

elif arr[mid] > x:

high = mid - 1

else:

return mid

return -1

# Sorted array

arr = [3, 9, 14, 19, 25, 31, 42, 47, 53]

x = 31

result = binary\_search(arr, x)

print("Element found at index:", result)

3. **Optimal Binary Search Trees**

1. Implement the Optimal Binary Search Tree algorithm for the keys A,B,C,D with frequencies 0.1,0.2,0.4,0.3 Write the code using any programming language to construct the OBST for the given keys and frequencies. Execute your code and display the resulting OBST and its cost. Print the cost and root matrix.

Program:

def optimal\_bst(keys, freq):

n = len(keys)

cost = [[0 for \_ in range(n)] for \_ in range(n)]

root = [[0 for \_ in range(n)] for \_ in range(n)]

for i in range(n):

cost[i][i] = freq[i]

root[i][i] = i

for l in range(2, n + 1):

for i in range(n - l + 1):

j = i + l - 1

cost[i][j] = float('inf')

for k in range(i, j + 1):

c = cost[i][k - 1] if k > i else 0

c += cost[k + 1][j] if k < j else 0

c += sum(freq[i:j + 1])

if c < cost[i][j]:

cost[i][j] = c

root[i][j] = k

return cost, root

keys = ['A', 'B', 'C', 'D']

freq = [0.1, 0.2, 0.4, 0.3]

cost\_table, root\_table = optimal\_bst(keys, freq)

print("Cost Table")

for row in cost\_table:

print(row)

print("\nRoot Table")

for row in root\_table:

print(row)

print("\nCost of Optimal Binary Search Tree:", cost\_table[0][-1])

2. Consider a set of keys 10,12,16,21 with frequencies 4,2,6,3 and the respective probabilities. Write a Program to construct an OBST in a programming language of your choice. Execute your code and display the resulting OBST, its cost and root matrix.

Program:

import sys

def obst(keys, freq, n):

cost = [[0 for \_ in range(n)] for \_ in range(n)]

root = [[0 for \_ in range(n)] for \_ in range(n)]

for i in range(n):

cost[i][i] = freq[i]

root[i][i] = i

for L in range(2, n + 1):

for i in range(n - L + 1):

j = i + L - 1

cost[i][j] = sys.maxsize

for r in range(i, j + 1):

c = 0 if r == i else cost[i][r - 1]

c += 0 if r == j else cost[r + 1][j]

c += sum(freq[i:j + 1])

if c < cost[i][j]:

cost[i][j] = c

root[i][j] = r

return cost, root

def display\_obst(keys, freq, n):

cost, root = obst(keys, freq, n)

print("Cost Matrix:")

for row in cost:

print(row)

print("\nRoot Matrix:")

for row in root:

print(row)

# Input

N = 4

Keys = [10, 12, 16, 21]

Frequencies = [4, 2, 6, 3]

# Display OBST, its cost, and root matrix

display\_obst(Keys, Frequencies, N)