Recursion

- Definition
- Recursive Methods
- Example 1
- How does Recursion work?
- Example 2
- Problems with Recursion
- Infinite Recursion
- Exercises
Definition

- Recursion is a powerful concept that helps to simplify the solution of complex problems.
- Recursion means defining something in terms of itself.
- This means that the solution of a problem is expressed in terms of a similar problem but simpler.
- That is, solving the simpler problem leads to the solution of the original one.
- Recursive solutions are shorter, easier to understand and implement.
Recursive Methods

• A recursive method is a method that calls itself directly or indirectly.

• A recursive method has two major steps:
  – **recursive step** in which the method calls itself
  – **base step** which specifies a case with a known solution

• The method should select one of two steps based on a criteria.

• The recursive step provides the repetition needed for the solution and the base step provides the termination.

• Executing recursive algorithms goes through two phases:
  – Expansion in which the recursive step is applied until hitting the base step
  – “Substitution” in which the solution is constructed backwards starting with the base step
Example 1

- Let us construct a recursive version of a program that evaluates the factorial of an integer, i.e. $\text{fact}(n) = n!$

- $\text{fact}(0)$ is defined to be 1. i.e. $\text{fact}(0) = 1$. This is the base step.

- For all $n > 0$, $\text{fact}(n) = n \times \text{fact}(n – 1)$. This is the recursive step.

- As an example, consider the following:

  - $\text{fact}(4) = 4 \times \text{fact}(3)$
    $= 4 \times (3 \times \text{fact}(2))$
    $= 4 \times (3 \times (2 \times \text{fact}(1)))$
    $= 4 \times (3 \times (2 \times (1 \times \text{fact}(0))))$
    $= 4 \times (3 \times (2 \times (1 \times 1)))$
    $= 4 \times (3 \times (2 \times 1))$
    $= 4 \times (3 \times 2)$
    $= 4 \times 6$
    $= 24$
import java.io.*;

public class Factorial
{
    public static void main(String[] args) throws IOException
    {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter an integer: ");
        long fact = ((long) Integer.parseInt(in.readLine()));
        long answer = factorial(fact);
        System.out.println("The factorial is: " + answer);
    }

    public static long factorial(long number)
    {
        if(number == 0) //base step
            return 1;
        else //recursive step
            return number * factorial(number - 1);
    }
}

Unit 18
How does recursion work?

• To appreciate how recursion works, at every recursive call a new activation record is created.

• An activation record is a collection of data in the main memory associated with each recursive call.

• The data includes the values of local variables at that call, the parameters passed to the recursive method and the return address in the main memory.

<table>
<thead>
<tr>
<th>Local Variables</th>
<th>Data</th>
<th>Return Address</th>
</tr>
</thead>
</table>

An Activation Record
Activation Records – Example 1

- To calculate fact(4), the following sequence of operations takes place.

- Each call to fact(x) creates a new activation record. The process ends when the base case is reached.

```
fact(4), Since 4 ≠ 0, Call fact(3)

fact(3), Since 3 ≠ 0, Call fact(2)

fact(2), Since 2 ≠ 0, Call fact(1)

fact(1), Since 1 ≠ 0, Call fact(0)

fact(0), Since 0 = 0, Return 1
```

```
Return 1 * fact(0) = 1
Return 2 * fact(1) = 2
Return 3 * fact(2) = 6
Return 4 * fact(3) = 24
```

Return 4 * fact(3) = 24
Example 2 – Fibonacci Numbers

• Consider the following sequence of numbers: 1, 1, 2, 3, 5, 8, 13

• Except for the first two integers, each integer is the sum of the previous two integers.

• This sequence is known as fibonacci (pronounced fibo – naachee) numbers.

• Fibonacci numbers have important applications in computer science and mathematics.

• A recursive solution for calculating the $n^{th}$ ($n \geq 1$) is as follows:

  • BASE CASE: $\text{fib}(n) = 1$ if $n = 1$ or 2 ($n \leq 2$)
  • RECURSIVE CASE: $\text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2)$
import java.io.*;

public class Fibonacci
{
    public static void main(String[] args) throws IOException
    {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter an integer: ");
        int n = Integer.parseInt(in.readLine());
        int answer = fib(n);
        System.out.println("The "+n+"th fibonacci number is: "+answer);
    }

    public static int fib(int number)
    {
        if(number <= 2) //base step
            return 1;
        else //recursive step
            return fib(number - 1) + fib(number - 2);
    }
}
Problems with Recursion

- We may use our previous program to illustrate some of the problems with recursive solutions.

- Consider the previous solution of calculating fibonacci numbers using the recursive relation:
  \[ \text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2), \text{fib}(2) = \text{fib}(1) = 1. \]

- Suppose we want to calculate \( \text{fib}(5) \).
- \[ \text{fib}(5) = \text{fib}(4) + \text{fib}(3) \]
  \[ = [\text{fib}(3) + \text{fib}(2)] + \text{fib}(3) \]
  \[ = [[\text{fib}(2) + \text{fib}(1)] + \text{fib}(2)] + \text{fib}(3) \]
  \[ = [[1 + 1] + 1] + [\text{fib}(2) + \text{fib}(1)] \]
  \[ = 3 + [1 + 1] \]
  \[ = 5. \]

- The solution is inefficient in terms of time as it repeats the evaluation of \( \text{fib}(n - 2) \) while evaluating \( \text{fib}(n - 1) \).

- For large values of \( n \) too many activation records are generated causing a stack overflow error.
Infinite Recursion

- If the base case is missing, or if a recursive call is made which does not lead to the base case, it may result in infinite recursion.

- Infinite recursion keeps making the recursive call.

- As an example, consider what happens when you make the call fact(-1) in Example 1?
Recursive solutions are simple, elegant and easier to write and understand.

However they are inefficient as compared to the corresponding iterative solutions (which we’ll cover in the next unit) as a certain amount of space is consumed in each recursive call.
Exercises

1. Write a recursive method to find the greatest common divisor (GCD) of two integers \( n \) and \( m \).
2. Write a recursive method to find \( X^n \) given the double \( X \) and the integer \( n \).
3. Consider a Boolean array \( b \) filled with Boolean values. Write a recursive method boolean allTrue() that returns true if all values are true and returns false otherwise.
4. Write a recursive method to print all permutations of a given string.