NAME : TEJASWINI M

REG NO. : 192411079

COURSE CODE : CSA0613

COURSE NAME : DESIGN AND ANALYSIS OF ALGORITHMS FOR OPTIMAL APPLICATIONS

SLOT : A

TOPIC 3 DIVIDE AND CONQUER

1. **Write a Program to find both the maximum and minimum values in the array. Implement using any programming language of your choice. Execute your code and provide the maximum and minimum values found.**

AIM:

To find both the maximum and minimum values in an array efficiently.

ALGORITHM:

1. Start
2. Read the array a[] of size N
3. Initialize min_val and max_val as the first element of the array
4. Traverse the array from index 1 to N-1:
    - If current element < min_val, update min_val
    - If current element > max_val, update max_val
5. Return min_val and max_val
6. Stop

CODE:

```
def find_min_max(arr):
    if not arr:
        return None, None
    min_val = max_val = arr[0]
    for num in arr[1:]:
        if num < min_val:
            min_val = num
        if num > max_val:
            max_val = num
    return min_val, max_val
```

INPUT:

```
# Test Case 1
arr1 = [5, 7, 3, 4, 9, 12, 6, 2]
min1, max1 = find_min_max(arr1)
print("Test Case 1:")
print("Min =", min1, ", Max =", max1)
# Test Case 2
arr2 = [1,3,5,7,9,11,13,15,17]
min2, max2 = find_min_max(arr2)
print("\nTest Case 2:")
print("Min =", min2, ", Max =", max2)
# Test Case 3
arr3 = [22,34,35,36,43,67,12,13,15,17]
min3, max3 = find_min_max(arr3)
print("\nTest Case 3:")
print("Min =", min3, ", Max =", max3)
```

OUTPUT:

```
Test Case 1:
Min = 2 , Max = 12

Test Case 2:
Min = 1 , Max = 17

Test Case 3:
Min = 12 , Max = 67
```

2. **Consider an array of integers sorted in ascending order: 2,4,6,8,10,12,14,18. Write a Program to find both the maximum and minimum values in the array. Implement using any programming language of your choice. Execute your code and provide the maximum and minimum values found.**

AIM:

To find both the maximum and minimum values in a given array of integers.

ALGORITHM:

```
1. Start
2. Read the array a[] of size N
3. Initialize min_val and max_val as the first element of the array
4. Traverse the array from index 1 to N-1:
      o If current element < min_val, update min_val
      o If current element > max_val, update max_val
5. Return min_val and max_val
6. Stop
```

CODE:

```
def find_min_max(arr):
    if not arr:
        return None, None
    min_val = max_val = arr[0]
    for num in arr[1:]:
        if num < min_val:
            min_val = num
        if num > max_val:
            max_val = num
    return min_val, max_val
```

INPUT:

```
# Test Case 1
arr1 = [2,4,6,8,10,12,14,18]
min1, max1 = find_min_max(arr1)
print("Test Case 1:")
print("Min =", min1, ", Max =", max1)
# Test Case 2
arr2 = [11,13,15,17,19,21,23,35,37]
min2, max2 = find_min_max(arr2)
print("\nTest Case 2:")
print("Min =", min2, ", Max =", max2)
# Test Case 3
arr3 = [22,34,35,36,43,67,12,13,15,17]
min3, max3 = find_min_max(arr3)
print("\nTest Case 3:")
print("Min =", min3, ", Max =", max3)
```

OUTPUT:

```
Test Case 1:
Min = 2 , Max = 18

Test Case 2:
Min = 11 , Max = 37

Test Case 3:
Min = 12 , Max = 67
```

3. **You are given an unsorted array 31,23,35,27,11,21,15,28. Write a program for Merge Sort and implement using any programming language of your choice.**

AIM:

To sort an unsorted array using the Merge Sort algorithm, which uses a divide-and-conquer approach to recursively split, sort, and merge subarrays.

ALGORITHM:

```
1. Start
2. Read the array a[] of size N
3. If array length > 1:
      o Divide the array into two halves
      o Recursively apply Merge Sort on left half
      o Recursively apply Merge Sort on right half
      o Merge the two sorted halves into a single sorted array
4. Return the sorted array
5. Stop
```

CODE:

```
def merge_sort(arr):
   if len(arr) > 1:
      mid = len(arr)//2
      left = arr[:mid]
      right = arr[mid:]

      merge_sort(left)
      merge_sort(right)

      i = j = k = 0
```

```
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            arr[k] = left[i]
            i += 1
        else:
            arr[k] = right[j]
            j += 1
        k += 1

    while i < len(left):
        arr[k] = left[i]
        i += 1
        k += 1

    while j < len(right):
        arr[k] = right[j]
        j += 1
        k += 1
```

INPUT:

```
# Test Case 1
arr1 = [31,23,35,27,11,21,15,28]
merge_sort(arr1)
print("Test Case 1 Sorted Array:", arr1)

# Test Case 2
arr2 = [22,34,25,36,43,67,52,13,65,17]
merge_sort(arr2)
print("\nTest Case 2 Sorted Array:", arr2)
```

OUTPUT:

```
Test Case 1 Sorted Array: [11, 15, 21, 23, 27, 28, 31, 35]

Test Case 2 Sorted Array: [13, 17, 22, 25, 34, 36, 43, 52, 65, 67]
```

4. **Write code for Insertion Sort that manages arrays with duplicate elements during the sorting process. Ensure the algorithm's behavior when encountering duplicate values, including whether it preserves the relative order of duplicates and how it affects the overall sorting outcome.**

AIM:

To implement the Merge Sort algorithm and count the number of comparisons made during the sorting process.

ALGORITHM:

1. Start
2. Read the array a[] of size N
3. Initialize a global variable comparisons = 0
4. If array length > 1:
    - Divide the array into two halves
    - Recursively apply Merge Sort on left half
    - Recursively apply Merge Sort on right half
    - While merging, increment comparisons for each comparison between elements of left and right halves

5. Return the sorted array and total comparison count
6. Stop

CODE:

```
comparisons = 0  # Global variable to count comparisons
def merge_sort_count(arr):
    global comparisons
    if len(arr) > 1:
        mid = len(arr)//2
        left = arr[:mid]
        right = arr[mid:]
        merge_sort_count(left)
        merge_sort_count(right)
        i = j = k = 0
        while i < len(left) and j < len(right):
            comparisons += 1  # Count each comparison
            if left[i] < right[j]:
                arr[k] = left[i]
                i += 1
            else:
                arr[k] = right[j]
                j += 1
            k += 1
        while i < len(left):
            arr[k] = left[i]
            i += 1
            k += 1
        while j < len(right):
            arr[k] = right[j]
            j += 1
            k += 1
```

INPUT:

```
# Test Case 1
comparisons = 0
arr1 = [12,4,78,23,45,67,89,1]
merge_sort_count(arr1)
print("Test Case 1 Sorted Array:", arr1)
print("Comparisons:", comparisons)

# Test Case 2
comparisons = 0
arr2 = [38,27,43,3,9,82,10]
merge_sort_count(arr2)
print("\nTest Case 2 Sorted Array:", arr2)
print("Comparisons:", comparisons)
```

OUTPUT:

```
Test Case 1 Sorted Array: [1, 4, 12, 23, 45, 67, 78, 89]

Comparisons: 16


Test Case 2 Sorted Array: [3, 9, 10, 27, 38, 43, 82]

Comparisons: 13
```

5. **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.**

AIM:

To sort an unsorted array using the Quick Sort algorithm, selecting the first element as the pivot and recursively sorting sub-arrays.

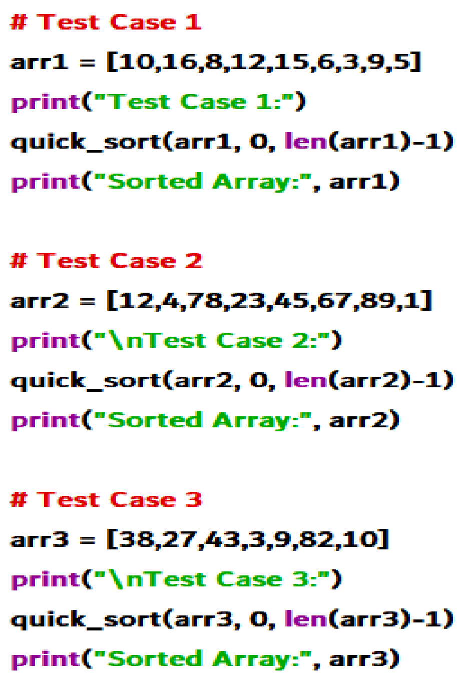
ALGORITHM:

1. Start
2. Read the array a[] of size N
3. Choose the first element of the array as the pivot
4. Partition the array into two sub-arrays:
   - Elements smaller than pivot go to the left
   - Elements greater than pivot go to the right
5. Recursively apply Quick Sort to left and right sub-arrays
6. Combine the sorted sub-arrays and pivot to get the sorted array
7. Print the array after each partition and after each recursive call
8. Stop

CODE:

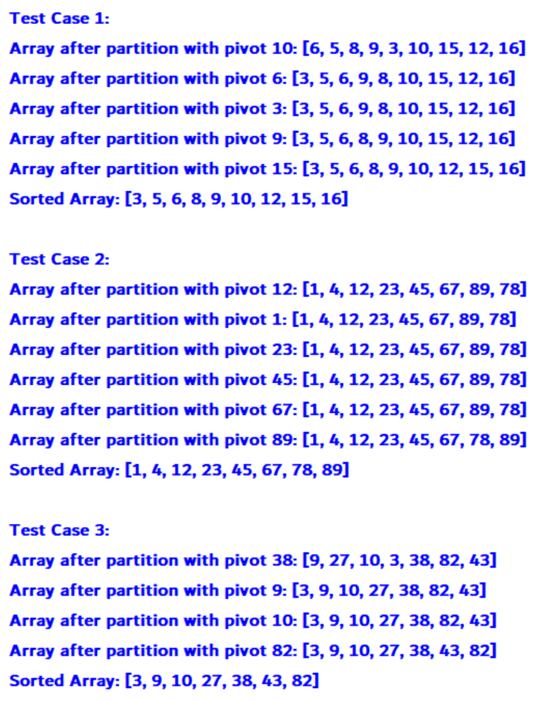
```python
def quick_sort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        print(f"Array after partition with pivot {arr[pi]}: {arr}")
        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)

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

    done = False
    while not done:
        while left <= right and arr[left] <= pivot:
            left += 1
        while arr[right] >= pivot and right >= left:
            right -= 1
        if right < left:
            done = True
        else:
            arr[left], arr[right] = arr[right], arr[left]

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

INPUT:



```
# Test Case 1
arr1 = [10,16,8,12,15,6,3,9,5]
print("Test Case 1:")
quick_sort(arr1, 0, len(arr1)-1)
print("Sorted Array:", arr1)

# Test Case 2
arr2 = [12,4,78,23,45,67,89,1]
print("\nTest Case 2:")
quick_sort(arr2, 0, len(arr2)-1)
print("Sorted Array:", arr2)

# Test Case 3
arr3 = [38,27,43,3,9,82,10]
print("\nTest Case 3:")
quick_sort(arr3, 0, len(arr3)-1)
print("Sorted Array:", arr3)
```

OUTPUT:



```
Test Case 1:
Array after partition with pivot 10: [6, 5, 8, 9, 3, 10, 15, 12, 16]
Array after partition with pivot 6: [3, 5, 6, 9, 8, 10, 15, 12, 16]
Array after partition with pivot 3: [3, 5, 6, 9, 8, 10, 15, 12, 16]
Array after partition with pivot 9: [3, 5, 6, 8, 9, 10, 15, 12, 16]
Array after partition with pivot 15: [3, 5, 6, 8, 9, 10, 12, 15, 16]
Sorted Array: [3, 5, 6, 8, 9, 10, 12, 15, 16]

Test Case 2:
Array after partition with pivot 12: [1, 4, 12, 23, 45, 67, 89, 78]
Array after partition with pivot 1: [1, 4, 12, 23, 45, 67, 89, 78]
Array after partition with pivot 23: [1, 4, 12, 23, 45, 67, 89, 78]
Array after partition with pivot 45: [1, 4, 12, 23, 45, 67, 89, 78]
Array after partition with pivot 67: [1, 4, 12, 23, 45, 67, 89, 78]
Array after partition with pivot 89: [1, 4, 12, 23, 45, 67, 78, 89]
Sorted Array: [1, 4, 12, 23, 45, 67, 78, 89]

Test Case 3:
Array after partition with pivot 38: [9, 27, 10, 3, 38, 82, 43]
Array after partition with pivot 9: [3, 9, 10, 27, 38, 82, 43]
Array after partition with pivot 10: [3, 9, 10, 27, 38, 82, 43]
Array after partition with pivot 82: [3, 9, 10, 27, 38, 43, 82]
Sorted Array: [3, 9, 10, 27, 38, 43, 82]
```

6. **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.**

AIM:

To sort an unsorted array using the Quick Sort algorithm, selecting the middle element as the pivot, and recursively sorting sub-arrays while displaying the array after each partition.

ALGORITHM:

1. Start
2. Read the array a[] of size N
3. Choose the middle element of the current sub-array as the pivot
4. Partition the array such that elements smaller than pivot go left, larger go right
5. Recursively apply Quick Sort on left and right sub-arrays
6. Print the array after each partition and recursive call

7. Stop

CODE:

```python
def quick_sort_middle(arr, low, high):
    if low < high:
        pi = partition_middle(arr, low, high)
        print(f"Array after partition with pivot {arr[pi]}: {arr}")
        quick_sort_middle(arr, low, pi - 1)
        quick_sort_middle(arr, pi + 1, high)

def partition_middle(arr, low, high):
    mid = (low + high) // 2
    pivot = arr[mid]
    arr[mid], arr[high] = arr[high], arr[mid]  # Move pivot to end for partitioning
    i = low
    for j in range(low, high):
        if arr[j] <= pivot:
            arr[i], arr[j] = arr[j], arr[i]
            i += 1
    arr[i], arr[high] = arr[high], arr[i]  # Move pivot to correct position
    return i
```

INPUT:

```python
# Test Case 1
arr1 = [19,72,35,46,58,91,22,31]
print("Test Case 1:")
quick_sort_middle(arr1, 0, len(arr1)-1)
print("Sorted Array:", arr1)

# Test Case 2
arr2 = [31,23,35,27,11,21,15,28]
print("\nTest Case 2:")
quick_sort_middle(arr2, 0, len(arr2)-1)
print("Sorted Array:", arr2)

# Test Case 3
arr3 = [22,34,25,36,43,67,52,13,65,17]
print("\nTest Case 3:")
quick_sort_middle(arr3, 0, len(arr3)-1)
print("Sorted Array:", arr3)
```

OUTPUT:

```
Test Case 1:
Array after partition with pivot 46: [19, 35, 31, 22, 46, 91, 72, 58]
Array after partition with pivot 35: [19, 22, 31, 35, 46, 91, 72, 58]
Array after partition with pivot 22: [19, 22, 31, 35, 46, 91, 72, 58]
Array after partition with pivot 72: [19, 22, 31, 35, 46, 58, 72, 91]
Sorted Array: [19, 22, 31, 35, 46, 58, 72, 91]

Test Case 2:
Array after partition with pivot 27: [23, 11, 21, 15, 27, 35, 28, 31]
Array after partition with pivot 11: [11, 15, 21, 23, 27, 35, 28, 31]
Array after partition with pivot 21: [11, 15, 21, 23, 27, 35, 28, 31]
Array after partition with pivot 28: [11, 15, 21, 23, 27, 28, 31, 35]
Array after partition with pivot 31: [11, 15, 21, 23, 27, 28, 31, 35]
Sorted Array: [11, 15, 21, 23, 27, 28, 31, 35]

Test Case 3:
Array after partition with pivot 43: [22, 34, 25, 36, 17, 13, 43, 67, 65, 52]
Array after partition with pivot 25: [22, 13, 17, 25, 34, 36, 43, 67, 65, 52]
Array after partition with pivot 13: [13, 17, 22, 25, 34, 36, 43, 67, 65, 52]
Array after partition with pivot 17: [13, 17, 22, 25, 34, 36, 43, 67, 65, 52]
Array after partition with pivot 34: [13, 17, 22, 25, 34, 36, 43, 67, 65, 52]
Array after partition with pivot 65: [13, 17, 22, 25, 34, 36, 43, 52, 65, 67]
Sorted Array: [13, 17, 22, 25, 34, 36, 43, 52, 65, 67]
```

7. 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.

AIM:

To implement the Binary Search algorithm to find the index of a given element in a sorted array and count the number of comparisons made during the search.

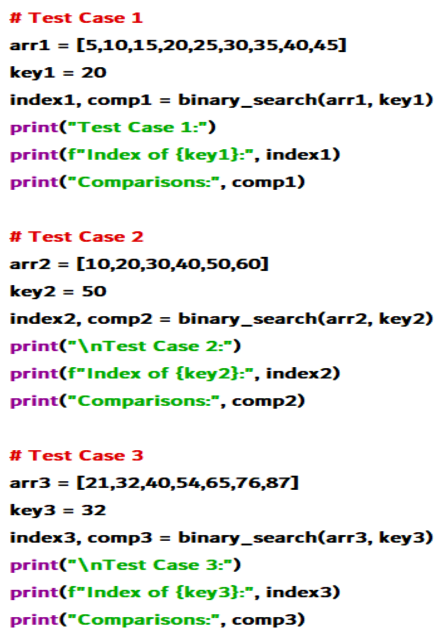
ALGORITHM:

1. Start
2. Read a sorted array a[] of size N and the search key
3. Initialize low = 0, high = N-1, and comparisons = 0
4. While low <= high:
    - Increment comparisons
    - Find mid = (low + high) // 2
    - If a[mid] == key, return mid
    - Else if a[mid] < key, set low = mid + 1
    - Else, set high = mid - 1
5. If key is not found, return -1
6. Stop

CODE:

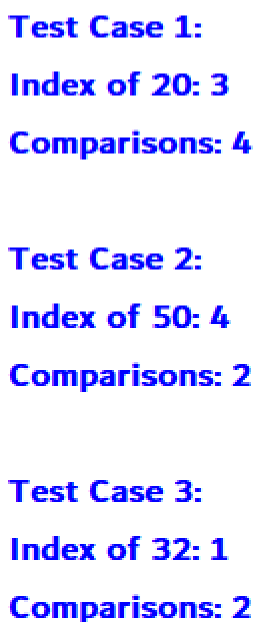
```python
def binary_search(arr, key):
    low = 0
    high = len(arr) - 1
    comparisons = 0

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

INPUT:



```
# Test Case 1
arr1 = [5,10,15,20,25,30,35,40,45]
key1 = 20
index1, comp1 = binary_search(arr1, key1)
print("Test Case 1:")
print(f"Index of {key1}:", index1)
print("Comparisons:", comp1)

# Test Case 2
arr2 = [10,20,30,40,50,60]
key2 = 50
index2, comp2 = binary_search(arr2, key2)
print("\nTest Case 2:")
print(f"Index of {key2}:", index2)
print("Comparisons:", comp2)

# Test Case 3
arr3 = [21,32,40,54,65,76,87]
key3 = 32
index3, comp3 = binary_search(arr3, key3)
print("\nTest Case 3:")
print(f"Index of {key3}:", index3)
print("Comparisons:", comp3)
```

OUTPUT:



```
Test Case 1:
Index of 20: 3
Comparisons: 4

Test Case 2:
Index of 50: 4
Comparisons: 2

Test Case 3:
Index of 32: 1
Comparisons: 2
```

8. 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?

AIM:

To implement Binary Search on a sorted array, trace the midpoint calculations, explain the steps involved in finding an element, and discuss the impact of searching on an unsorted array.

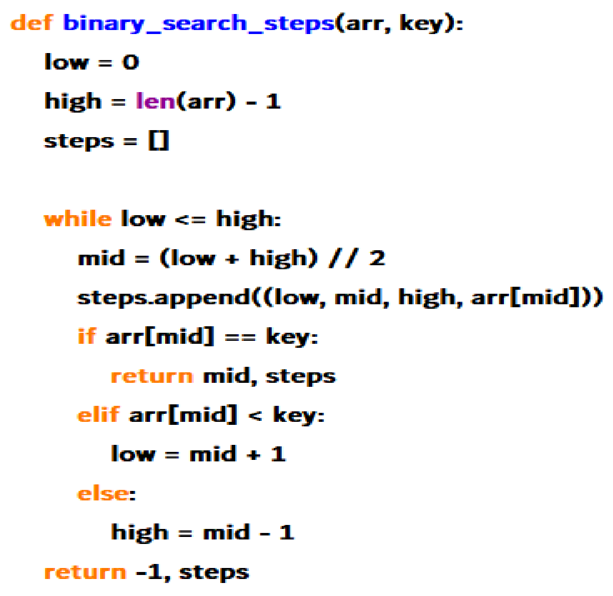
ALGORITHM:

1. Start
2. Read a sorted array a[] of size N and search key
3. Initialize low = 0, high = N-1
4. While low <= high:
   - Compute mid = (low + high) // 2
   - If a[mid] == key, return mid
   - Else if a[mid] < key, set low = mid + 1
   - Else, set high = mid - 1

   - Print the current low, mid, high and a[mid]

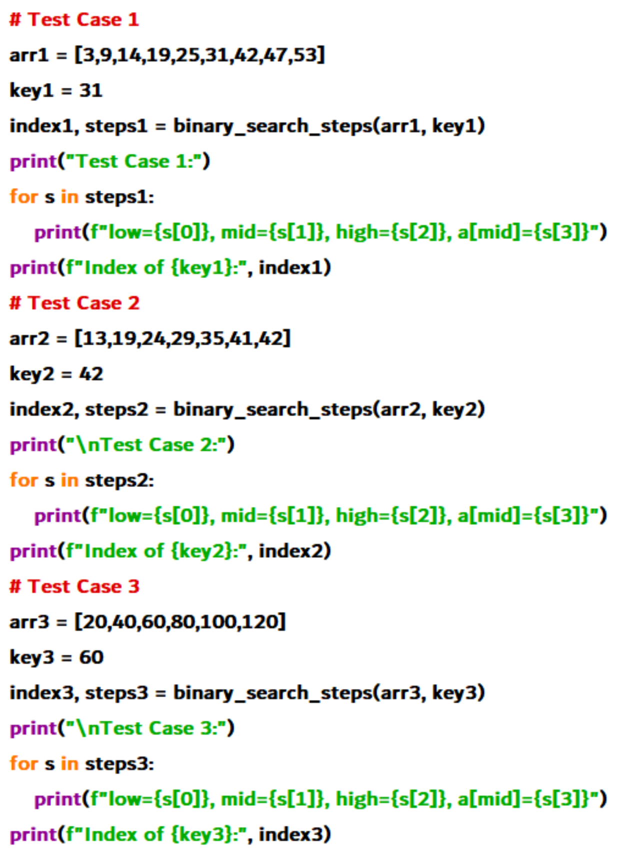
5. If key not found, return -1
6. Stop

CODE:



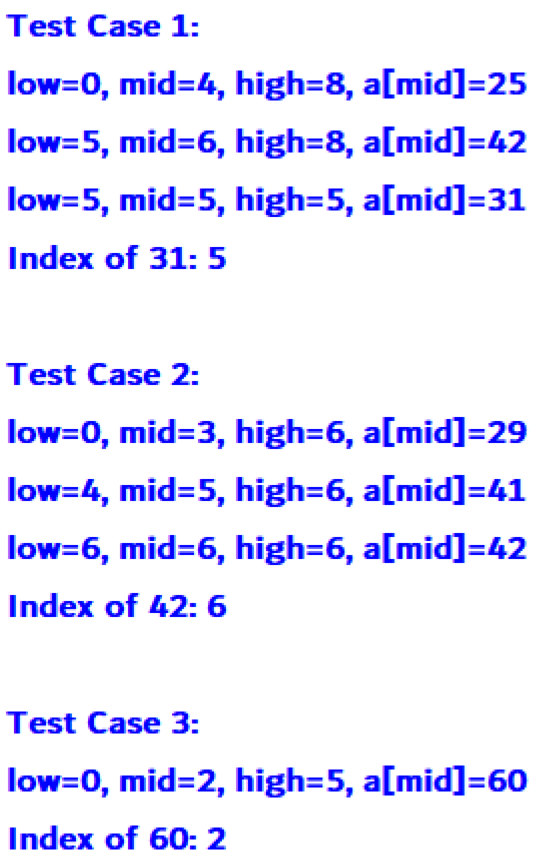
```
def binary_search_steps(arr, key):
    low = 0
    high = len(arr) - 1
    steps = []

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

INPUT:



```
# Test Case 1
arr1 = [3,9,14,19,25,31,42,47,53]
key1 = 31
index1, steps1 = binary_search_steps(arr1, key1)
print("Test Case 1:")
for s in steps1:
    print(f"low={s[0]}, mid={s[1]}, high={s[2]}, a[mid]={s[3]}")
print(f"Index of {key1}:", index1)
# Test Case 2
arr2 = [13,19,24,29,35,41,42]
key2 = 42
index2, steps2 = binary_search_steps(arr2, key2)
print("\nTest Case 2:")
for s in steps2:
    print(f"low={s[0]}, mid={s[1]}, high={s[2]}, a[mid]={s[3]}")
print(f"Index of {key2}:", index2)
# Test Case 3
arr3 = [20,40,60,80,100,120]
key3 = 60
index3, steps3 = binary_search_steps(arr3, key3)
print("\nTest Case 3:")
for s in steps3:
    print(f"low={s[0]}, mid={s[1]}, high={s[2]}, a[mid]={s[3]}")
print(f"Index of {key3}:", index3)
```

OUTPUT:



```
Test Case 1:
low=0, mid=4, high=8, a[mid]=25
low=5, mid=6, high=8, a[mid]=42
low=5, mid=5, high=5, a[mid]=31
Index of 31: 5

Test Case 2:
low=0, mid=3, high=6, a[mid]=29
low=4, mid=5, high=6, a[mid]=41
low=6, mid=6, high=6, a[mid]=42
Index of 42: 6

Test Case 3:
low=0, mid=2, high=5, a[mid]=60
Index of 60: 2
```

9. Given an array of points where points[i] = [xi, yi] represents a point on the X-Y plane and an integer k, return the k closest points to the origin (0, 0).

AIM:

To find the k closest points to the origin (0,0) from a given list of points on the X-Y plane using the Euclidean distance.

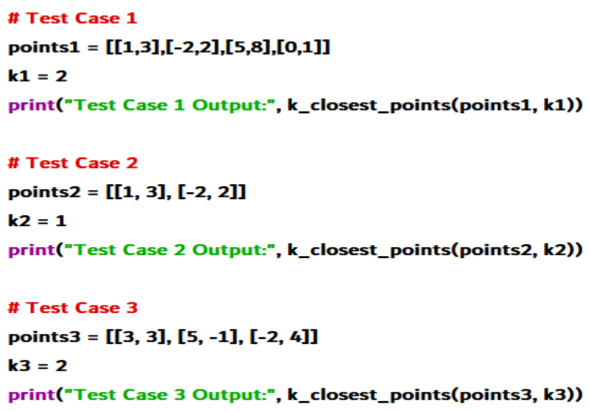
ALGORITHM:

1. Start
2. Read the list of points points[] and integer k
3. For each point [x, y], calculate the squared distance from the origin: distance = x^2 + y^2
4. Sort the points based on their distance from the origin
5. Select the first k points from the sorted list
6. Return these k points as the result
7. Stop

CODE:

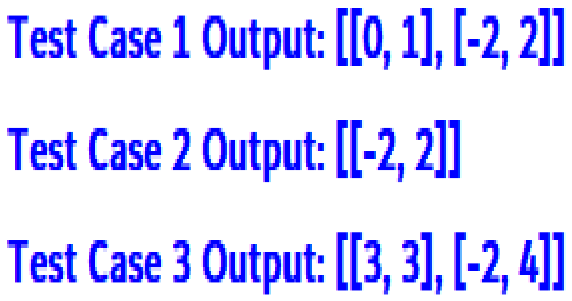
```python
def k_closest_points(points, k):
    # Sort points based on squared distance from origin
    points.sort(key=lambda point: point[0]**2 + point[1]**2)
    return points[:k]
```

INPUT:



```python
# Test Case 1
points1 = [[1,3],[-2,2],[5,8],[0,1]]
k1 = 2
print("Test Case 1 Output:", k_closest_points(points1, k1))

# Test Case 2
points2 = [[1, 3], [-2, 2]]
k2 = 1
print("Test Case 2 Output:", k_closest_points(points2, k2))

# Test Case 3
points3 = [[3, 3], [5, -1], [-2, 4]]
k3 = 2
print("Test Case 3 Output:", k_closest_points(points3, k3))
```

OUTPUT:



```
Test Case 1 Output: [[0, 1], [-2, 2]]

Test Case 2 Output: [[-2, 2]]

Test Case 3 Output: [[3, 3], [-2, 4]]
```

10. Given four lists A, B, C, D of integer values,Write a program to compute how many tuples n(i, j, k, l) there are such that A[i] + B[j] + C[k] + D[l] is zero.

AIM:

To count the number of tuples (i, j, k, l) such that A[i] + B[j] + C[k] + D[l] = 0 given four lists of integers.

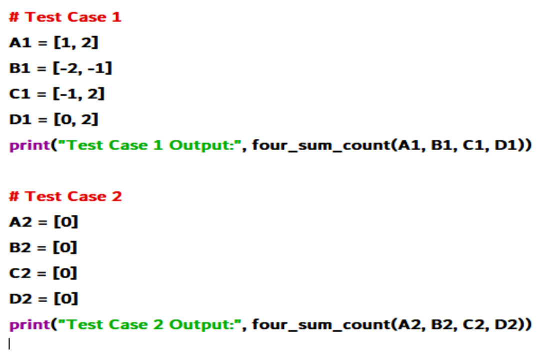
ALGORITHM:

1. Start
2. Read four lists A, B, C, D
3. Create a dictionary sum_ab to store sums of pairs from A and B and their frequency
4. For each a in A and each b in B:
    - Compute s = a + b
    - Increment sum_ab[s] by 1
5. Initialize count = 0
6. For each c in C and each d in D:
    - Compute target = -(c + d)
    - If target exists in sum_ab, increment count by sum_ab[target]
7. Return count
8. Stop

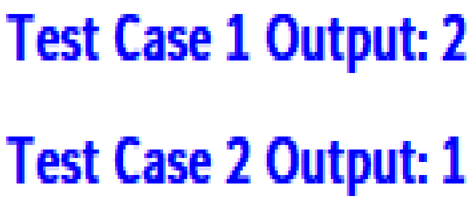
CODE:

```python
from collections import defaultdict

def four_sum_count(A, B, C, D):
    sum_ab = defaultdict(int)
    for a in A:
        for b in B:
            sum_ab[a + b] += 1

    count = 0
    for c in C:
        for d in D:
            target = -(c + d)
            if target in sum_ab:
                count += sum_ab[target]
    return count
```

INPUT: **OUTPUT:**



```
# Test Case 1
A1 = [1, 2]
B1 = [-2, -1]
C1 = [-1, 2]
D1 = [0, 2]
print("Test Case 1 Output:", four_sum_count(A1, B1, C1, D1))

# Test Case 2
A2 = [0]
B2 = [0]
C2 = [0]
D2 = [0]
print("Test Case 2 Output:", four_sum_count(A2, B2, C2, D2))
```



```
Test Case 1 Output: 2

Test Case 2 Output: 1
```

11. 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.

AIM:

To implement the Median of Medians algorithm to efficiently find the k-th smallest element in an unsorted array, ensuring worst-case linear time complexity.

ALGORITHM:

1. Start
2. Read the unsorted array arr and integer k
3. If the array length is small (≤5), sort it and return the k-th smallest element
4. Divide the array into groups of 5 elements each
5. Find the median of each group
6. Recursively find the median of these medians; call this pivot p
7. Partition the array into three groups:
   - L = elements less than p
   - E = elements equal to p
   - G = elements greater than p
8. If k <= len(L), recurse on L
9. Else if k <= len(L) + len(E), return p
10. Else, recurse on G with adjusted k = k - len(L) - len(E)
11. Stop

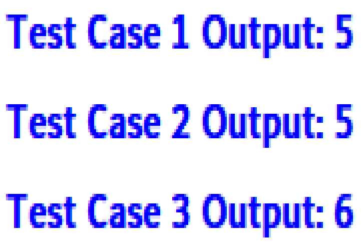
CODE:

```python
def median_of_medians(arr, k):
    if len(arr) <= 5:
        return sorted(arr)[k-1]

    # Divide arr into groups of 5
    groups = [arr[i:i+5] for i in range(0, len(arr), 5)]
    medians = [sorted(group)[len(group)//2] for group in groups]

    # Find pivot (median of medians)
    pivot = median_of_medians(medians, (len(medians)+1)//2)

    # Partition
    L = [x for x in arr if x < pivot]
    E = [x for x in arr if x == pivot]
    G = [x for x in arr if x > pivot]

    if k <= len(L):
        return median_of_medians(L, k)
    elif k <= len(L) + len(E):
        return pivot
    else:
        return median_of_medians(G, k - len(L) - len(E))
```

INPUT:

```python
# Test Case 1
arr1 = [12, 3, 5, 7, 19]
k1 = 2
print("Test Case 1 Output:", median_of_medians(arr1, k1))

# Test Case 2
arr2 = [12, 3, 5, 7, 4, 19, 26]
k2 = 3
print("Test Case 2 Output:", median_of_medians(arr2, k2))

# Test Case 3
arr3 = [1,2,3,4,5,6,7,8,9,10]
k3 = 6
print("Test Case 3 Output:", median_of_medians(arr3, k3))
```

OUTPUT:



```
Test Case 1 Output: 5

Test Case 2 Output: 5

Test Case 3 Output: 6
```

12. 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.

AIM:

To implement a function median_of_medians(arr, k) that finds the k-th smallest element in an unsorted array efficiently using the Median of Medians algorithm.

ALGORITHM:

1. Start
2. Read unsorted array arr and integer k
3. If the array length ≤ 5, sort it and return the k-th smallest element
4. Divide arr into groups of 5 elements
5. Find the median of each group
6. Recursively find the median of medians; this is the pivot p
7. Partition the array into three lists:
    - L = elements less than p
    - E = elements equal to p
    - G = elements greater than p
8. If k ≤ len(L), recurse on L
9. Else if k ≤ len(L) + len(E), return p
10. Else, recurse on G with k - len(L) - len(E)
11. Stop

CODE:

```python
def median_of_medians(arr, k):
    if len(arr) <= 5:
        return sorted(arr)[k-1]

    # Divide into groups of 5 and find medians
    groups = [arr[i:i+5] for i in range(0, len(arr), 5)]
    medians = [sorted(group)[len(group)//2] for group in groups]

    # Pivot is median of medians
    pivot = median_of_medians(medians, (len(medians)+1)//2)

    # Partition
    L = [x for x in arr if x < pivot]
    E = [x for x in arr if x == pivot]
    G = [x for x in arr if x > pivot]

    if k <= len(L):
        return median_of_medians(L, k)
    elif k <= len(L) + len(E):
        return pivot
    else:
        return median_of_medians(G, k - len(L) - len(E))
```

INPUT:

```
# Test Case 1
arr1 = [1,2,3,4,5,6,7,8,9,10]
k1 = 6
print("Test Case 1 Output:", median_of_medians(arr1, k1))

# Test Case 2
arr2 = [23,17,31,44,55,21,20,18,19,27]
k2 = 5
print("Test Case 2 Output:", median_of_medians(arr2, k2))
```

OUTPUT:

```
Test Case 1 Output: 6
Test Case 2 Output: 21
```

13. 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.

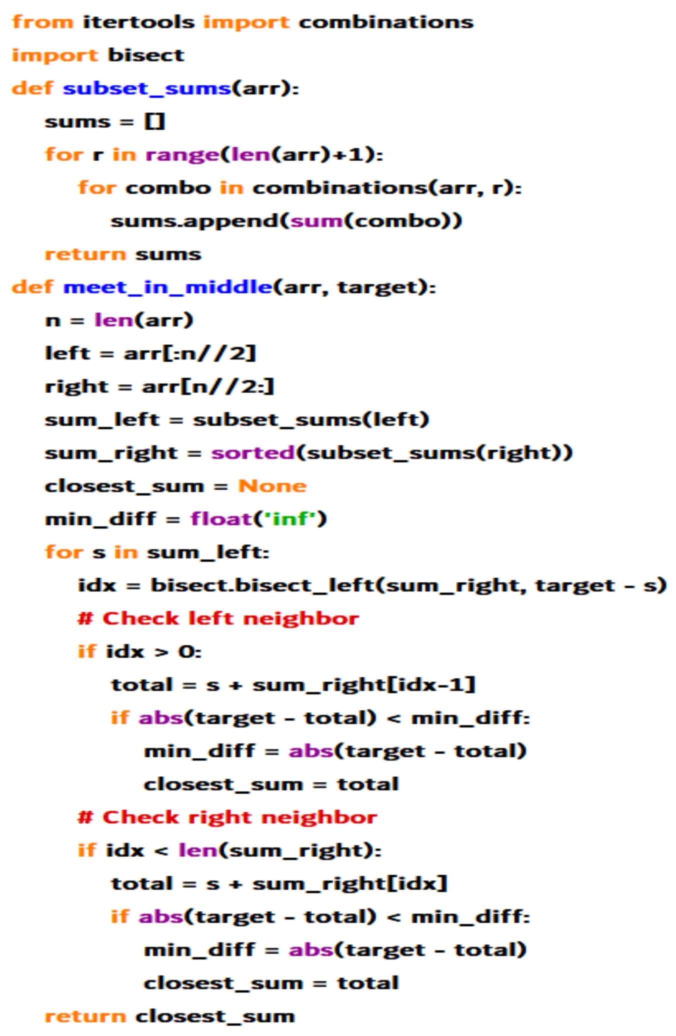
AIM:

To implement the Meet in the Middle technique to find a subset whose sum is closest to a given target sum efficiently.

ALGORITHM:

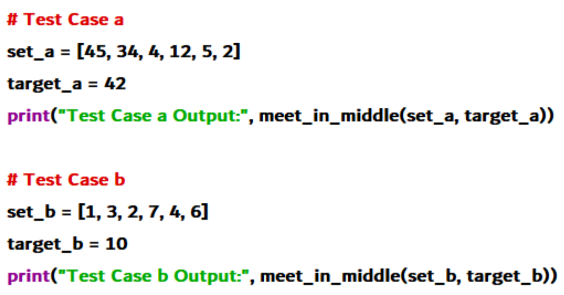
```
1. Start
2. Split the array arr into two halves: left and right
3. Generate all possible subset sums for each half (sum_left and sum_right)
4. Sort sum_right
5. Initialize closest_sum and min_diff
6. For each sum s in sum_left:
     o Find the value t in sum_right such that s + t is closest to the target using binary search
     o If abs(target - (s + t)) < min_diff, update closest_sum and min_diff
7. Return closest_sum
8. Stop
9.
```

CODE:



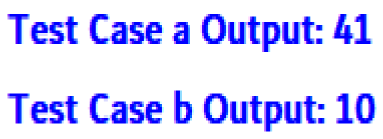
```python
from itertools import combinations
import bisect
def subset_sums(arr):
    sums = []
    for r in range(len(arr)+1):
        for combo in combinations(arr, r):
            sums.append(sum(combo))
    return sums
def meet_in_middle(arr, target):
    n = len(arr)
    left = arr[:n//2]
    right = arr[n//2:]
    sum_left = subset_sums(left)
    sum_right = sorted(subset_sums(right))
    closest_sum = None
    min_diff = float('inf')
    for s in sum_left:
        idx = bisect.bisect_left(sum_right, target - s)
        # Check left neighbor
        if idx > 0:
            total = s + sum_right[idx-1]
            if abs(target - total) < min_diff:
                min_diff = abs(target - total)
                closest_sum = total
        # Check right neighbor
        if idx < len(sum_right):
            total = s + sum_right[idx]
            if abs(target - total) < min_diff:
                min_diff = abs(target - total)
                closest_sum = total
    return closest_sum
```

INPUT:



```python
# Test Case a
set_a = [45, 34, 4, 12, 5, 2]
target_a = 42
print("Test Case a Output:", meet_in_middle(set_a, target_a))

# Test Case b
set_b = [1, 3, 2, 7, 4, 6]
target_b = 10
print("Test Case b Output:", meet_in_middle(set_b, target_b))
```

OUTPUT:



```
Test Case a Output: 41

Test Case b Output: 10
```

14. 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.

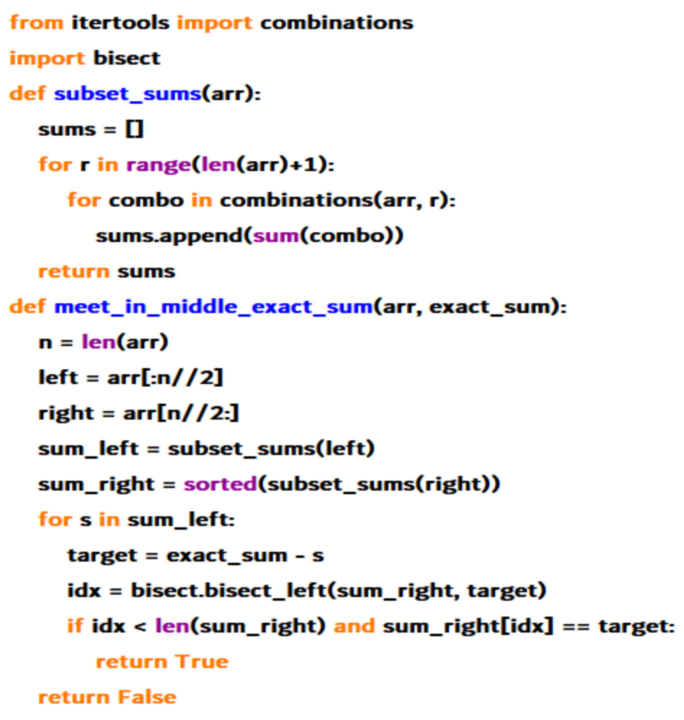
AIM:

To implement the Meet in the Middle technique to determine if there exists a subset of a large array whose sum equals a given exact sum.

ALGORITHM:

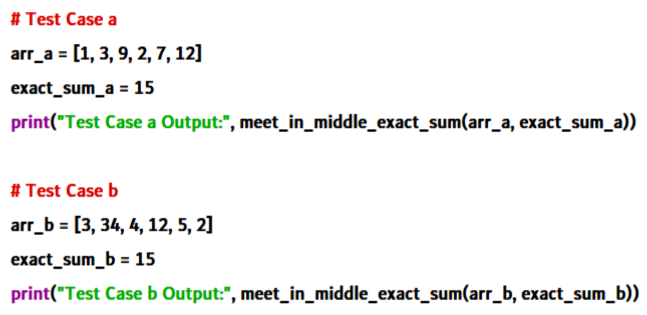
1. Start
2. Split the array arr into two halves: left and right
3. Generate all possible subset sums for each half (sum_left and sum_right)
4. Sort sum_right for efficient lookup
5. For each sum s in sum_left:
    - Use binary search to check if (exact_sum - s) exists in sum_right
    - If found, return True
6. If no combination produces the exact sum, return False
7. Stop

CODE:



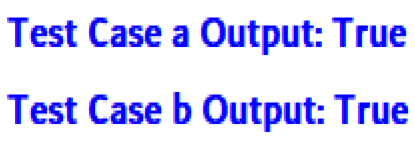
```
from itertools import combinations
import bisect
def subset_sums(arr):
    sums = []
    for r in range(len(arr)+1):
        for combo in combinations(arr, r):
            sums.append(sum(combo))
    return sums
def meet_in_middle_exact_sum(arr, exact_sum):
    n = len(arr)
    left = arr[:n//2]
    right = arr[n//2:]
    sum_left = subset_sums(left)
    sum_right = sorted(subset_sums(right))
    for s in sum_left:
        target = exact_sum - s
        idx = bisect.bisect_left(sum_right, target)
        if idx < len(sum_right) and sum_right[idx] == target:
            return True
    return False
```

INPUT:



```
# Test Case a
arr_a = [1, 3, 9, 2, 7, 12]
exact_sum_a = 15
print("Test Case a Output:", meet_in_middle_exact_sum(arr_a, exact_sum_a))

# Test Case b
arr_b = [3, 34, 4, 12, 5, 2]
exact_sum_b = 15
print("Test Case b Output:", meet_in_middle_exact_sum(arr_b, exact_sum_b))
```

OUTPUT:



```
Test Case a Output: True
Test Case b Output: True
```

15. Given two 2×2 Matrices A and B: A=(1 7 3 5) B=(1 3 7 5). Use Strassen's matrix multiplication algorithm to compute the product matrix

AIM:

To implement Strassen's Matrix Multiplication algorithm to multiply two 2×2 matrices efficiently.

ALGORITHM:

1. Read two 2×2 matrices A and B.
2. Split both matrices into four parts.
3. Compute the 7 products using Strassen's formula:

   $M1 = (A11 + A22)(B11 + B22)$

   $M2 = (A21 + A22)B11$

   $M3 = A11(B12 - B22)$

   $M4 = A22(B21 - B11)$

   $M5 = (A11 + A12)B22$

   $M6 = (A21 - A11)(B11 + B12)$

   $M7 = (A12 - A22)(B21 + B22)$

4. Compute result matrix elements:

   $C11=M1+M4-M5+M7$

   $C12=M3+M5$

   $C21=M2+M4$

   $C22 = M1 - M2 + M3 + M6$

5. Display the resultant matrix.

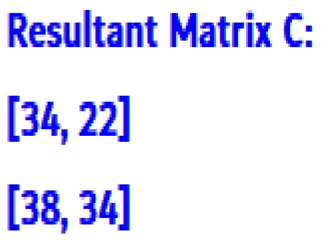
CODE:

```
def strassen_2x2(A, B):
    A11, A12 = A[0][0], A[0][1]
    A21, A22 = A[1][0], A[1][1]

    B11, B12 = B[0][0], B[0][1]
    B21, B22 = B[1][0], B[1][1]

    M1 = (A11 + A22) * (B11 + B22)
    M2 = (A21 + A22) * B11
    M3 = A11 * (B12 - B22)
    M4 = A22 * (B21 - B11)
    M5 = (A11 + A12) * B22
    M6 = (A21 - A11) * (B11 + B12)
    M7 = (A12 - A22) * (B21 + B22)

    C11 = M1 + M4 - M5 + M7
    C12 = M3 + M5
    C21 = M2 + M4
    C22 = M1 - M2 + M3 + M6

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

    return C
```

INPUT:

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

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

result = strassen_2x2(A, B)

print("Resultant Matrix C:")
for row in result:
    print(row)
```

OUTPUT:



```
Resultant Matrix C:
[34, 22]
[38, 34]
```