QUICK SORT

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

Program:

```
def quick_sort(arr, low, high):
    if low < high:
        pivot_index = partition(arr, low, high)
        print(f"Array after partitioning with pivot index {pivot_index}: {arr}")

        quick_sort(arr, low, pivot_index - 1)

        quick_sort(arr, pivot_index + 1, high)

def partition(arr, low, high):
    pivot = arr[low]
    left = low + 1
    right = high

    while True:
        while left <= right and arr[left] <= pivot:
            left += 1
        while left <= right and arr[right] >= pivot:
            right -= 1
```

```
        if left <= right:
            arr[left], arr[right] = arr[right], arr[left]
        else:
            break

    arr[low], arr[right] = arr[right], arr[low]
    return right

# Test cases
test_cases = [
    [10, 16, 8, 12, 15, 6, 3, 9, 5],
    [12, 4, 78, 23, 45, 67, 89, 1],
    [38, 27, 43, 3, 9, 82, 10]
]

for case in test_cases:
    print(f"Original array: {case}")
    quick_sort(case, 0, len(case) - 1)
    print(f"Sorted array: {case}\n")
```

2. 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.

Input : N= 8, a[] = {19,72,35,46,58,91,22,31}
Output : 19,22,31,35,46,58,72,91
Test Cases :
Input : N= 8, a[] = {31,23,35,27,11,21,15,28}
Output : 11,15,21,23,27,28,31,35
Test Cases :
Input : N= 10, a[] = {22,34,25,36,43,67, 52,13,65,17}
Output : 13,17,22,25,34,36,43,52,65,67

Program:

```
def quick_sort(arr, low, high):
    if low < high:
```

```
        pivot_index = partition(arr, low, high)
        print(f"Array after partitioning with pivot index {pivot_index}: {arr}")

        quick_sort(arr, low, pivot_index - 1)

        quick_sort(arr, pivot_index + 1, high)

def partition(arr, low, high):
    mid = (low + high) // 2
    pivot = arr[mid]
    arr[mid], arr[low] = arr[low], arr[mid]
    left = low + 1
    right = high

    while True:
        while left <= right and arr[left] <= pivot:
            left += 1
        while left <= right and arr[right] >= pivot:
            right -= 1
        if left <= right:
            arr[left], arr[right] = arr[right], arr[left]
        else:
            break

    arr[low], arr[right] = arr[right], arr[low]
    return right

# Test cases
test_cases = [
    [19, 72, 35, 46, 58, 91, 22, 31],
    [31, 23, 35, 27, 11, 21, 15, 28],
    [22, 34, 25, 36, 43, 67, 52, 13, 65, 17]
]

for case in test_cases:
    print(f"Original array: {case}")
    quick_sort(case, 0, len(case) - 1)
    print(f"Sorted array: {case}\n")
```

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.

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

Program:

```
def binary_search(arr, low, high, key):
    comparisons = 0
    while low <= high:
        mid = (low + high) // 2
        comparisons += 1
        if arr[mid] == key:
            return mid, comparisons
        elif arr[mid] < key:
            low = mid + 1
        else:
            high = mid - 1
    return -1, comparisons

# Test cases
test_cases = [
    ([5, 10, 15, 20, 25, 30, 35, 40, 45], 20),
    ([10, 20, 30, 40, 50, 60], 50),
    ([21, 32, 40, 54, 65, 76, 87], 32)
]

for arr, key in test_cases:
    index, comparisons = binary_search(arr, 0, len(arr) - 1, key)
    print(f"Array: {arr}")
    print(f"Search key: {key}")
    print(f"Index of {key}: {index} (0-based index)")
    print(f"Number of comparisons: {comparisons}\n")
```

2. 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?
Input : N= 9, a[] = {3,9,14,19,25,31,42,47,53}, search key = 31
Output : 6
Test cases
Input : N= 7, a[] = {13,19,24,29,35,41,42}, search key = 42
Output : 7

Test cases
Input : N= 6, a[] = {20,40,60,80,100,120}, search key = 60
Output : 3

Program:

```
def binary_search_with_steps(arr, low, high, key):
    comparisons = 0
    steps = []
    while low <= high:
        mid = (low + high) // 2
        comparisons += 1
        steps.append((low, mid, high))
        if arr[mid] == key:
            return mid, comparisons, steps
        elif arr[mid] < key:
            low = mid + 1
        else:
            high = mid - 1
    return -1, comparisons, steps

# Test case 1
arr1 = [3, 9, 14, 19, 25, 31, 42, 47, 53]
key1 = 31
index1, comparisons1, steps1 = binary_search_with_steps(arr1, 0, len(arr1) - 1, key1)
print(f"Array: {arr1}")
print(f"Search key: {key1}")
print(f"Index of {key1}: {index1} (0-based index)")
print(f"Number of comparisons: {comparisons1}")
print("Steps (low, mid, high):")
for step in steps1:
    print(step)
print("\n")

# Test case 2
arr2 = [13, 19, 24, 29, 35, 41, 42]
key2 = 42
index2, comparisons2, steps2 = binary_search_with_steps(arr2, 0, len(arr2) - 1, key2)
print(f"Array: {arr2}")
print(f"Search key: {key2}")
print(f"Index of {key2}: {index2} (0-based index)")
print(f"Number of comparisons: {comparisons2}")
print("Steps (low, mid, high):")
for step in steps2:
    print(step)
print("\n")
```

```
# Test case 3
arr3 = [20, 40, 60, 80, 100, 120]
key3 = 60
index3, comparisons3, steps3 = binary_search_with_steps(arr3, 0, len(arr3) - 1, key3)
print(f"Array: {arr3}")
print(f"Search key: {key3}")
print(f"Index of {key3}: {index3} (0-based index)")
print(f"Number of comparisons: {comparisons3}")
print("Steps (low, mid, high):")
for step in steps3:
    print(step)
print("\n")
```

OPTIMAL BINARY SEARCH TREE:

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

a) Test cases

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

Output = 118

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

Output = 142

PROGRAM:

```
def optimal_bst(keys, freq):
    n = len(keys)

    cost = [[0] * (n + 1) for _ in range(n + 1)]
    root = [[0] * (n + 1) for _ in range(n + 1)]
    cfreq = [0] * (n + 1)

    cfreq[0] = 0
    for i in range(1, n + 1):
        cfreq[i] = cfreq[i - 1] + freq[i - 1]

    for length in range(1, n + 1):
        for i in range(1, n - length + 2):
            j = i + length - 1
            cost[i][j] = float('inf')
            for r in range(i, j + 1):
                c = cost[i][r - 1] + cost[r + 1][j] + cfreq[j] - cfreq[i - 1]
                if c < cost[i][j]:
                    cost[i][j] = c
                    root[i][j] = r

    print(f"Minimum cost of OBST: {cost[1][n]}")

    print("Cost Table:")
    for i in range(1, n + 1):
        for j in range(1, n + 1):
            if j >= i:
                print(cost[i][j], end="\t")
            else:
                print("-", end="\t")
        print()
```

```
    print("Root Table:")
    for i in range(1, n + 1):
        for j in range(1, n + 1):
            if j >= i:
                print(root[i][j], end="\t")
            else:
                print("-", end="\t")
        print()

# Example usage:
keys = ["A", "B", "C", "D"]
freq = [0.1, 0.2, 0.4, 0.3]

optimal_bst(keys, freq)
```

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	8^0	20^2	26^2
1		2	10^2	16^2
2			6	12
3				3

a) Test cases
Input: keys[] = {10, 12}, freq[] = {34, 50}
Output = 118
b) Input: keys[] = {10, 12, 20}, freq[] = {34, 8, 50}
Output = 142

PROGRAM:

```
def optimal_bst(keys, freq):
    n = len(keys)
```

```
cost = [[0] * (n) for _ in range(n)]
root = [[0] * (n) for _ in range(n)]
cfreq = [0] * (n + 1)

cfreq[0] = 0
for i in range(1, n + 1):
    cfreq[i] = cfreq[i - 1] + freq[i - 1]

for i in range(n):
    cost[i][i] = freq[i]
    root[i][i] = i

for length in range(2, n + 1):
    for i in range(n - length + 1):
        j = i + length - 1
        cost[i][j] = float('inf')
        sum_freq = cfreq[j + 1] - cfreq[i]

        for r in range(i, j + 1):
            c = (cost[i][r - 1] if r > i else 0) + (cost[r + 1][j] if r < j else 0) + sum_freq
            if c < cost[i][j]:
                cost[i][j] = c
                root[i][j] = r

print(f"Minimum cost of OBST: {cost[0][n - 1]}")

# Print cost table
print("Cost Table:")
for i in range(n):
    for j in range(n):
        if j >= i:
            print(cost[i][j], end="\t")
        else:
            print("-", end="\t")
    print()

print("Root Table:")
for i in range(n):
    for j in range(n):
        if j >= i:
            print(root[i][j], end="\t")
        else:
            print("-", end="\t")
    print()
```

```
# Example usage:
keys = [10, 12, 16, 21]
freq = [4, 2, 6, 3]

optimal_bst(keys, freq)
```