TOPIC 3 - DIVIDE & CONQUER

Q1. Merge Sort

Aim:
To sort an array using divide-and-conquer (Merge Sort) technique.
Algorithm:

1. Divide array into halves.
2. Recursively sort each half.
3. Merge two sorted halves.
4. Compare elements while merging.
5. Return merged sorted array.

Code:

```python
# merge_sort_with_count.py
def merge_sort(arr):
    comparisons = 0

    def merge(left, right):
        nonlocal comparisons
        i = j = 0
        merged = []
        while i < len(left) and j < len(right):
            comparisons += 1
            if left[i] <= right[j]:
                merged.append(left[i]); i += 1
            else:
                merged.append(right[j]); j += 1
        merged.extend(left[i:])
        merged.extend(right[j:])
        return merged

    def sort(a):
        if len(a) <= 1:
            return a
        mid = len(a) // 2
        L = sort(a[:mid])
        R = sort(a[mid:])
        return merge(L, R)

    sorted_arr = sort(arr)
```

Input: `[12, 4, 78, 23, 45, 67, 89, 1]`
Output:

```
PS E:\studies\DAA\daa lab> & "C:/Program Files/Pytho
Input: [12, 4, 78, 23, 45, 67, 89, 1]
Sorted: [1, 4, 12, 23, 45, 67, 78, 89]
Comparisons: 16
PS E:\studies\DAA\daa lab>
```

Result: Array sorted successfully via merge sort.

Q2. Quick Sort

Aim:
To sort numbers using quick sort by selecting a pivot and partitioning recursively.
Algorithm:

1. Choose pivot (first or middle element).
2. Partition array into < pivot and > pivot groups.
3. Recursively sort subarrays.
4. Combine results.
5. Return sorted list.

Code:

```python
# quick_sort_first_pivot.py
def partition_first(a, low, high):
    pivot = a[low]
    i = low + 1
    j = high
    while True:
        while i <= j and a[i] <= pivot:
            i += 1
        while i <= j and a[j] >= pivot:
            j -= 1
        if i <= j:
            a[i], a[j] = a[j], a[i]
        else:
            break
    a[low], a[j] = a[j], a[low]
    return j

def quick_sort_first(a, low=0, high=None, record_partitions=None):
    if high is None:
        high = len(a) - 1
    if record_partitions is None:
        record_partitions = []
    if low < high:
        p = partition_first(a, low, high)
        # record snapshot after partition
        record_partitions.append((low, high, p, a.copy()))
        quick_sort_first(a, low, p - 1, record_partitions)
        quick_sort_first(a, p + 1, high, record_partitions)
    return record_partitions

if __name__ == "__main__":
    arr = [10,16,8,12,15,6,3,9,5]
    parts = quick_sort_first(arr.copy())
    print("Snapshots after partitions (low,high,pivot_index,array_snapshot):")
    for s in parts:
        print(s)
    # final sorted:
    arr2 = arr.copy()
    # run full quicksort for sorted output
    quick_sort_first(arr2)
```

Input: [10,16,8,12,15,6,3,9,5]

Output:

```
PS E:\studies\DAA\daa lab> & "C:/Program Files/Python313/python.exe" "e:/
Snapshots after partitions (low,high,pivot_index,array_snapshot):
(0, 8, 5, [6, 5, 8, 9, 3, 10, 15, 12, 16])
(0, 4, 2, [3, 5, 6, 9, 8, 10, 15, 12, 16])
(0, 1, 0, [3, 5, 6, 9, 8, 10, 15, 12, 16])
(3, 4, 4, [3, 5, 6, 8, 9, 10, 15, 12, 16])
(6, 8, 7, [3, 5, 6, 8, 9, 10, 12, 15, 16])
Final sorted: [3, 5, 6, 8, 9, 10, 12, 15, 16]
PS E:\studies\DAA\daa lab>
```

Result: Array sorted with quick sort logic.

Q3. Binary Search

Aim:
To find an element in a sorted array efficiently using binary search.
Algorithm:

1. Set `low=0`, `high=n-1`.
2. Find mid index.
3. Compare mid value with key.
4. Adjust range accordingly.
5. Continue until found or range ends.

Code:

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

if __name__ == "__main__":
    arr = [5,10,15,20,25,30,35,40,45]
    idx, comps = binary_search(arr, 20)
    print("Index:", idx, "Comparisons:", comps)
```

Input: `[5,10,15,20,25,30,35,40,45], key=20`

Output:

```
PS E:\studies\DAA\daa lab> & "C:/Program Files/Python313/python.e
Index: 3 Comparisons: 4
PS E:\studies\DAA\daa lab>
```

Result: Binary search found the element in O(log n).

Q4. Median of Medians

Aim:
To find the k-th smallest element in an unsorted array using the median-of-medians method.
Algorithm:

1. Divide array into groups of 5.
2. Sort each group and find median.
3. Find median of medians recursively.
4. Partition array around pivot.
5. Recurse in desired partition.

```python
# median_of_medians.py
import math

def partition(arr, pivot):
    L, E, G = [], [], []
    for x in arr:
        if x < pivot: L.append(x)
        elif x > pivot: G.append(x)
        else: E.append(x)
    return L, E, G

def select_kth(arr, k):  # k: 1-based (1 = smallest)
    if not arr:
        raise ValueError("Empty array")
    if len(arr) <= 5:
        return sorted(arr)[k-1]
    # split into groups of 5
    groups = [arr[i:i+5] for i in range(0, len(arr), 5)]
    medians = [sorted(g)[len(g)//2] for g in groups]
    pivot = select_kth(medians, (len(medians)+1)//2)
    L, E, G = partition(arr, pivot)
    if k <= len(L):
        return select_kth(L, k)
    elif k <= len(L) + len(E):
        return pivot
    else:
        return select_kth(G, k - len(L) - len(E))

if __name__ == "__main__":
    arr = [12, 3, 5, 7, 19]
    k = 2
    print(f"{k}-th smallest:", select_kth(arr, k))
```

Input: `[12,3,5,7,19], k=2`

Output:

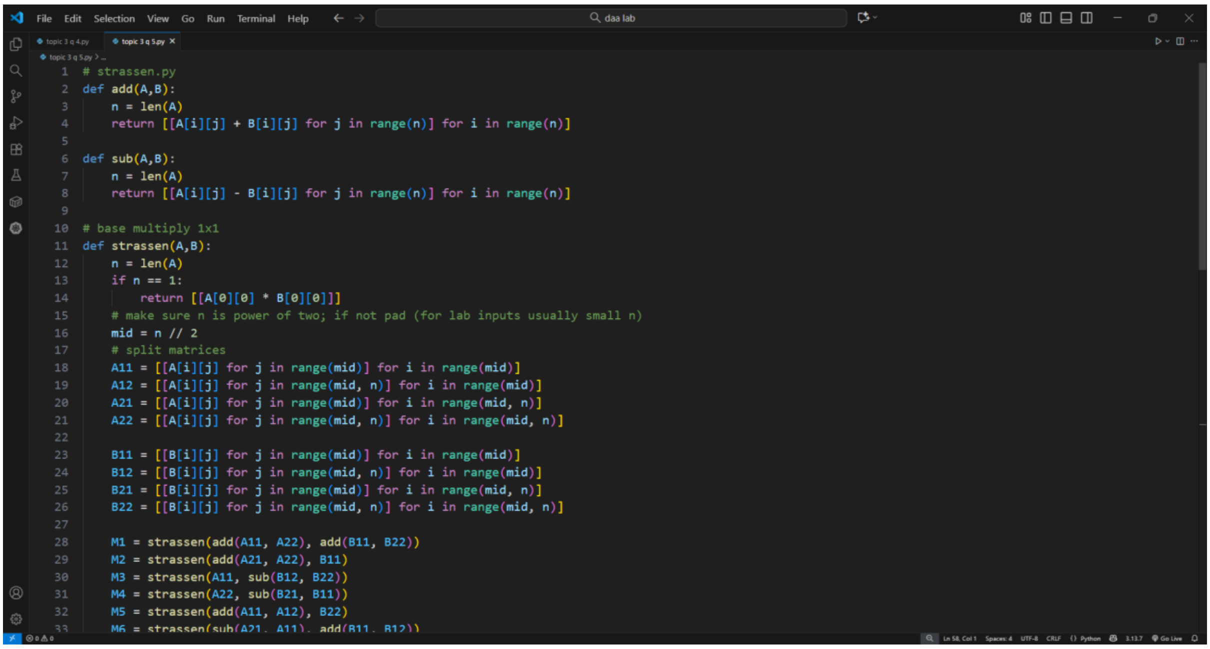
```
PS E:\studies\DAA\daa lab> code '.\topic 3 q 4.py'
PS E:\studies\DAA\daa lab> & "C:/Program Files/Python313/python.exe" "e:
2-th smallest: 5
PS E:\studies\DAA\daa lab>
```

Result: 2nd smallest element correctly found.

Q5. Strassen's Matrix Multiplication

Aim:
To multiply two matrices efficiently using Strassen's algorithm.
Algorithm:

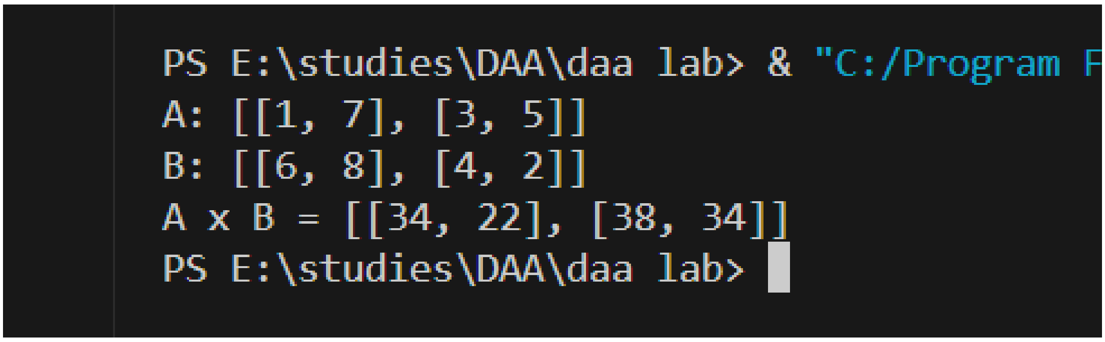
1. Split matrices into four sub-matrices.
2. Compute 7 products recursively.
3. Combine them using Strassen's formulas.
4. Form resultant matrix.
5. Display output.



```python
# strassen.py
def add(A,B):
    n = len(A)
    return [[A[i][j] + B[i][j] for j in range(n)] for i in range(n)]

def sub(A,B):
    n = len(A)
    return [[A[i][j] - B[i][j] for j in range(n)] for i in range(n)]

# base multiply 1x1
def strassen(A,B):
    n = len(A)
    if n == 1:
        return [[A[0][0] * B[0][0]]]
    # make sure n is power of two; if not pad (for lab inputs usually small n)
    mid = n // 2
    # split matrices
    A11 = [[A[i][j] for j in range(mid)] for i in range(mid)]
    A12 = [[A[i][j] for j in range(mid, n)] for i in range(mid)]
    A21 = [[A[i][j] for j in range(mid)] for i in range(mid, n)]
    A22 = [[A[i][j] for j in range(mid, n)] for i in range(mid, n)]

    B11 = [[B[i][j] for j in range(mid)] for i in range(mid)]
    B12 = [[B[i][j] for j in range(mid, n)] for i in range(mid)]
    B21 = [[B[i][j] for j in range(mid)] for i in range(mid, n)]
    B22 = [[B[i][j] for j in range(mid, n)] for i in range(mid, n)]

    M1 = strassen(add(A11, A22), add(B11, B22))
    M2 = strassen(add(A21, A22), B11)
    M3 = strassen(A11, sub(B12, B22))
    M4 = strassen(A22, sub(B21, B11))
    M5 = strassen(add(A11, A12), B22)
    M6 = strassen(sub(A21, A11), add(B11, B12))
```

Input:
A = [[1,7],[3,5]], B = [[6,8],[4,2]]

Output:



```
PS E:\studies\DAA\daa lab> & "C:/Program F
A: [[1, 7], [3, 5]]
B: [[6, 8], [4, 2]]
A x B = [[34, 22], [38, 34]]
PS E:\studies\DAA\daa lab>
```

Result: Matrix multiplication done efficiently.