Fall 2024 – CC-213 Data Structure and Algorithms
Week 02 – Recursion

Date – Thursday, September 26, 2024, Time – 10:15 AM to 12:15 AM

1 Recursion:

- Recursion is a method of solving a problem where a function calls itself to solve smaller instances of the same problem.
- **What is recursion?** Recursion is a powerful programming technique where a function solves a problem by calling itself repeatedly with modified arguments. The idea is to break down a complex problem into smaller, more manageable subproblems, which are essentially of the same nature as the original problem. The function continues to call itself until it reaches a base case — a condition where no further recursive calls are needed. Once the base case is met, the function starts returning values, ultimately solving the overall problem.
- **How Recursion Works?** To understand recursion, think of it as breaking down a task into similar subtasks until you reach a simple problem that can be solved directly. This direct solution is known as the base case. Then, the solution to each subtask is used to construct a solution to the bigger problem, step by step.
- **Example of Recursion:**
    - **Factorial of a Number:** One of the simplest examples is calculating the factorial of a number (n!). Factorial of n is defined as:
      $n!=n\times(n-1)\times(n-2)\times...\times1$ For example, $4! = 4 \times 3 \times 2 \times 1$.
      This problem can be broken down using recursion because: $n!=n\times(n-1)!$ The recursive solution involves a function that calls itself with a reduced value of n:

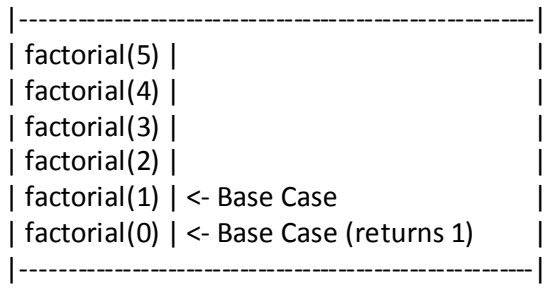
```
int factorial(int n) {
   if (n == 0 || n == 1)  // Base case
      return 1;
   else
      return n * factorial(n - 1);  // Recursive call
}
```

- **Why Use Recursion?**
    - **Simplifies Code for Complex Problems:** Recursive solutions are often simpler and more elegant for problems that can be naturally divided into similar subproblems, such as sorting algorithms (e.g., Quick Sort, Merge Sort) and data structure traversal (e.g., traversing trees).
    - **Breaking Down Problems:** It helps in breaking down problems into smaller, easier-to-solve chunks, making complex algorithms easier to implement and understand.
    - **Backtracking:** Recursion is ideal for backtracking problems like the Tower of Hanoi, maze traversal, and combinatorics, where you need to explore different paths and then backtrack.
- **Two Types of Recursion**:
    - **Direct Recursion**: A function calls itself directly.
    - **Indirect Recursion**: A function calls another function, which then calls the original function.
- **Base Case and Recursive Case**
    - **Base Case**: This is the condition that stops further recursion. It's essential to define a base case to prevent the function from calling itself indefinitely, which would lead to a stack overflow error. Example: In a factorial function, n == 0 is the base case since 0! = 1.

- **Recursive Case**: The function calls itself with modified arguments, reducing the problem's size and moving closer to the base case with each call. The recursive step defines how the function reduces the original problem into smaller sub-problems.

2 Function Calls and the Call Stack in Recursion

- In computer programming, whenever a function is called, a special data structure known as the call stack is used to keep track of the function calls. Each time a function is invoked, a new stack frame (or activation record) is created on the call stack. This frame contains important details such as:
  - The function's parameters.
  - Local variables within the function.
  - The return address to resume execution after the function finishes.
  - The value of any other registers or states at the point of the function call.
- When the function completes (reaches its base case in recursion), the frame is removed from the stack, and the program resumes execution from the return address, often with a value being passed back. This mechanism of pushing and popping frames on the call stack is fundamental to how both recursive and non-recursive function calls are handled.
- **Understanding Function Calls and the Call Stack in Recursion:** With recursive functions, each recursive call results in a new frame being added to the top of the call stack. When the base case is reached, the recursive function stops calling itself, and the function begins to unwind. This means the program starts removing the frames from the top of the stack and processes each return statement in reverse order of their addition to the stack.
- **How It Works: Step-by-Step Example:** Let's look at a simple example: Calculating the Factorial of a Number Using Recursion. The factorial of a positive integer n is calculated as: $n!=n\times(n-1)!n!$
- **Call Stack Visualization for factorial(5):** Let's see how the call stack evolves with this code for factorial(5):
  - **Initial Call**: factorial(5)
  - **First Recursive Call**: factorial(4)
  - **Second Recursive Call**: factorial(3)
  - **Third Recursive Call**: factorial(2)
  - **Fourth Recursive Call**: factorial(1)
  - **Fifth Recursive Call**: factorial(0) — This hits the **base case**, returning 1.
- Now, the stack unwinds, and the return values are passed back:
  - factorial(0) returns 1 to factorial(1).
  - factorial(1) returns $1 \times 1 = 1$ to factorial(2).
  - factorial(2) returns $2 \times 1 = 2$ to factorial(3).
  - factorial(3) returns $3 \times 2 = 6$ to factorial(4).
  - factorial(4) returns $4 \times 6 = 24$ to factorial(5).
  - factorial(5) returns $5 \times 24 = 120$ to main().



- The call stack grows until the base case is reached, and then it shrinks as the values are returned.
- **Key Points About the Call Stack in Recursion**
  - **Stack Frame Creation:** Each time a recursive function is called, a new frame is pushed onto the stack.
  - **Memory Usage:** The size of the stack can grow significantly if there are many recursive calls. For deep recursion, this can lead to a stack overflow error if the function does not reach a base case quickly enough.
  - **Unwinding:** Once the base case is reached, the recursive calls start to return. Each call's frame is popped off the stack in the reverse order they were added. This is called unwinding the stack.
- **Example: Recursive Function with a Deeper Stack:** Let's consider the **Fibonacci series**, where F(n) = F(n-1) + F(n-2):

```
int fibonacci(int n) {
   if (n <= 1)  // Base cases: F(0) = 0, F(1) = 1
      return n;
   else
      return fibonacci(n - 1) + fibonacci(n - 2);
}
```

- When fibonacci(5) is called, it triggers multiple nested recursive calls, filling up the stack with a large number of frames. Visualizing it would show overlapping calls, and managing such deep recursion can be inefficient due to the extensive use of stack space.
- **Practical Considerations**
  - **Stack Size:** If the recursion depth is too high, you may encounter a stack overflow error. For deep recursive problems, consider converting the algorithm to an iterative solution.
  - **Tail Recursion:** A special type of recursion, tail recursion, occurs when the recursive call is the last statement in the function. This can be optimized by some compilers to prevent extra stack frames, reducing memory usage.
- **Function Calls and the Call Stack**: Each recursive call adds a new frame to the call stack. When a base case is reached, the function starts resolving the stack by returning values back to the previous calls.
- **Stack Overflow**: Discuss the limitations of recursion, such as excessive memory use, which can lead to stack overflow if the recursion goes too deep.

3 Stack Overflow

- **Understanding the Limitations of Recursion:** When a function is called, a new frame (or activation record) is pushed onto the call stack to keep track of the function's execution context, including local variables, parameters, and the return address. In recursive functions, these frames are continuously added as each new recursive call is made. However, there is a finite limit to the amount of stack space available in a program, typically defined by the system or the compiler. If the recursion depth becomes too large, the program can exhaust this stack space, leading to a stack overflow error.
- **What is a Stack Overflow?** A stack overflow occurs when the call stack reaches its maximum size, meaning that no additional stack frames can be allocated. This happens when the recursive function calls itself too many times without reaching the base case. As a result, the program is unable to continue execution, and it may crash or throw a runtime error.
- **Why Does Stack Overflow Occur?**

  - **Excessive Recursion Depth:** When a recursive function goes too deep, such as in cases where the base case is not reached, or the input size is very large, the call stack fills up with too many frames.
  - **Memory Limitations:** Each function call, including recursive calls, consumes stack space. If the number of recursive calls exceeds the system's stack memory capacity, it results in a stack overflow.
  - **Uncontrolled Recursion:** A poorly defined base case or a logic error that prevents the base case from being reached will keep the function calling itself indefinitely, leading to an overflow.
- **Example: Stack Overflow in Recursive Function:** Let's look at a simple example of a recursive function that does not reach its base case correctly:

```
#include <stdio.h>

void recursiveFunction() {
   printf("Calling function...\n");
   recursiveFunction(); // No base case, function keeps calling itself indefinitely
}

int main() {
   recursiveFunction(); // Start the recursive function
   return 0;
}
```

- In this code snippet, the recursiveFunction() calls itself repeatedly without ever terminating. Since there is no base case, the recursion continues infinitely, consuming stack space with each call until a stack overflow occurs.
- **Practical Example: Stack Overflow in Deep Recursion:** Suppose we have a recursive function to calculate the factorial of a number, but we call it with an extremely large input, such as factorial(10000). Even though the base case is defined, the function must make 10,000 recursive calls, which is likely to exceed the stack space limit:

```
#include <stdio.h>

long long factorial(int n) {
   if (n == 0)  // Base case
      return 1;
   else
      return n * factorial(n - 1);  // Recursive call
}

int main() {
   int n = 10000;   // Very large number
   printf("Factorial of %d is %lld\n", n, factorial(n));  // Stack overflow will likely occur
   return 0;
}
```

- **Consequences of Stack Overflow**
  - **Program Crash:** The most common result of a stack overflow is an abrupt program crash.

- **Unpredictable Behavior:** In some cases, a stack overflow can lead to undefined or unpredictable program behavior, especially if the overflow overwrites critical memory areas.
- **Runtime Error Messages:** Many programming environments will display a specific runtime error, such as "StackOverflowError" in Java or a segmentation fault in C/C++.

- **How to Prevent Stack Overflow in Recursion?**
  - **Define a Proper Base Case:** Ensure that every recursive function has a well-defined base case that halts further recursive calls.
  - **Limit Recursion Depth:** Set a maximum depth for recursion to prevent excessive use of stack space, or use language-specific mechanisms like setrecursionlimit in Python.
  - **Switch to Iterative Solutions:** If a problem is prone to deep recursion, consider rewriting it as an iterative solution using loops. Iterative solutions do not consume additional stack space with each loop iteration.
  - **Tail Recursion Optimization:** If using a language that supports it, consider using tail recursion optimization, which reuses stack frames and prevents the stack from growing excessively.
- **Analyzing Stack Overflow Through Fibonacci Example**
  - The Fibonacci series is a classic example that can lead to a stack overflow if calculated recursively with a large value of n. The recursive calls for fib(n) will break down into overlapping calls like fib(n-1) and fib(n-2) repeatedly. For larger values of n, the number of calls grows exponentially, resulting in a huge number of stack frames being created, ultimately causing a stack overflow.

```
#include <stdio.h>

int fibonacci(int n) {
   if (n <= 1)
      return n;
   else
      return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
   int n = 1000;   // Large value for Fibonacci sequence
   printf("Fibonacci of %d is %d\n",n,fibonacci(n)); //Will cause stack overflow for large n
   return 0;
}
```

4 Examples of Recursive Problems:

- **Towers of Hanoi:** A more complex example that demonstrates the power of recursion in solving problems with multiple recursive calls. In the Tower of Hanoi problem, you have three rods and n disks of varying sizes. The goal is to move all disks from one rod to another, obeying the rules that only one disk can be moved at a time and a larger disk cannot be placed on a smaller disk.
  - This problem can be solved using recursion by breaking it down as follows:
    - Move n-1 disks from the source rod to an auxiliary rod.
    - Move the nth disk directly to the destination rod.
    - Move the n-1 disks from the auxiliary rod to the destination rod.
  - **Recursive Code Example:** Tower of Hanoi

```
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod) {
```

```
    if (n == 1) {
        printf("Move disk 1 from rod %c to rod %c\n", from_rod, to_rod);
        return;
    }
    towerOfHanoi(n - 1, from_rod, aux_rod, to_rod);
    printf("Move disk %d from rod %c to rod %c\n", n, from_rod, to_rod);
    towerOfHanoi(n - 1, aux_rod, to_rod, from_rod);
}
```

- **In this code:** The base case is when n == 1, meaning only one disk is left to move directly. The recursive call breaks the problem into smaller subproblems (n-1 disks) until the base case is reached.
- **Step Count Equations for Tower of Hanoi:** The **step count** measures the number of moves or steps required to solve the problem for a given number of disks (n). Let: T(n) be the total number of moves required to transfer n disks.
  - **Recurrence Relation**: For n disks: $T(n) = 2 \times T(n-1)+1$
  - where: $2 \times T(n - 1)$: Two recursive steps, one for moving n-1 disks initially and one for returning them after moving the largest disk. +1: Moving the nth (largest) disk directly from source to destination.
  - **Base Case**: When n = 1 (single disk), the number of moves required is: T(1)=1
  - **Solving the Recurrence Relation**: Using the recurrence relation, we can derive the exact number of steps needed:
    $T(n)=2 \times T(n-1)+1$
    Expanding it recursively: $T(n)=2 \times (2 \times T(n-2)+1)+1 = 2^2 \times T(n-2)+2+1$
    Continuing this expansion: $T(n)=2^3 \times T(n-3)+2^2+2+1$
    Generalizing: $T(n)=2^{n-1} \times T(1)+\sum k=0 \text{ to } n-2\ (2^k)$
    Since T(1) = 1: $T(n)=2^{n-1}+\sum k=0 \text{ to } n-2\ (2^k)$
    The sum of a geometric series: $\sum k=0 \text{ to } n-2\ (2^k)=2^{n-1}-1$
    Thus, the exact solution is: $T(n)=2^{n-1}+2^{n-1}-1=2^n-1$
- Time **Complexity** Analysis
  The number of steps required for n disks is: $T(n)=2^n-1$
  As n grows, the time complexity is **exponential**: $T(n)=O(2^n)$
  This indicates that the time required to solve the problem doubles with each additional disk. For even a relatively small value of n (e.g., n = 20), the number of moves becomes significantly large.

- **Tower of Hanoi Recursive Tree for 3 Disks**
  - Problem Setup
  - Pegs: A (Source), B (Auxiliary), C (Destination)
  - Number of Disks (n): 3
- Recursive Tree Structure for n = 3
  - The recursive solution for n = 3 disks involves the following steps:
    - Move the top 2 disks from A to B using C as auxiliary.
    - Move the 3rd disk (largest) from A to C.
    - Move the 2 disks from B to C using A as auxiliary.
  - Step-by-Step Analysis
    - Let's break down the problem and construct the recursive tree to understand each move. Below is the tree structure and detailed explanation:
    - Root Node (T(3)):

- Action: Move 3 disks from A to C using B as auxiliary.
- Breakdown: The problem splits into 3 subproblems:
  - Move 2 disks (T(2)) from A to B using C as auxiliary.
  - Move disk 3 from A to C.
  - Move 2 disks (T(2)) from B to C using A as auxiliary.

▪ Subproblem 1 (T(2)): Move 2 disks from A to B using C.
- Breakdown:
  - Move 1 disk (T(1)) from A to C using B.
  - Move disk 2 from A to B.
  - Move 1 disk (T(1)) from C to B using A.

▪ Subproblem 2 (T(1)): Move 1 disk from A to C using B.
- Base Case: Move 1 disk directly.

- **Recursive Tree Diagram**
  - Below is a visualization of the recursive tree for n = 3. Each node represents a subproblem, breaking down into smaller subproblems until the base case is reached.

T(3): Move 3 Disks from A to C

/ | \

T(2): Move 2 Disks from A to B *Move Disk 3* **T(2): Move 2 Disks from B to C**

/ | \ | / | \

T(1): Move 1 Disk from A to C | Move Disk 2 from A to B | **T(1): Move 1** from C to B | *Disk Move Disk 3* from A to C | **T(1): Move 1 Disk** from B to A | *Move Disk 2* from B to C | **T(1): Move 1 Disk** from A to C

  - Recursive Tree Explanation
    - Root (T(3)):
      - This represents the problem of moving 3 disks from A to C. The problem is broken into:
        - Moving 2 disks (T(2)) from A to B.
        - Moving the 3rd disk from A to C.
        - Moving 2 disks (T(2)) from B to C.
    - Subproblem (T(2)):
      - Moving 2 disks from A to B is broken down into:
        - Moving 1 disk (T(1)) from A to C.
        - Moving the 2nd disk from A to B.
        - Moving 1 disk (T(1)) from C to B.
    - Base Case (T(1)):
      - For a single disk, the problem is simple: move it directly from the source peg to the destination peg.
  - Step-by-Step Disk Moves for n = 3
    - From the recursive tree, the complete sequence of disk moves is:
      - Move disk 1 from A to C.
      - Move disk 2 from A to B.
      - Move disk 1 from C to B.
      - Move disk 3 from A to C.
      - Move disk 1 from B to A.
      - Move disk 2 from B to C.
      - Move disk 1 from A to C.
  - Step Count and Time Complexity

        - Step Count: The number of moves for n = 3 is:
            - T(3) = 2^3 - 1 = 7
        - Time Complexity: The time complexity for the Tower of Hanoi problem is exponential:
            - O(2^n)
    - The recursive tree for the Tower of Hanoi problem illustrates how a seemingly complex problem is broken down into smaller subproblems until the base case is reached. Understanding the recursive structure helps in visualizing the flow of recursive calls and their resolutions, which is key to mastering recursion.
- **Recursive Binary Search**: Recursive Binary Search is a technique used to search for an element in a sorted array by repeatedly dividing the search interval in half. The idea is to leverage the sorted nature of the array to eliminate half of the remaining elements with each comparison, reducing the problem's size significantly at each step. Here's how recursive binary search works:
    - **Divide and Conquer:** The algorithm starts by comparing the target element (the element we're searching for) with the middle element of the array.
        - If the target is equal to the middle element, the search is complete, and the algorithm returns the position of the target.
        - If the target is smaller than the middle element, it eliminates the right half of the array and recursively searches in the left half.
        - If the target is larger than the middle element, it eliminates the left half of the array and recursively searches in the right half.
    - **Recursive Calls:** Each recursive call works on a smaller portion of the array. This continues until either the target element is found or the search range is reduced to zero (meaning the element is not present in the array).
    - **Base Case:** The recursion stops when either the element is found, or the search range becomes invalid (i.e., the low index exceeds the high index), indicating that the target element does not exist in the array.

```
#include <stdio.h>
// Recursive function for binary search
int binarySearch(int arr[], int low, int high, int target) {
  if (low <= high) {
    int mid = low + (high - low) / 2;  // Calculate the middle index
    // Base case: Check if the target is at mid
    if (arr[mid] == target)
      return mid;
    // If target is smaller, search the left half
    if (arr[mid] > target)
      return binarySearch(arr, low, mid - 1, target);
    // If target is larger, search the right half
    return binarySearch(arr, mid + 1, high, target);
  }
  // Target is not present in the array
  return -1;
}

int main() {
  int arr[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
  int n = sizeof(arr) / sizeof(arr[0]);
```

```
    int target = 7;
    int result = binarySearch(arr, 0, n - 1, target);

    if (result != -1)
        printf("Element found at index %d\n", result);
    else
        printf("Element not found\n");
    return 0;
}
```

- **Breakdown of the Recursive Binary Search:** Let's take an example of searching for 7 in the array {1, 3, 5, 7, 9, 11, 13, 15, 17, 19}:
  - **Initial Call**:
    - low = 0, high = 9 (the entire array).
    - The middle index is mid = (0 + 9) / 2 = 4, and arr[mid] = 9.
    - Since 7 < 9, the algorithm recursively searches in the left half, from index 0 to 3.
  - **First Recursive Call**:
    - low = 0, high = 3 (array segment {1, 3, 5, 7}).
    - The middle index is mid = (0 + 3) / 2 = 1, and arr[mid] = 3.
    - Since 7 > 3, the algorithm recursively searches in the right half, from index 2 to 3.
  - **Second Recursive Call**:
    - low = 2, high = 3 (array segment {5, 7}).
    - The middle index is mid = (2 + 3) / 2 = 2, and arr[mid] = 5.
    - Since 7 > 5, the algorithm recursively searches in the right half, at index 3.
  - **Third Recursive Call**:
    - low = 3, high = 3 (single element 7).
    - The middle index is mid = 3, and arr[mid] = 7.
    - Since 7 == 7, the search is complete, and the function returns index 3.
- **Time Complexity Analysis**
  - **Best Case**: The element is found at the first comparison, which occurs at the middle of the array. This takes constant time, O(1).
  - **Worst Case**: Each recursive call reduces the search space by half. In the worst case, when the element is not found or is located near the ends of the array, the time complexity is O(log n). This is because the algorithm halves the array size in each step, and after log2(n) divisions, only one element remains.
- **Step Count Equations:** For a binary search with an array size n, the number of steps taken by the recursive calls depends on the following:
  - **Best Case**: The element is found in the first comparison.
    - **Step count equation**: T(n) = 1 for n elements.
  - **Worst Case**: The algorithm must make multiple recursive calls, reducing the array size by half each time.
    - **Step count equation**: T(n) = T(n/2) + 1 with the base case T(1) = 1. Solving this recurrence gives T(n) = O(log n).

5 Advantages of Recursion

- **Simpler Code for Complex Problems**: Recursion can make code more elegant and readable for problems that have a naturally recursive structure (e.g., tree traversal, backtracking algorithms).

- **Breakdown of Complex Problems**: It allows breaking down large problems into smaller, more manageable pieces.

6 Disadvantages of Recursion

- **Memory Overhead**: Each recursive call adds a new stack frame, which can lead to higher memory usage, especially in languages without tail call optimization.
- **Performance Considerations**: In some cases, recursive solutions may be less efficient than iterative solutions due to repeated calculations or function call overhead.

7 Tail Recursion

- **What is Tail Recursion?** A specific type of recursion where the recursive call is the last operation in the function. Tail recursion is important because it can be optimized by compilers to use less memory (Tail Call Optimization).

8 Mathematical Foundation of Recursion (Recurrence Relations)

- **Defining Recurrence Relations**: Some recursive problems can be represented using recurrence relations (equations that define a sequence recursively).
- **Example**: The Fibonacci sequence can be defined as $F(n)=F(n-1)+F(n-2)$ with base cases $F(0)=0$ and $F(1)=1$.
- **Solving Recurrence Relations**: For complex problems, recurrence relations can be used to analyze the performance and complexity of recursive algorithms.

9 Time and Space Complexity of Recursive Functions

- **Time Complexity**: Analyzing how the recursive function's time complexity grows with input size. Example: In the Fibonacci sequence, the naive recursive implementation has an exponential time complexity O(2^n), while a dynamic programming approach reduces it to O(n).
- **Space Complexity**: Recursive algorithms often have higher space complexity due to the memory used by the call stack. Discuss how to calculate space complexity based on the depth of recursion.

10 Recursion in Data Structures

- **Recursion in Trees**: Tree traversal algorithms (preorder, inorder, postorder) naturally lend themselves to recursion.
- **Recursion in Graphs**: Depth-First Search (DFS) on a graph is another classic example of recursion in data structures.
- **Recursion in Divide and Conquer Algorithms**: Recursion plays a crucial role in divide and conquer strategies like Merge Sort, Quick Sort, and algorithms like Strassen's Matrix Multiplication.

11 Practical Applications of Recursion

- **Backtracking Algorithms**: Examples include solving the N-Queens problem, maze-solving algorithms, and generating permutations and combinations.
- **Recursion in Parsing**: Recursive Descent Parsers used in compilers.
- **Fractals and Graphics**: Recursion can generate complex patterns like fractals in computer graphics.

12 Common Pitfalls and How to Avoid Them

- **Missing Base Case**: Ensure every recursive function has a clear and reachable base case to avoid infinite recursion.
- **Overlapping Subproblems**: In problems like Fibonacci, overlapping subproblems can lead to exponential time complexity unless optimized using memoization.
- **Stack Overflow**: Be mindful of the recursion depth, especially in languages without tail call optimization.