1).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.

Input : N= 9, a[]= {10,16,8,12,15,6,3,9,5}

Output : 3,5,6,8,9,10,12,15,16

Test Cases :

Input : N= 8, a[] = {12,4,78,23,45,67,89,1}

Output : 1,4,12,23,45,67,78,89

Test Cases :

Input : N= 7, a[] = {38,27,43,3,9,82,10}

Output : 3,9,10,27,38,43,82,

SOL:

def quick\_sort(arr):

if len(arr) <= 1:

return arr

else:

pivot = arr[0]

less\_than\_pivot = [x for x in arr[1:] if x <= pivot]

greater\_than\_pivot = [x for x in arr[1:] if x > pivot]

return quick\_sort(less\_than\_pivot) + [pivot] + quick\_sort(greater\_than\_pivot)

# Test Case 1

arr1 = [10, 16, 8, 12, 15, 6, 3, 9, 5]

sorted\_arr1 = quick\_sort(arr1)

print(sorted\_arr1)

# Test Case 2

arr2 = [12, 4, 78, 23, 45, 67, 89, 1]

sorted\_arr2 = quick\_sort(arr2)

print(sorted\_arr2)

# Test Case 3

arr3 = [38, 27, 43, 3, 9, 82, 10]

sorted\_arr3 = quick\_sort(arr3)

print(sorted\_arr3)

2) **.** 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.

Input : N= 9, a[] = {5,10,15,20,25,30,35,40,45}, search key = 20

Output : 4

Test cases

Input : N= 6, a[] = {10,20,30,40,50,60}, search key = 50

Output : 5

Input : N= 7, a[] = {21,32,40,54,65,76,87}, search key = 32

Output : 2

SOL:

def binary\_search(arr, target):

low = 0

high = len(arr) - 1

count = 0

while low <= high:

mid = (low + high) // 2

count += 1

if arr[mid] == target:

return mid, count

elif arr[mid] < target:

low = mid + 1

else:

high = mid - 1

return -1, count

# Test the binary search algorithm

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

target = 20

index, comparisons = binary\_search(arr, target)

print(f"Index of {target} is: {index}")

print(f"Number of comparisons made: {comparisons}")

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.

Input N =4, Keys = {A,B,C,D} Frequencies = {01.02.,0.3,0.4}

Output : 1.7

Cost Table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
| 1 | 0 | 0.1 | 0.4 | 1.1 | 1.7 |
| 2 |  | 0 | 0.2 | 0.8 | 0.4 |
| 3 |  |  | 0 | 0.4 | 1.0 |
| 4 |  |  |  | 0 | 0.3 |
| 5 |  |  |  |  | 0 |

Root table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 |
| 1 | 1 | 2 | 3 | 3 |
| 2 |  | 2 | 3 | 3 |
| 3 |  |  | 3 | 3 |
| 4 |  |  |  | 4 |

1. Test cases

Input: keys[] = {10, 12}, freq[] = {34, 50}

Output = 118

1. Input: keys[] = {10, 12, 20}, freq[] = {34, 8, 50}

Output = 142

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.

Input N =4, Keys = {10,12,16,21} Frequencies = {4,2,6,3}

Output : 26

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 |
| 0 | 4 | 80 | 202 | 262 |
| 1 |  | 2 | 102 | 162 |
| 2 |  |  | 6 | 12 |
| 3 |  |  |  | 3 |

1. Test cases

Input: keys[] = {10, 12}, freq[] = {34, 50}

Output = 118

1. Input: keys[] = {10, 12, 20}, freq[] = {34, 8, 50}

Output = 142

SOL:

import numpy as np

def optimal\_bst(keys, freq, n):

# Initialize cost and root matrices

cost = np.zeros((n+1, n+1))

root = np.zeros((n, n), dtype=int)

# Fill diagonal elements in cost matrix with frequencies

for i in range(n):

cost[i][i+1] = freq[i]

# Fill the cost and root matrices

for length in range(2, n+1): # length is the chain length

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

j = i + length

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

total\_freq = sum(freq[i:j])

for r in range(i, j):

c = cost[i][r] + cost[r+1][j] + total\_freq

if c < cost[i][j]:

cost[i][j] = c

root[i][j-1] = r

return cost, root

def print\_matrix(matrix):

for row in matrix:

print("\t".join(map(str, row)))

# Example input

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

freq = [0.1, 0.2, 0.4, 0.3]

n = len(keys)

# Calculate the OBST

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

# Display the cost and root matrices

print("Cost Matrix:")

print\_matrix(cost)

print("\nRoot Matrix:")

print\_matrix(root)

# The optimal cost

print(f"\nOptimal cost: {cost[0][n]}")