Data Structures and Algorithms
Department of Computer Science
University of Engineering and Technology, Lahore

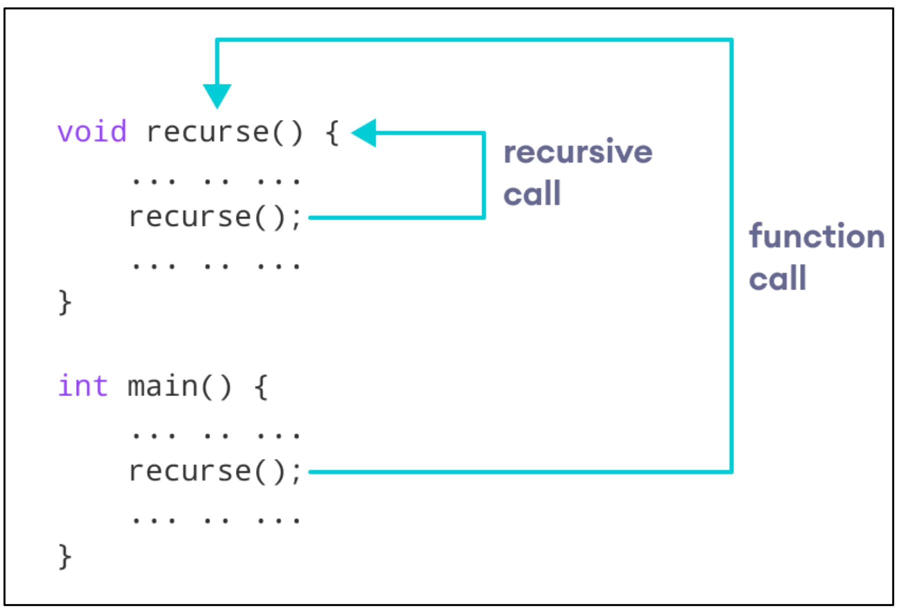
CLO1, CLO2

1. Recursion

Recursion is a way a function calls itself to solve a smaller version of the same problem. A recursive function has:

- **Base case(s):** condition(s) where the function returns directly without calling itself (prevents infinite recursion).
- **Recursive case(s):** the function calls itself with smaller or simpler inputs.

Recursion is useful when the problem can be divided into smaller subproblems of the same type (self-similar problems).



Key ideas for students:

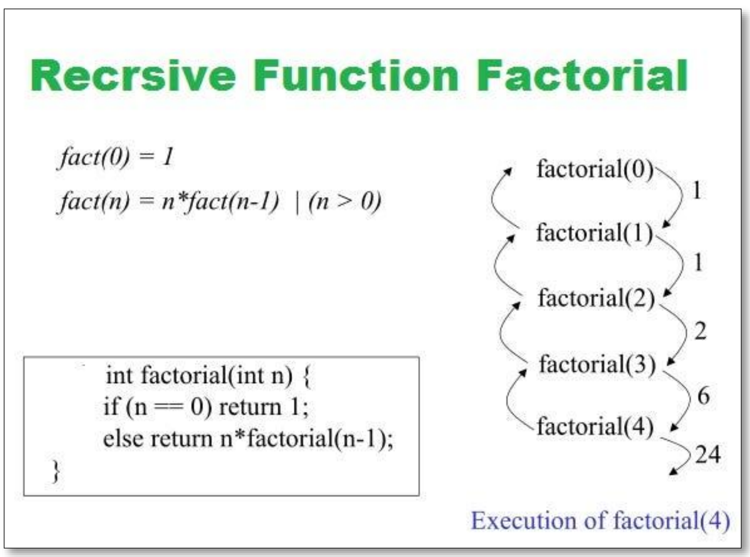
- Think of recursion like ***matryoshka dolls*** one inside another.
- Each function call gets its own memory (stack frame) with local variables.
- When the base case is reached, the calls unwind (return) step by step.

DSA Points to Discuss (Recursion)

- **Stack frames & call stack:** Each recursive call pushes a frame on the call stack; when a function returns, its frame pops.
- **Depth of recursion:** How many times the function calls itself (affects stack usage).
- **Time complexity:** Often determined by how many calls are made; can be exponential, linear, logarithmic, etc.
- **Space complexity:** Includes extra memory used and recursion call stack depth.
- **When to use recursion:** Natural for tree traversals, divide and conquer algorithms, or when a problem definition is recursive.
- **When to avoid recursion:** When it leads to deep recursion (stack overflow) or when iterative solution is simpler and more efficient.

Example (Factorial)

Problem: compute n! = n * (n-1) * (n-2) * ... * 1. By definition, 0! = 1.
Recursive idea: n! = n * (n-1)! with base case 0! = 1.



Trace for n=4:

factorial(4) -> 4 * factorial(3)

factorial(3) -> 3 * factorial(2)

factorial(2) -> 2 * factorial(1)

factorial(1) -> 1 * factorial(0)

factorial(0) -> 1 (base case)

Now unwind factorial(1) = 1*1 = 1

factorial(2) = 2*1 = 2

factorial(3) = 3*2 = 6

factorial(4) = 4*6 = 24

Complexity: Time O(n), Space O(n) due to recursion depth.

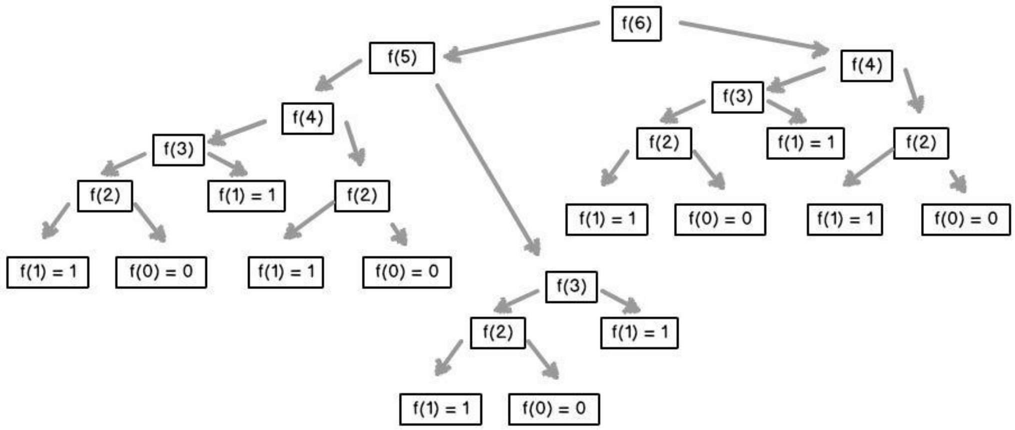
Coding Factorial (C++)

```
#include <iostream>
using namespace std;

long long factorial(int n) {
    // Base case
    if (n <= 1) return 1;
    // Recursive case
    return n * factorial(n - 1);
}

int main() {
    int n;
    cout << "Enter n: ";
    cin >> n;
    cout << n << "! = " << factorial(n) << endl;
    return 0;
}
```

Tips: Use long long for larger n (but beware overflow). Try printing when entering and exiting function to see the stack frames.

2. Fibonacci Sequence (Recursion)

Fibonacci numbers: F(0)=0, F(1)=1. For n>=2: F(n) = F(n-1) + F(n-2).
This definition is directly recursive.



DSA Point

- **Naive recursion leads to repeated work:** computing F(n) recursively does a lot of duplicate calculations exponential time.

Example (n=5)

Trace naive recursion:

F(5) = F(4) + F(3)

F(4) = F(3) + F(2)

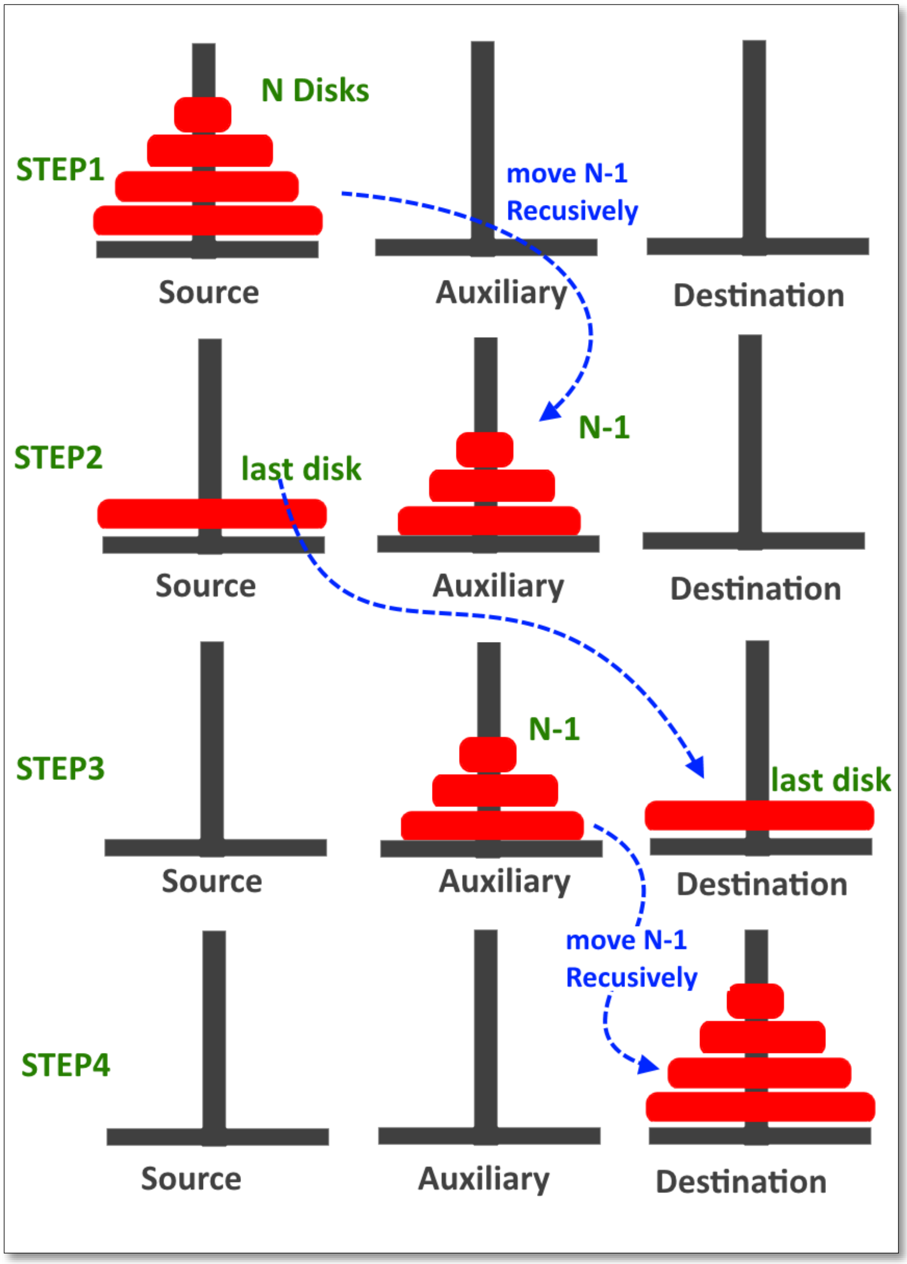
F(3) = F(2) + F(1) ... many repeats of F(2), F(3)

Coding Fibonacci (naive recursive)

```
#include <iostream>
using namespace std;

int fib(int n) {
if (n<2){ return 1;
  };
   // if (n == 0) return 0;
   // if (n == 1) return 1;
   return fib(n - 1) + fib(n - 2);
}

int main() {
   int n;
   cout << "Enter n: ";
```

```
    cin >> n;
    cout << "Fibonacci(" << n << ") = " << fib(n) << endl;
    return 0;
}
```

3. Tower of Hanoi (Recursion & Game)



Three pegs: Source (A), Auxiliary (B), Destination (C). n disks start on A (largest at bottom). Goal: move all disks to C following rules:

1. Move one disk at a time.
2. Never place a larger disk on a smaller disk.

Recursive idea: To move n disks from A to C:

- Move n-1 disks from A to B (using C as auxiliary).

- Move the largest disk from A to C.
- Move n-1 disks from B to C (using A as auxiliary).

Minimum number of moves = 2^n - 1.

Example (n=3)

Steps:

1. Move disk 1 from A to C
2. Move disk 2 from A to B
3. Move disk 1 from C to B
4. Move disk 3 from A to C
5. Move disk 1 from B to A
6. Move disk 2 from B to C
7. Move disk 1 from A to C

Coding Tower of Hanoi (C++)

```
#include <iostream>
using namespace std;
void towerOfHanoi(int n, char from, char to, char aux) {
    if (n == 0) return; // base case: no disk to move
    // move n-1 disks from 'from' to 'aux'
    towerOfHanoi(n-1, from, aux, to);
    // move last disk from 'from' to 'to'
    cout << "Move disk " << n << " from " << from << " to " << to << endl;
    // move n-1 disks from 'aux' to 'to'
    towerOfHanoi(n-1, aux, to, from);
}
int main() {
    int n;
```

```
    cout << "Enter number of disks: ";
    cin >> n;
    cout << "Sequence of moves:\n";
    towerOfHanoi(n, 'A', 'C', 'B');
    cout << "Total moves = " << ( (1<<n) - 1 ) << endl; // 2^n -1
    return 0;
}
```

DSA Notes: Exponential time O(2^n) and space O(n) for recursion stack.

4. Merge Sort (Divide and Conquer)

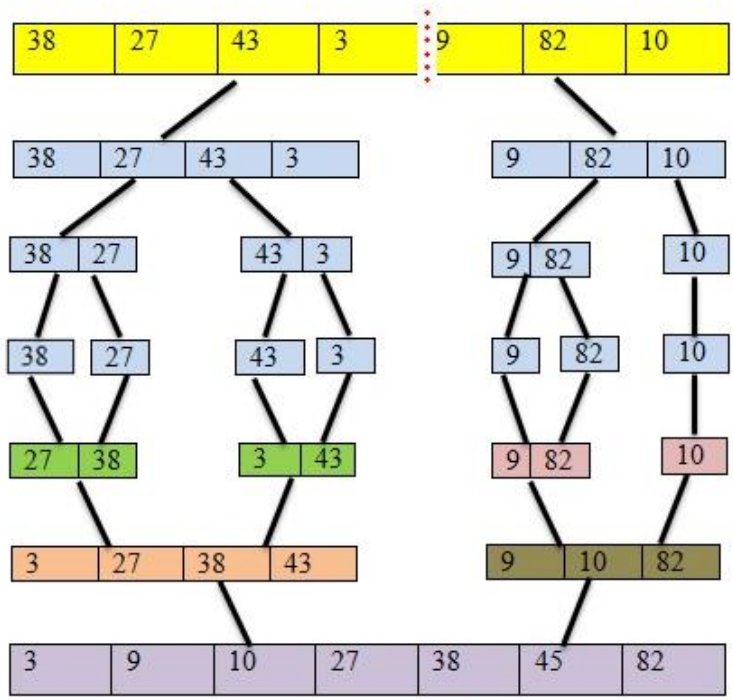
Merge Sort is a stable sorting algorithm that uses the divide-and-conquer idea:

1. **Divide:** Split array into two halves.
2. **Conquer:** Recursively sort each half.
3. **Combine:** Merge two sorted halves into one sorted array.

MERGE SORT EXAMPLE 1:

LIST OF 7 ELEMENTS							
	38	27	43	3	9	82	10

Divide the given list into two sub-lists. As the given list consists of odd number of elements, we can take 4 elements into sub-list 1 and remaining 3 elements into sub-list 2.



Merge sort guarantees O(n log n) time and is stable (equal elements keep order). It uses extra space for merging (unless implemented carefully).

DSA Points (Merge Sort)

- **Recursion depth:** $\log_2(n)$ levels (since we keep halving the array).
- **Work per level:** O(n) for merging.
- **Time complexity:** O(n log n) in worst, average, and best cases.
- **Space complexity:** O(n) extra for merging arrays (or O(1) for bottom-up in-place variations, but standard merge uses O(n)).

- **Stable vs unstable:** Merge Sort is stable.
- **Use cases:** When stable sort is required, or guarantees of O(n log n) are needed.

Coding

```
#include <iostream>
using namespace std;

// Function to merge two halves
void mergeArrays(int arr[], int left, int mid, int right) {

    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Temporary arrays (static size because no vector is allowed)
    int L[1000], R[1000];

    // Copy elements to L[] and R[]
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];

    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    // Merge temporary arrays back into arr[]
    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy remaining elements of L[]
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy remaining elements of R[]
```

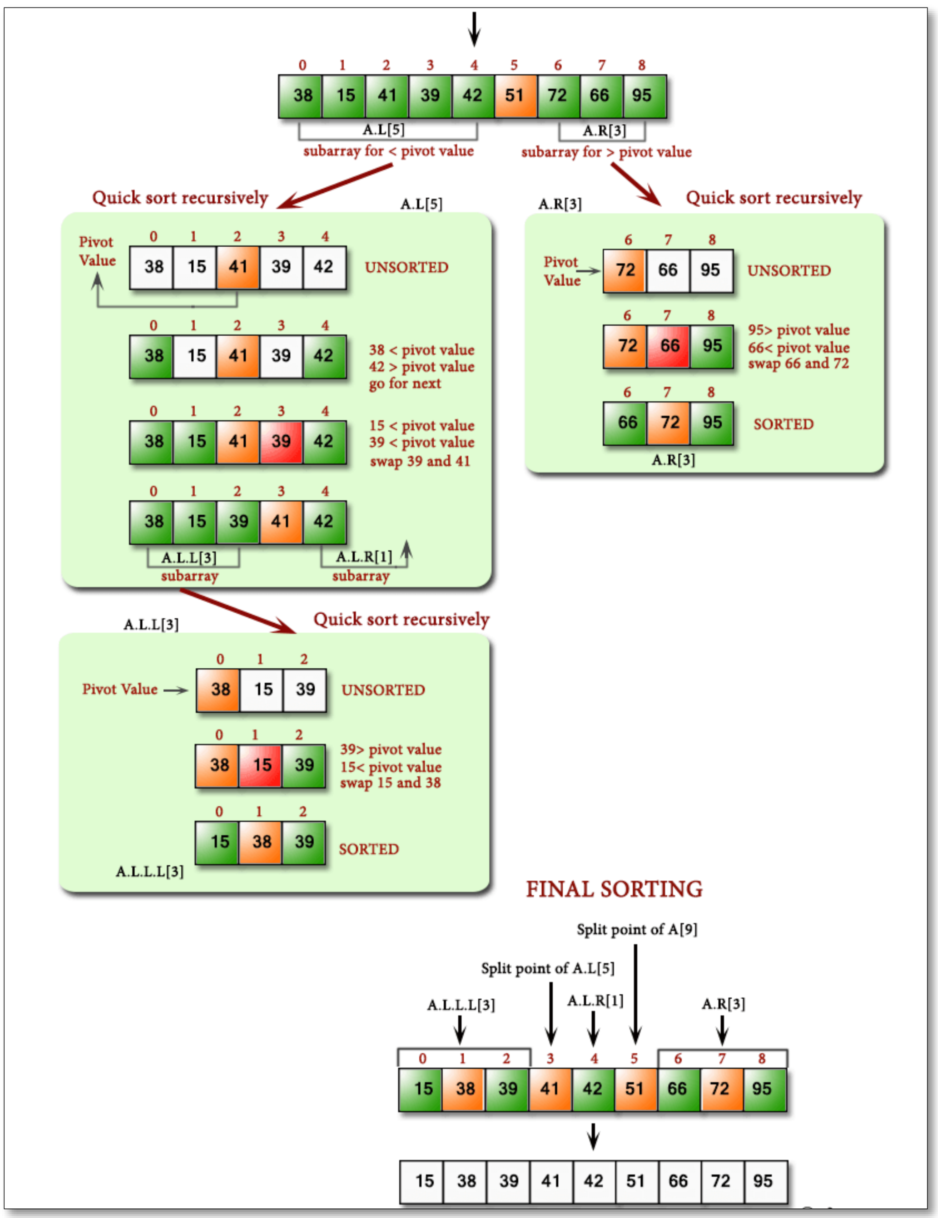
```
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Recursive merge sort
void mergeSort(int arr[], int left, int right) {
    if (left >= right)
        return; // base case: 1 element

    int mid = left + (right - left) / 2;

    mergeSort(arr, left, mid);       // left half
    mergeSort(arr, mid + 1, right); // right half
    mergeArrays(arr, left, mid, right);
}

int main() {

    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int n = 7; // number of elements

    mergeSort(arr, 0, n - 1);

    // Print result
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";

    cout << endl;

    return 0;
}
```

5. Quick Sort

Quick Sort is a divide-and-conquer sorting algorithm:

1. Choose a **pivot** element.
2. **Partition** the array so that elements ≤ pivot come before it, and elements > pivot come after.
3. Recursively sort the two partitions.

Quick sort is usually in-place and fast in practice, but worst-case time is O(n^2) if pivot is chosen poorly.



DSA Points (Quick Sort)

- **Partition schemes:** Lomuto partition, Hoare partition.
- **Average time complexity:** O(n log n). Worst-case O(n^2) (e.g., already sorted array with bad pivot).
- **Space complexity:** O(log n) average stack depth; O(n) worst-case.
- **In-place vs stable:** Quick Sort is in-place but not stable by default.
- **Pivot choices:** first element, last element, random pivot, median-of-three — better pivot selection reduces chance of worst-case.
- **Tail recursion optimization:** reduce stack usage by recursing on smaller partition first.

Example (Lomuto partition)

Array: [3,6,8,10,1,2,1] choose pivot=last element (1). After partition, pivot moves to correct place, partitions formed.

Coding Quick Sort (C++) using Lomuto

```
#include <iostream>
using namespace std;

// Custom swap function (because no built-in swap allowed)
void mySwap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

// Lomuto Partition Scheme
int partitionLomuto(int arr[], int low, int high) {

    int pivot = arr[high];   // pick last element as pivot
    int i = low - 1;         // index of smaller element

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] <= pivot) {
            i++;
            mySwap(arr[i], arr[j]);
        }
    }

    mySwap(arr[i + 1], arr[high]);
    return i + 1;
}
```

```cpp
// QuickSort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partitionLomuto(arr, low, high);  // partition index
        quickSort(arr, low, pi - 1);               // left side
        quickSort(arr, pi + 1, high);              // right side
    }
}

int main() {

    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = 6;

    quickSort(arr, 0, n - 1);

    // Print sorted array
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";

    cout << endl;
    return 0;
}
```

Tip for students: If stability matters or memory is available, use Merge Sort. If memory is tight and average speed matters, Quick Sort is often preferred.

Practice Questions

1. Convert the recursive factorial into an iterative version (loop).
2. Show how many times fib(2) is called in naive fib(5) (count calls).
3. Trace merge sort on the array [5,2,9,1,5,6]. Show the divided steps and merges.
4. Modify quick sort to choose a random pivot.

Assignment

1. Implement Quick Sort using Hoare partition scheme.