Divide and Conquer Approach

Merge Sort, Quick Sort, Binary Search

Rashed Hassan Siam
Department of CSE
Eastern University

Recursion

- Recursion is a method of solving a problem where the solution depends on solutions to smaller instances of the same problem.

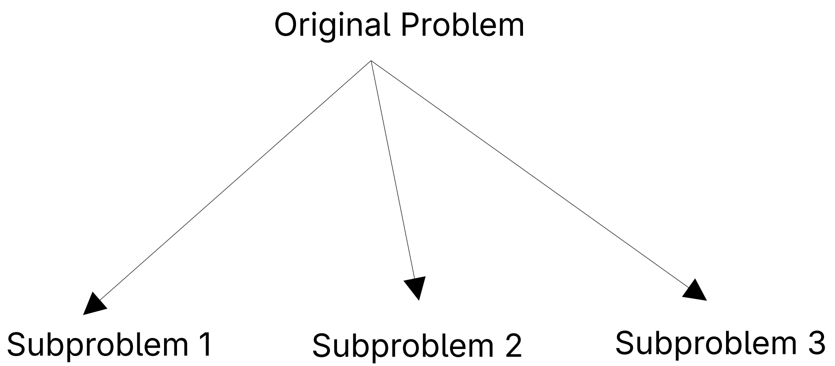
- In programming, this means a function calls itself.

- Example: Calculating a Factorial – $n! = n(n-1)! = n(n-1)(n-2)!$ and so on.

Divide and Conquer Approach

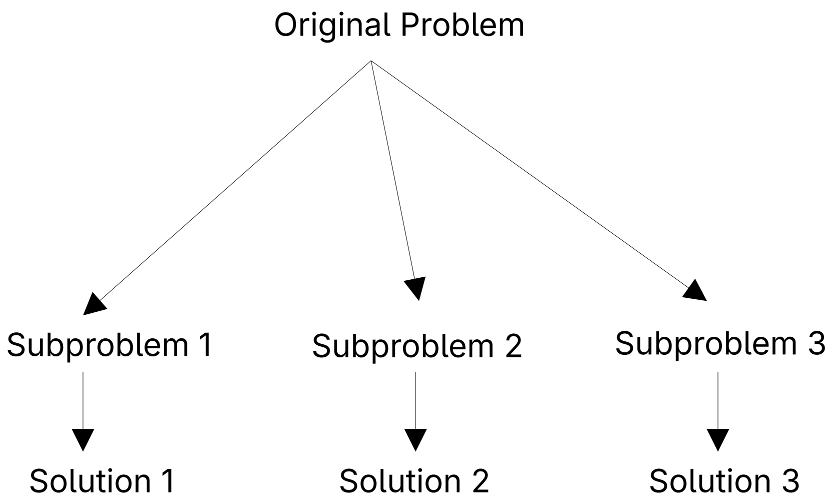
The divide and conquer paradigm involves three steps at each level of the recursion:

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem.



Divide and Conquer Approach

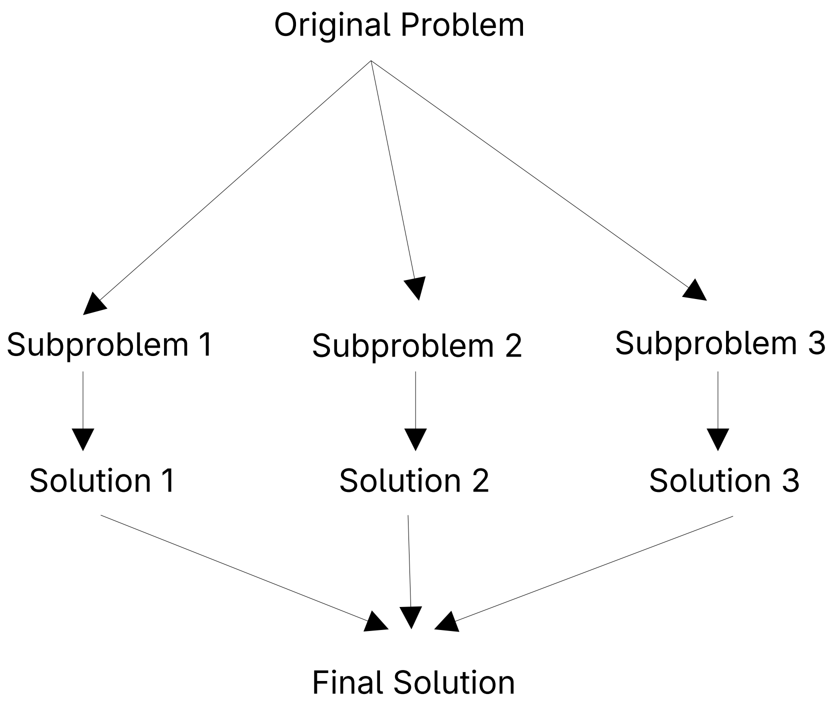
- **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.



Divide and Conquer Approach

- **Combine** the solutions to the subproblems into the solution for the original problem.

Examples: Merge sort, Quick sort, Binary search.

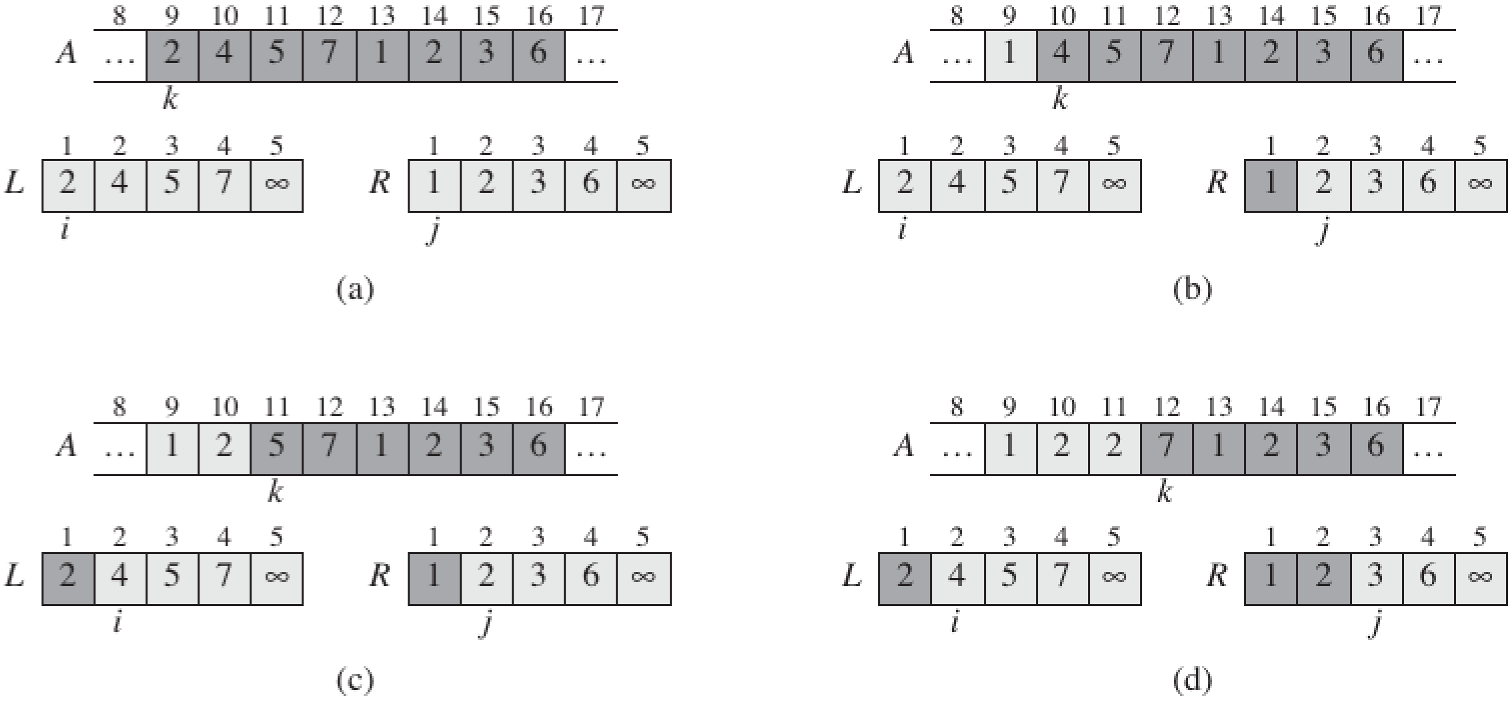


Merge Sort

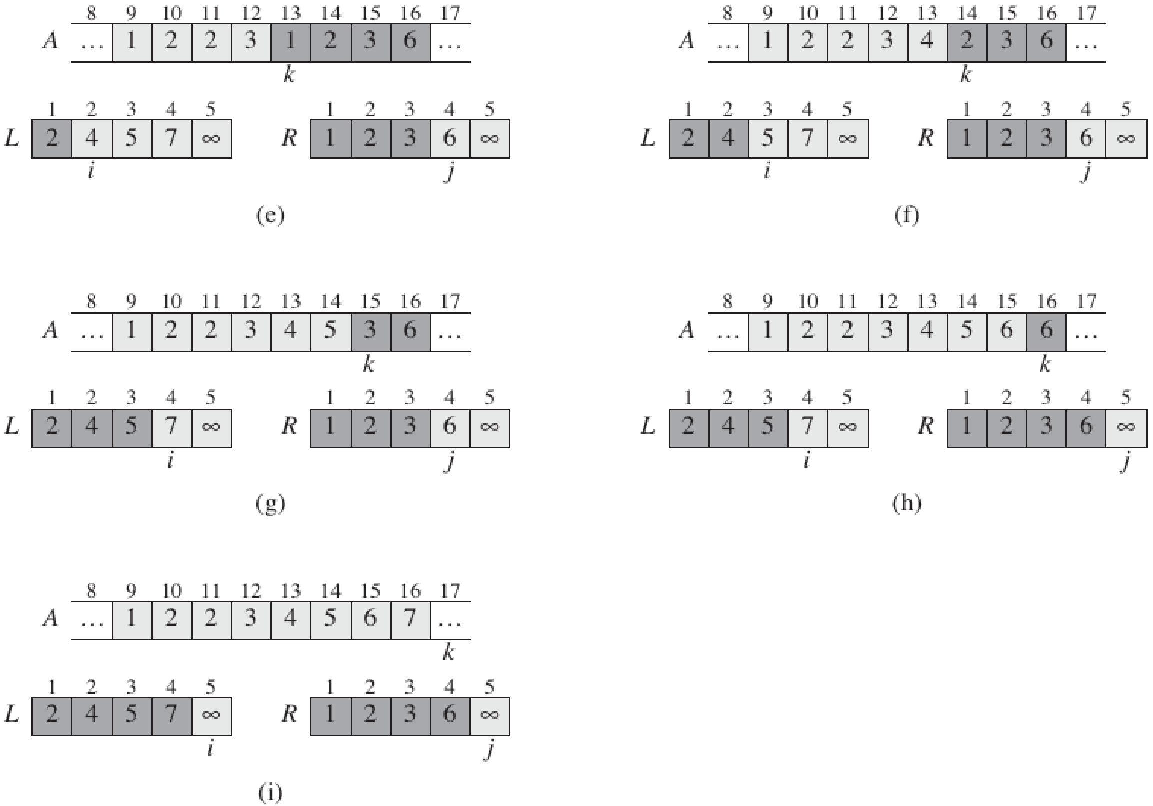
The merge sort algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

- **Divide:** Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.
- **Conquer:** Sort the two subsequences recursively using merge sort.
- **Combine:** Merge the two sorted subsequences to produce the sorted answer.

Merge Sort Simulation



Merge Sort Simulation



Algorithms and Complexities

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

- **Time Complexity:** O(n log n) in all cases. MERGE procedure takes time Θ(n) at each of the log n levels of recursion.

- **Space Complexity:** O(n) because it requires an auxiliary array of size proportional to the input array size, n, to hold the merged elements during the sorting process.

```
MERGE(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Quick Sort

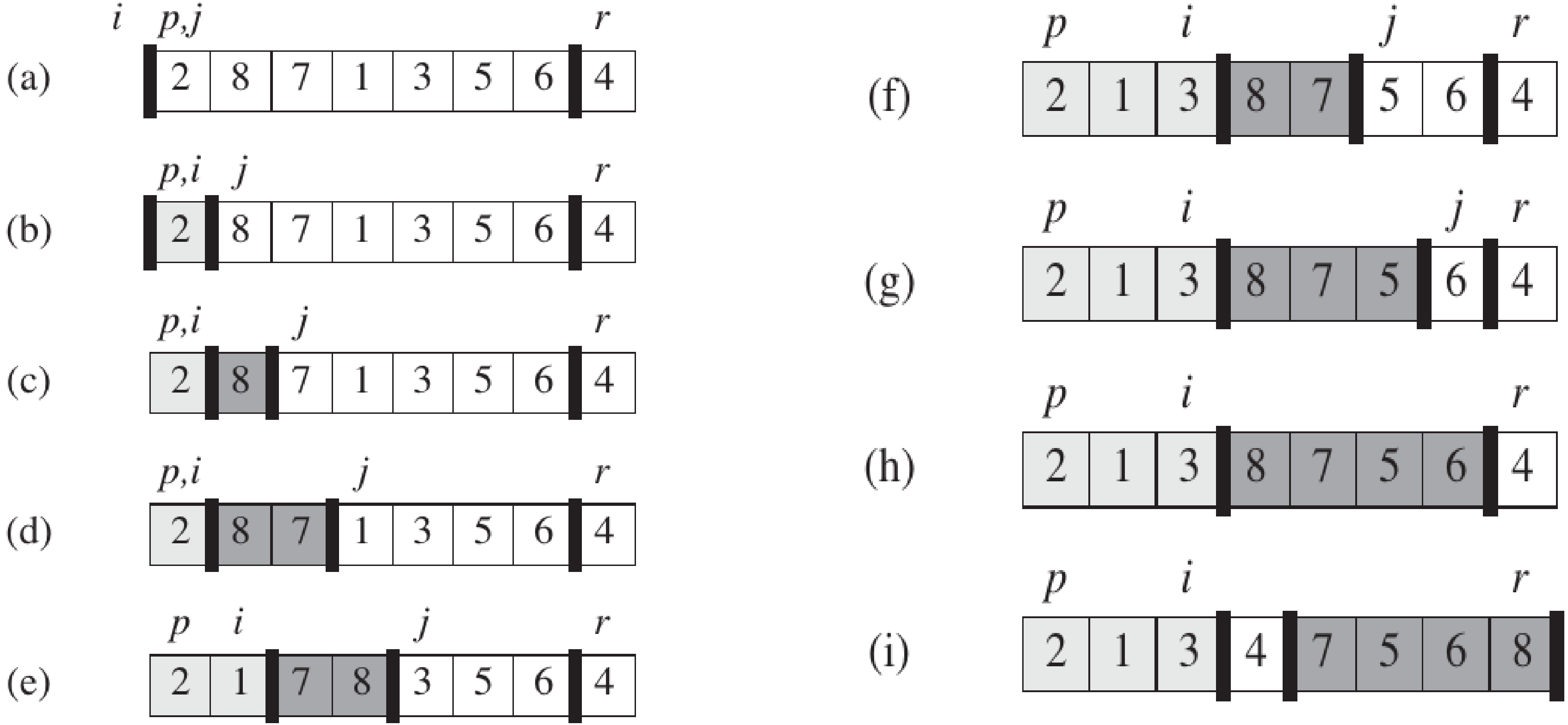
Here is the three-step divide-and-conquer process for sorting a typical subarray A[p ... r]:

- **Divide:** Partition (rearrange) the array A[p ... r] into two (possibly empty) subarrays A[p ... q-1] and A[q+1 ... r] such that each element of A[p ... q-1] is less than or equal to A[q], which is, in turn, less than or equal to each element of A[q+1 ... r]. Compute the index q as part of this partitioning procedure.

- **Conquer:** Sort the two subarrays A[p ... q-1] and A[q+1 ... r] by recursive calls to quicksort.

- **Combine:** Because the subarrays are already sorted, no work is needed to combine them. The entire array A[p ... r] is now sorted.

The Partition Operation



Algorithms and Complexities

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

To sort an entire array A, the initial call is QUICKSORT(A, 1, A.length).

- **Time Complexity:** $O(n^2)$ in the worst case when the pivot selection consistently results in highly unbalanced partitions (e.g., always picking the smallest or largest element). Time complexity is O(n log n) in other cases.
- **Space Complexity:** Typically O(log n), because it uses a recursive call stack whose maximum depth is proportional to the level of partitions.

Binary Search

Binary search is an efficient algorithm for finding an item from a sorted list of items.

- **Divide:** The algorithm "divides" the search space by finding the middle element of the current array (or subarray). The index is calculated as mid = (low + high) / 2.

- **Conquer:** It "conquers" by comparing the target value to the middle element (array[mid]).

  - Case 1 (Base Case): If target == array[mid], the element is found. The search is over.
  - Case 2: If target < array[mid], the target must be in the left half. The algorithm recursively solves the problem for the subarray from low to mid – 1.
  - Case 3: If target > array[mid], the target must be in the right half. The algorithm recursively solves the problem for the subarray from mid + 1 to high.

- **Combine:** This phase is trivial. Because the "conquer" step only solves one of the subproblems (either the left or right half), there is no work needed to combine results.

Binary Search Simulation

Let's search for the target value 23 in this sorted array:

Array: [4, 7, 10, 13, 19, 23, 29, 31, 40]

Indices: 0 1 2 3 4 5 6 7 8

Step 1:

```
[4, 7, 10, 13, 19, 23, 29, 31, 40]
 L              M              H
```

- low = 0, high = 8
- mid = (0 + 8) / 2 = 4
- Check array[4], which is 19.
- Result: 23 > 19. The target must be in the right half.

Binary Search Simulation

Step 2:

- New search space: low = 5, high = 8
- mid = (5 + 8) / 2 = 6 (using integer division)
- Check array[6], which is 29.
- Result: 23 < 29. The target must be in the left half.

```
[4, 7, 10, 13, 19, 23, 29, 31, 40]
                   L   M       H
```

Binary Search Simulation

Step 3:

- New search space: low = 5, high = 5
- mid = (5 + 5) / 2 = 5
- Check array[5], which is 23.
- Result: 23 == 23. Target found at index 5.

```
[4, 7, 10, 13, 19, 23, 29, 31, 40]
                    L,M,H
```

Binary Search Algorithm

```
FUNCTION BinarySearch(array, target):
    // Set pointers for the start and end of the array
    low = 0
    high = length(array) - 1

    // Loop as long as the search space is valid
    WHILE low <= high:
        // Find the middle index
        mid = floor((low + high) / 2)

        // Target found
        IF array[mid] == target:
            RETURN mid

        // Target is in the right half
        ELSE IF array[mid] < target:
            low = mid + 1

        // Target is in the left half
        ELSE:
            high = mid - 1

    // Target was not found in the array
    RETURN -1
```

Binary Search Complexities

- **Time Complexity:** O(log n) in worst and average cases, because the algorithm halves the search space with each comparison, so it takes a logarithmic number of steps to find the element or conclude it's not there. O(1) in best case, this occurs if the target element is the middle element on the very first check.

- **Space Complexity:**

  - Iterative: O(1), because the iterative approach only uses a few constant-size variables (low, high, mid), regardless of the array's size.
  - Recursive: O(log n), because the recursive approach requires space on the call stack for each function call, up to a maximum depth of log n.

Reference

- ***Introduction to Algorithms, Third Edition***
  - ***Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein***