**Introduction to Recursion**

**Definition:**

Recursion is a programming technique where a function calls itself to solve a problem. The idea is to break down a complex problem into smaller sub-problems, and the function continues to call itself with smaller instances of the problem until it reaches a base case (the simplest form of the problem) that can be directly solved without further recursion.

**Why Use Recursion?**

Recursion is useful when the problem can be broken down into similar smaller problems. It provides a clean and elegant solution to problems that exhibit a recursive structure, such as factorial calculation, tree traversal, and the Tower of Hanoi problem.

**Example Problem: Factorial**

Factorial is often used as a classic example of recursion. The factorial of a number n (denoted as n!) is defined as the product of all positive integers less than or equal to n. The recursive definition of factorial is:

* n! = n \* (n-1)!
* Base Case: 0! = 1

**Writing a Recursive Function**

To write a recursive function, two key things must be done:

1. **Base Case**: This is the condition under which the function stops calling itself and returns a value.
2. **Recursive Case**: This is where the function calls itself with a smaller or simpler version of the problem.

**Example: Factorial Function in Python**

python

Copy code

def factorial(n):

# Base case: if n is 0, return 1

if n == 0:

return 1

# Recursive case: otherwise, return n \* factorial(n-1)

else:

return n \* factorial(n-1)

* **Base Case**: if n == 0: return 1 – When n is 0, we return 1 directly. This stops the recursion.
* **Recursive Case**: return n \* factorial(n-1) – The function keeps calling itself with n-1, breaking down the problem.

**Example Walkthrough:**

If factorial(3) is called:

1. factorial(3) calls factorial(2)
2. factorial(2) calls factorial(1)
3. factorial(1) calls factorial(0)
4. factorial(0) returns 1 (base case)
5. Now the recursive calls return:
   * factorial(1) returns 1 \* 1 = 1
   * factorial(2) returns 2 \* 1 = 2
   * factorial(3) returns 3 \* 2 = 6

**How Does Recursion Work?**

1. **Function Call Stack**: When a recursive function is called, each recursive call is pushed onto the call stack. The function keeps calling itself with a smaller problem until it reaches the base case.
2. **Base Case**: When the base case is reached, the function does not make any further recursive calls. Instead, it starts returning values back up the call stack.
3. **Return Values**: Once the base case is reached, the function returns values to the previous recursive calls. These return values are multiplied (or processed) to give the final result.

**Example Execution of Factorial(3):**

scss

Copy code

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

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

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

factorial(0) -> 1 (base case)

After the base case is hit, the recursive calls start returning:

scss

Copy code

factorial(1) = 1

factorial(2) = 2 \* 1 = 2

factorial(3) = 3 \* 2 = 6

**Analysis of Recursive Functions**

**Time Complexity:**

Time complexity in recursion depends on how many recursive calls are made and what work is done at each level.

* For example, the time complexity of the factorial(n) function is O(n) because we have n recursive calls and constant work (multiplication) is done in each call.

**Space Complexity:**

Space complexity depends on how deep the recursion goes (i.e., the depth of the call stack). For a simple recursion like factorial(n), the space complexity is also O(n) since there are n function calls on the stack.

* In general, for a recursive function with n recursive calls, the space complexity is O(n).

**Example Complexity Analysis:**

For factorial(n):

* **Time Complexity**: O(n) (each call does constant work).
* **Space Complexity**: O(n) (each call adds a new layer to the call stack).

**Drawbacks of Recursion**

1. **Stack Overflow**: Recursive functions use the system's call stack to keep track of function calls. If recursion is too deep (e.g., for large inputs), it can cause a **stack overflow**, where the call stack exceeds the system's memory limit. This can result in program crashes.
2. **Inefficiency in Some Cases**: Some recursive solutions (such as naive Fibonacci) can result in repeated calculations, leading to inefficiency. For example, calculating Fibonacci recursively leads to an exponential time complexity (O(2^n)).
3. **Memory Usage**: Recursive functions consume more memory due to the call stack. Each recursive call consumes memory for storing local variables, parameters, and return addresses.

**Example of Stack Overflow:**

If you define a recursive function with no base case or a very deep recursion, it could result in a stack overflow:

python

Copy code

def infinite\_recursion():

return infinite\_recursion()

This function will keep calling itself indefinitely, eventually leading to a stack overflow error.

**Memoization: A Solution to Recursion's Drawbacks**

Memoization is a technique where results of expensive function calls are cached, so they aren't recalculated multiple times. This can greatly improve the efficiency of recursive solutions.

**Example: Optimizing Fibonacci with Memoization**

python

Copy code

def fibonacci(n, memo={}):

if n in memo:

return memo[n]

if n == 0:

return 0

elif n == 1:

return 1

memo[n] = fibonacci(n-1, memo) + fibonacci(n-2, memo)

return memo[n]

Here, memo is a dictionary that stores already computed Fibonacci numbers, which eliminates redundant calculations.

**How to Answer in a Quiz?**

**1. What is recursion?**

Recursion is a programming technique where a function calls itself to solve a problem. It breaks a problem into smaller sub-problems, and the process repeats until a base case is reached.

**2. How does recursion work?**

Recursion works by calling the same function with smaller inputs. Each recursive call adds to the call stack until it reaches the base case, where no further recursive calls are made. After that, the function starts returning values back through the call stack.

**3. Explain the base case in recursion.**

The base case in recursion is the condition under which the function stops calling itself. It defines when the function has broken down the problem enough to return a value without further recursive calls.

**4. What are the time and space complexities of a recursive function?**

* **Time Complexity**: The total amount of time taken by the recursive function depends on the number of recursive calls and the work done in each call.
* **Space Complexity**: The space complexity is determined by the depth of the recursion (i.e., the number of function calls on the stack).

**5. What are the drawbacks of recursion?**

* Stack overflow due to deep recursion.
* Higher memory consumption due to function call stack.
* Potential inefficiency if repeated computations are not avoided.

**6. What is memoization in recursion?**

Memoization is a technique used in recursive algorithms where intermediate results are stored (cached) to avoid recalculating them, thereby improving efficiency.

**Conclusion:**

When answering quiz questions related to recursion, ensure that you highlight:

1. The **basic definition** of recursion.
2. The importance of the **base case**.
3. The **time and space complexities**.
4. **Drawbacks** like stack overflow and inefficiency.
5. Solutions like **memoization** that optimize recursive solutions.

**Long Answer Questions for Quiz:**

1. **Explain how recursion works with an example.**
2. **What are the advantages and disadvantages of recursion?**
3. **How would you optimize a recursive solution to avoid inefficiency or stack overflow?**
4. **Describe the time and space complexity of recursive functions.**
5. **Compare recursion and iteration. When would you choose recursion over iteration?**
6. **How can memoization be used to improve the efficiency of recursive algorithms?**