

# Session 5

## Divide and Conquer Technique

In Divide and conquer approach, the original problem is divided into two or more sub-problems recursively, till it is small enough to be solved easily. Each sub-problem is some fraction of the original problem. Next, the solutions of the sub-problems are combined together to generate the solution of the original problem.

## Objectives

✗ Do not copy. Read for understanding and the viva

- Understand how divide and conquer splits a problem into smaller independent subproblems.
- Implement recursive binary search, merge sort, quick sort, and Strassen's matrix multiplication.
- Show dry runs, recursive call trees, loop counts, comparisons, exchanges, and operation counts where the lab manual asks for them.

## Questions Covered

| Question | Requirement                                                                                  | Completion Notes                                              |
|----------|----------------------------------------------------------------------------------------------|---------------------------------------------------------------|
| Q1       | Recursive binary search for 100 in a sorted array                                            | Add implementation, trace, and recursive call tree.           |
| Q2       | Maximum searches among 512 million items using binary search                                 | Add logarithmic calculation.                                  |
| Q3       | Merge sort for <code>200 150 50 100 75 25 10 5</code>                                        | Add split/merge steps and recursion tree.                     |
| Q4       | Quick sort for <code>12 20 22 16 25 18 8 10 6 15</code>                                      | Add partition steps and output.                               |
| Q5       | Quick sort performance for sorted list <code>6 8 10 12 15 16 18 20 22 25</code>              | Add comparisons, exchanges, and loop iterations.              |
| Q6       | Strassen's multiplication for <code>n x n</code> matrices where <code>n</code> is power of 2 | Add multiple instances and compare multiplications/additions. |

## Question 1

### Problem Statement

■ Write in lab record

Implement a recursive binary search algorithm on your system to search for number `100` in the following array of integers. Show the process step by step and draw recursive calls.

1 10 35 40 45 50 55 60 65 70 100

### Approach

✗ Do not copy. Read for understanding and the viva

Binary search compares the target with the middle element. If the target is larger, the search continues in the right half; if smaller, it continues in the left half. Because the array is sorted, every recursive call reduces the search range by half.

## Step-by-Step Trace

■ Write in lab record

| Call | Low | High | Mid | A[mid] | Decision                                     |
|------|-----|------|-----|--------|----------------------------------------------|
| 1    | 0   | 9    | 4   | 50     | <code>100 &gt; 50</code> , search right half |
| 2    | 5   | 9    | 7   | 65     | <code>100 &gt; 65</code> , search right half |
| 3    | 8   | 9    | 8   | 70     | <code>100 &gt; 70</code> , search right half |
| 4    | 9   | 9    | 9   | 100    | Found                                        |

## Recursive Call Tree

```
1  binarySearch(0, 9)
2    → binarySearch(5, 9)
3      → binarySearch(8, 9)
4        → binarySearch(9, 9)
5          → found at index 9
```

## Complexity

✗ Do not copy. Read for understanding and the viva

- Time complexity: `O(log n)`
- Space complexity: `O(log n)` due to recursive call stack

## Implementation

■ Write in lab record

■

Lab record: write one language only. Pick yours once and every page opens on it; the other tabs are the same solution for comparison.

**Python**

## binary-search.py

```
1 def binary_search(arr, low, high, target):
2     """
3     Recursive Binary Search in Python
4     :param arr: Sorted list of integers
5     :param low: Starting index
6     :param high: Ending index
7     :param target: Value to find
8     :return: Index of target, or -1 if not found
9     """
10    # Base case: search range is invalid
11    if low > high:
12        return -1
13
14    # Integer division for mid calculation
15    mid = low + (high - low) // 2
16
17    # Step-by-step trace for lab submission
18    print(f"[Call] low={low}, high={high}, mid={mid}, arr[mid]={arr[mid]}")
19
20    # Base case: target found
21    if arr[mid] == target:
22        return mid
23
24    # Target is in the right half
25    elif arr[mid] < target:
26        return binary_search(arr, mid + 1, high, target)
27
28    # Target is in the left half
29    else:
30        return binary_search(arr, low, mid - 1, target)
31
32 # Driver code
33 if __name__ == "__main__":
34     arr = [10, 35, 40, 45, 50, 55, 60, 65, 70, 100]
35     target = 100
36     print("=== Recursive Binary Search in Python ===")
37
38     result = binary_search(arr, 0, len(arr) - 1, target)
39
40     if result != -1:
41         print(f"\n✅ Target {target} found at index {result}")
42     else:
43         print(f"\n❌ Target {target} not found in array")
```

## Question 2

### Problem Statement

■ Write in lab record

Suppose we are required to search among 512 million items in a list using binary search. What is the maximum number of searches needed to find a given item or conclude that it is not present?

### Solution

■ Write in lab record

Binary search needs at most  $\text{ceil}(\log_2 n)$  comparisons.

```
1 n = 512,000,000
2  $\log_2(512,000,000)$  is slightly less than 29
3 Maximum searches = 29
```

JS

If the list is interpreted as exactly  $512 * 2^{20} = 536,870,912$ , then:

```
1  $536,870,912 = 2^{29}$ 
2 Maximum searches = 29
```

## Question 3

### Problem Statement

■ Write in lab record

Implement Merge Sort algorithm to sort the following list and show the process step by step. Draw a tree of recursive calls.

```
1 200 150 50 100 75 25 10 5
```

## Approach

✖ Do not copy. Read for understanding and the viva

Merge sort divides the list into two halves until each sublist has one element, then merges the sorted sublists.

## Recursive Split Tree

✖ Do not copy. Read for understanding and the viva

```

1  [200,150,50,100,75,25,10,5]
2  └─ [200,150,50,100]
3  │   └─ [200,150]
4  │   │   └─ [200]
5  │   │   └─ [150]
6  │   └─ [50,100]
7  │       └─ [50]
8  │       └─ [100]
9  └─ [75,25,10,5]
10     └─ [75,25]
11     │   └─ [75]
12     │   └─ [25]
13     └─ [10,5]
14         └─ [10]
15         └─ [5]
```

## Implementation

■ Write in lab record

■

Lab record: write one language only. Pick yours once and every page opens on it; the other tabs are the same solution for comparison.

### Python

**merge-sort.py**

```
1 def merge_sort(arr, left=0, right=None, depth=0):
2     """
3     Recursive Merge Sort Implementation
4     :param arr: List to sort
5     :param left: Left index
6     :param right: Right index
7     :param depth: Recursion depth for indentation
8     :return: Sorted list
9     """
10    if right is None:
11        right = len(arr) - 1
12
13    # Print current state with indentation
14    indent = " " * depth
15    print(f"{indent}mergeSort({left},{right}) → {arr[left:right+1]}")
16
17    if left < right:
18        # Calculate mid point (avoid overflow)
19        mid = left + (right - left) // 2
20
21        # Recursively sort first half
22        merge_sort(arr, left, mid, depth + 1)
23
24        # Recursively sort second half
25        merge_sort(arr, mid + 1, right, depth + 1)
26
27        # Merge the sorted halves
28        merge(arr, left, mid, right, depth)
29    else:
30        print(f"{indent} Base case: single element")
31
32    return arr
33
34 def merge(arr, left, mid, right, depth):
35     """
36     Merge two sorted subarrays
37     :param arr: Original array
38     :param left: Start of left subarray
39     :param mid: End of left subarray
40     :param right: End of right subarray
41     :param depth: Indentation level
42     """
43     indent = " " * depth
```

```
44
45     # Create temporary arrays
46     left_arr = arr[left:mid + 1]
47     right_arr = arr[mid + 1:right + 1]
48
49     print(f"{indent} Merge: Left={left_arr} Right={right_arr}")
50
51     i = j = 0 # Initial indices for left and right subarrays
52     k = left  # Initial index of merged subarray
53
54     # Merge the temp arrays back
55     while i < len(left_arr) and j < len(right_arr):
56         if left_arr[i] ≤ right_arr[j]:
57             arr[k] = left_arr[i]
58             print(f"{indent} Compare {left_arr[i]} ≤ {right_arr[j]} → Take left")
59             i += 1
60         else:
61             arr[k] = right_arr[j]
62             print(f"{indent} Compare {left_arr[i]} > {right_arr[j]} → Take right")
63             j += 1
64         k += 1
65
66     # Copy remaining elements
67     while i < len(left_arr):
68         arr[k] = left_arr[i]
69         print(f"{indent} Append remaining left: {left_arr[i]}")
70         i += 1
71         k += 1
72
73     while j < len(right_arr):
74         arr[k] = right_arr[j]
75         print(f"{indent} Append remaining right: {right_arr[j]}")
76         j += 1
77         k += 1
78
79     print(f"{indent} Result: {arr[left:right+1]}\n")
80
81 # Driver code
82 if __name__ == "__main__":
83     arr = [200, 150, 50, 100, 75, 25, 10, 5]
84
85     print("=== Merge Sort Implementation in Python ===")
86     print(f"Original array: {arr}\n")
87
```

```
88     merge_sort(arr)
89
90     print(f"Sorted array: {arr}")
```

## Final Merge Result

```
1  5 10 25 50 75 100 150 200
```

## Question 4

### Problem Statement

■ Write in lab record

Implement Quick Sort's algorithm to sort the following list and show step-by-step processes.

```
1  12 20 22 16 25 18 8 10 6 15
```

### Approach

✗ Do not copy. Read for understanding and the viva

Choose a pivot, partition the list so smaller elements move before the pivot and larger elements move after it, then recursively sort the two partitions.

### Recursive Split Tree

✗ Do not copy. Read for understanding and the viva

Here, we are using array index for understanding how the loop will work for quicksort.

SH

```
1 quickSort(0,9) pivot=15
2 |
3 |─ quickSort(0,3) pivot=6
4 |  |─ quickSort(0,-1) → base case
5 |  |─ quickSort(1,3) pivot=12
6 |  |  |─ quickSort(1,2) pivot=10
7 |  |  |  |─ quickSort(1,1) → base case
8 |  |  |  |─ quickSort(3,2) → base case
9 |  |  |─ quickSort(4,3) → base case
10 |
11 |─ quickSort(5,9) pivot=25
12 |  |─ quickSort(5,8) pivot=16
13 |  |  |─ quickSort(5,4) → base case
14 |  |  |─ quickSort(6,8) pivot=18
15 |  |  |  |─ quickSort(6,5) → base case
16 |  |  |  |─ quickSort(7,8) pivot=20
17 |  |  |  |  |─ quickSort(7,6) → base case
18 |  |  |  |  |─ quickSort(8,7) → base case
19 |  |  |─ quickSort(10,9) → base case
```



Lab record: write one language only. Pick yours once and every page opens on it; the other tabs are the same solution for comparison.

Python

**quick-sort.py**

```
1 def partition(arr, low, high):
2     """
3     Lomuto Partition: Rearranges array so elements ≤ pivot are left,
4     and elements > pivot are right. Returns pivot's final index.
5     """
6     pivot = arr[high]
7     i = low - 1 # Boundary for smaller elements
8
9     print(f" Partitioning [{low}..{high}] with pivot={pivot}")
10
11    for j in range(low, high):
12        print(f"    Compare arr[{j}]= {arr[j]} with pivot={pivot} → ", end="")
13        if arr[j] ≤ pivot:
14            i += 1
15            if i ≠ j:
16                arr[i], arr[j] = arr[j], arr[i]
17                print(f"SWAP {arr[i]} ↔ {arr[j]}")
18            else:
19                print("No swap needed")
20        else:
21            print("Skip (greater)")
22
23    # Place pivot in correct sorted position
24    arr[i + 1], arr[high] = arr[high], arr[i + 1]
25    print(f" Final swap: place pivot at index {i + 1}")
26    return i + 1
27
28 def quick_sort(arr, low, high):
29     """Recursive Quick Sort"""
30     if low < high:
31         print(f"quickSort({low}, {high})")
32         pi = partition(arr, low, high)
33         quick_sort(arr, low, pi - 1) # Sort left partition
34         quick_sort(arr, pi + 1, high) # Sort right partition
35
36 if __name__ == "__main__":
37     arr = [12, 20, 22, 16, 25, 18, 8, 10, 6, 15]
38     print("≡ Quick Sort Implementation in Python ≡")
39     print(f"Original: {arr}\n")
40
41     quick_sort(arr, 0, len(arr) - 1)
42
43     print(f"\nSorted:    {arr}")
```

## Final Sorted Output

1 6 8 10 12 15 16 18 20 22 25

## Complexity

✕ Do not copy. Read for understanding and the viva

| Metric            | Value         | Explanation                                       |
|-------------------|---------------|---------------------------------------------------|
| Best Case Time    | $O(n \log n)$ | Pivot divides array evenly every time             |
| Average Case Time | $O(n \log n)$ | Random/pivot selection yields balanced partitions |
| Worst Case Time   | $O(n^2)$      | Already sorted/reverse sorted + last/first pivot  |
| Space Complexity  | $O(\log n)$   | Recursion stack depth (average)                   |
| Stable Sort       | No            | Relative order of equal elements may change       |
| In-Place          | Yes           | Sorts within original array (no extra arrays)     |

## Question 5

### Problem Statement

■ Write in lab record

Examine the performance of Quick Sort for the following list in terms of comparisons, exchange operations, and loop iterations.

1 6 8 10 12 15 16 18 20 22 25

## Note

This input is already sorted. If the implementation always chooses the first or last element as pivot, this becomes the worst-case pattern for Quick Sort.

| Metric          | Worst-Case Pattern       |
|-----------------|--------------------------|
| Time complexity | $O(n^2)$                 |
| Recursion depth | $n$                      |
| Comparisons     | Approximately $n(n-1)/2$ |

For  $n = 10$ , the worst-case comparison count is:

$$1 \cdot 10 * 9 / 2 = 45$$

## Performance Summary

× Do not copy. Read for understanding and the viva

| Metric                          | Count Formula/Explanation                                              |
|---------------------------------|------------------------------------------------------------------------|
| Total Comparisons               | 45                                                                     |
| Meaningful Swaps ( $i \neq j$ ) | 0                                                                      |
| Self-Swaps ( $i = j$ )          | 45 Each comparison where $arr[j] \leq pivot$ triggers swap with itself |
| Final Pivot Placements          | 9                                                                      |
| Total Loop Iterations           | 45                                                                     |
| Recursive Calls                 | 10                                                                     |
| Recursion Depth                 | 10                                                                     |

## Mathematical Verification

RS

```

1 For n = 10, sorted array, Lomuto partition:
2
3 Comparisons = (n-1) + (n-2) + ... + 1 + 0
4               =  $\sum(k)$  for k=0 to n-1
5               =  $n(n-1)/2$ 
6               =  $10 \times 9 / 2 = 45$  ✓
7
8 Swaps (meaningful) = 0 (since array is already partitioned)
9
10 Time Complexity =  $O(n^2)$  ← Worst Case
11 Space Complexity =  $O(n)$  ← Recursion stack depth

```

## Question 6

### Problem Statement

■ Write in lab record

Implement Strassen's multiplication algorithm for two  $n \times n$  matrices, where  $n$  is a power of 2, on different problem instances and compare all instances in terms of number of multiplications and additions required.

### Strassen's Algorithm

Traditional matrix multiplication performs 8 multiplications for two  $2 \times 2$  matrices. Strassen's method reduces this to 7 multiplications by using additional additions and subtractions.

RS

```

1 M1 = (A11 + A22) × (B11 + B22)
2 M2 = (A21 + A22) × B11
3 M3 = A11 × (B12 - B22)
4 M4 = A22 × (B21 - B11)
5 M5 = (A11 + A12) × B22
6 M6 = (A21 - A11) × (B11 + B12)
7 M7 = (A12 - A22) × (B21 + B22)

```

## Comparison

| Method                   | Multiplications | Additions/Subtractions      | Time Complexity |
|--------------------------|-----------------|-----------------------------|-----------------|
| Classical multiplication | 8 subproblems   | Fewer additions             | $O(n^3)$        |
| Strassen multiplication  | 7 subproblems   | More additions/subtractions | $O(n^{2.807})$  |

## Implementation

■ Write in lab record

■

Lab record: write one language only. Pick yours once and every page opens on it; the other tabs are the same solution for comparison.

## Python

matrix-multiplication.py

```
1 # 6.py
2 # Strassen's Matrix Multiplication with Deterministic Operation Counting
3 # Language: Python 3.x | Complexity:  $O(n^{2.807})$  mults,  $O(n^{2.807})$  adds
4 # Run: python3 6.py
5
6 class OpCounter:
7     """Tracks scalar multiplications and additions/subtractions."""
8     def __init__(self):
9         self.mults = 0
10        self.adds = 0
11
12 def add_mat(A, B, n, ctr):
13     """Element-wise addition:  $C = A + B$ . n is the actual size of A and B."""
14     C = [[A[i][j] + B[i][j] for j in range(n)] for i in range(n)]
15     ctr.adds += n * n
16     return C
17
18 def sub_mat(A, B, n, ctr):
19     """Element-wise subtraction:  $C = A - B$ . n is the actual size of A and B.
20     C = [[A[i][j] - B[i][j] for j in range(n)] for i in range(n)]
21     ctr.adds += n * n
22     return C
23
24 def strassen(A, B, n, ctr):
25     """Recursive Strassen multiplication. Base case: n=1."""
26     if n == 1:
27         ctr.mults += 1
28         return [[A[0][0] * B[0][0]]]
29
30     mid = n // 2
31
32     # Extract quadrants (all are mid x mid)
33     A11 = [row[:mid] for row in A[:mid]]
34     A12 = [row[mid:] for row in A[:mid]]
35     A21 = [row[:mid] for row in A[mid:]]
36     A22 = [row[mid:] for row in A[mid:]]
37     B11 = [row[:mid] for row in B[:mid]]
38     B12 = [row[mid:] for row in B[:mid]]
39     B21 = [row[:mid] for row in B[mid:]]
40     B22 = [row[mid:] for row in B[mid:]]
41
42     # 7 recursive products
43     # FIX: All add_mat/sub_mat calls now correctly use `mid`, not `n`
```

```

44     M1 = strassen(add_mat(A11, A22, mid, ctr), add_mat(B11, B22, mid, ctr),
45     M2 = strassen(add_mat(A21, A22, mid, ctr), B11, mid, ctr)
46     M3 = strassen(A11, sub_mat(B12, B22, mid, ctr), mid, ctr)
47     M4 = strassen(A22, sub_mat(B21, B11, mid, ctr), mid, ctr)
48     M5 = strassen(add_mat(A11, A12, mid, ctr), B22, mid, ctr)
49     M6 = strassen(sub_mat(A21, A11, mid, ctr), add_mat(B11, B12, mid, ctr),
50     M7 = strassen(sub_mat(A12, A22, mid, ctr), add_mat(B21, B22, mid, ctr),
51
52     # Combine quadrants into result
53     # FIX: All helper calls correctly use `mid`
54     C11 = add_mat(sub_mat(add_mat(M1, M4, mid, ctr), M5, mid, ctr), M7, mid,
55     C12 = add_mat(M3, M5, mid, ctr)
56     C21 = add_mat(M2, M4, mid, ctr)
57     C22 = add_mat(sub_mat(add_mat(M1, M3, mid, ctr), M2, mid, ctr), M6, mid,
58
59     # Merge into full n x n matrix
60     C = [[0]*n for _ in range(n)]
61     for i in range(mid):
62         for j in range(mid):
63             C[i][j] = C11[i][j]
64             C[i][j+mid] = C12[i][j]
65             C[i+mid][j] = C21[i][j]
66             C[i+mid][j+mid] = C22[i][j]
67     return C
68
69 def naive_multiply(A, B, n, ctr):
70     """Standard  $O(n^3)$  multiplication for baseline comparison."""
71     C = [[0]*n for _ in range(n)]
72     for i in range(n):
73         for j in range(n):
74             for k in range(n):
75                 ctr.mults += 1
76                 if k > 0: ctr.adds += 1
77                 C[i][j] += A[i][k] * B[k][j]
78     return C
79
80 def init_matrix(n):
81     """Deterministic initialization for reproducible grading."""
82     return [(i * n + j + 1) for j in range(n)] for i in range(n)
83
84 if __name__ == "__main__":
85     print(f'{'n':<4} | {'Strassen Mults':<14} | {'Strassen Adds':<13} | {'Na
86     print("-" * 85)
87

```

```
88     sizes = [2, 4, 8, 16, 32, 64]
89     for n in sizes:
90         A = init_matrix(n)
91         B = init_matrix(n)
92
93         ctr_str = OpCounter()
94         C_str = strassen(A, B, n, ctr_str)
95
96         ctr_naive = OpCounter()
97         C_naive = naive_multiply(A, B, n, ctr_naive)
98
99         print(f"{n:<4} | {ctr_str.mults:<14} | {ctr_str.adds:<13} | {ctr_nai
```

## Viva Questions

✖ Do not copy. Read for understanding and the viva

- Why must the input array be sorted for binary search?
- Why is merge sort stable?
- What causes Quick Sort's worst case?
- Why does Strassen's method reduce multiplication count?

 Edit this page

< Previous  
**Session 4**

Next >  
**Session 6**