4 : Recursion

IT 3206 – Data Structures and Algorithms

Level II - Semester 3

Overview

- This section introduces recursion and discusses the differences between iteration and recursion approaches.
- How recursion works and theories behind recursion will be also illustrated in this section.
- Further in this section, different implementations of recursions will be discussed.

Intended Learning Outcomes

- At the end of this lesson, you will be able to;
  - Explain the importance of recursion
  - Classify recursive methods
  - Explain method calls in a recursion
  - Compare different types of recursion

List of Subtopics

4.1. Introduction to Recursion
4.2. Recursion versus iteration
4.3. How recursion works
4.4. Implementation in recursion
    4.4.1. Tail recursion
    4.4.2. Nontail recursion
    4.4.3. Indirect recursion
    4.4.4. Nested Recursion
    4.4.5. Excessive recursion

4.1. Introduction to Recursion

- Any function which calls another instance of the same function is called recursive.
- A recursive method solves a problem by calling another instance of itself to work on a smaller problem. This is called the **recursion step**. The recursion step can result in many more such recursive calls.
- It is important to ensure that the **recursion terminates**. Each time the function calls itself with a slightly simpler version of the original problem. The sequence of smaller problems must eventually converge on the base case.

4.1. Introduction to Recursion Contd.

- Recursion breaks a problem into smaller identical problems.
  - Each recursion call solves an identical but smaller problem.

- Recursion stops the break-down process at a special case whose solution is obvious. That case is called **a base case.**
  - Each recursive call will check the base case condition to eventually stop.
  - Otherwise, recursion falls into an infinite recursion.

4.2. Recursion versus Iteration

Recursion	Iteration
Terminates when a base case is reached.	Terminates when a condition is proven to be false.
Each recursive call requires extra space on the stack frame (memory).	Each iteration does not require extra space.
If we get infinite recursion, the program may run out of memory and result in stack overflow.	An infinite loop could loop forever since there is no extra memory being created.
Solutions to some problems are easier to formulate recursively.	Iterative solutions to a problem may not always be as obvious as a recursive solution.

4.3. How Recursion Works

- When we use recursion, we solve a problem by reducing it to a simpler problem of the same kind.
- We keep doing this until we reach a problem that is simple enough to be solved directly.
- This simplest problem is known as the base case.

```
public static void printSeries(int n1, int n2) {
        if (n1 == n2) {                    // base case
                System.out.println(n2);
        } else {
                System.out.print(n1 + ", ");
                printSeries(n1 + 1, n2);
        }
}
```

4.3. How Recursion Works Contd.

- The base case stops the recursion, because it doesn't make another call to the method.
- If the base case hasn't been reached, we execute the recursive case.

```
public static void printSeries(int n1, int n2) {
        if (n1 == n2) {                     // base case
                System.out.println(n2);
        } else {                            // recursive case
                System.out.print(n1 + ", ");
                printSeries(n1 + 1, n2);
        }
}
```

4.3. How Recursion Works Contd.

- The recursive case:
  - reduces the overall problem to one or more simpler problems of the same kind
  - makes recursive calls to solve the simpler problems

- Structure of a recursive method:

```
recursiveMethod(parameters) {
        if (stopping condition) {
                // handle the base case
        } else {
                // recursive case:
                // possibly do something here
                recursiveMethod(modified parameters);
                possibly do something here
        }
}
```

4.3. How Recursion Works Contd.

- There can be multiple base cases and recursive cases.
- When we make the recursive call, we typically use parameters that bring us closer to a base case.
- We have to ensure that a recursive method will eventually reach a base case, regardless of the initial input.
- Otherwise, we can get infinite recursion, which produces stack overflow.

Example : Factorial

- Let us write a recursive program to perform a popular mathematical calculation.
- Consider the factorial of a positive integer n, written n!
- Which is the product

$$n * (n – 1) * (n – 2) * … * 1$$

with 1! equal to 1 and 0! defined to be 1.

Example : Factorial Contd.

- A recursive declaration of the factorial method is arrived at by observing the following relationship:

$$n! = n * (n – 1)$$

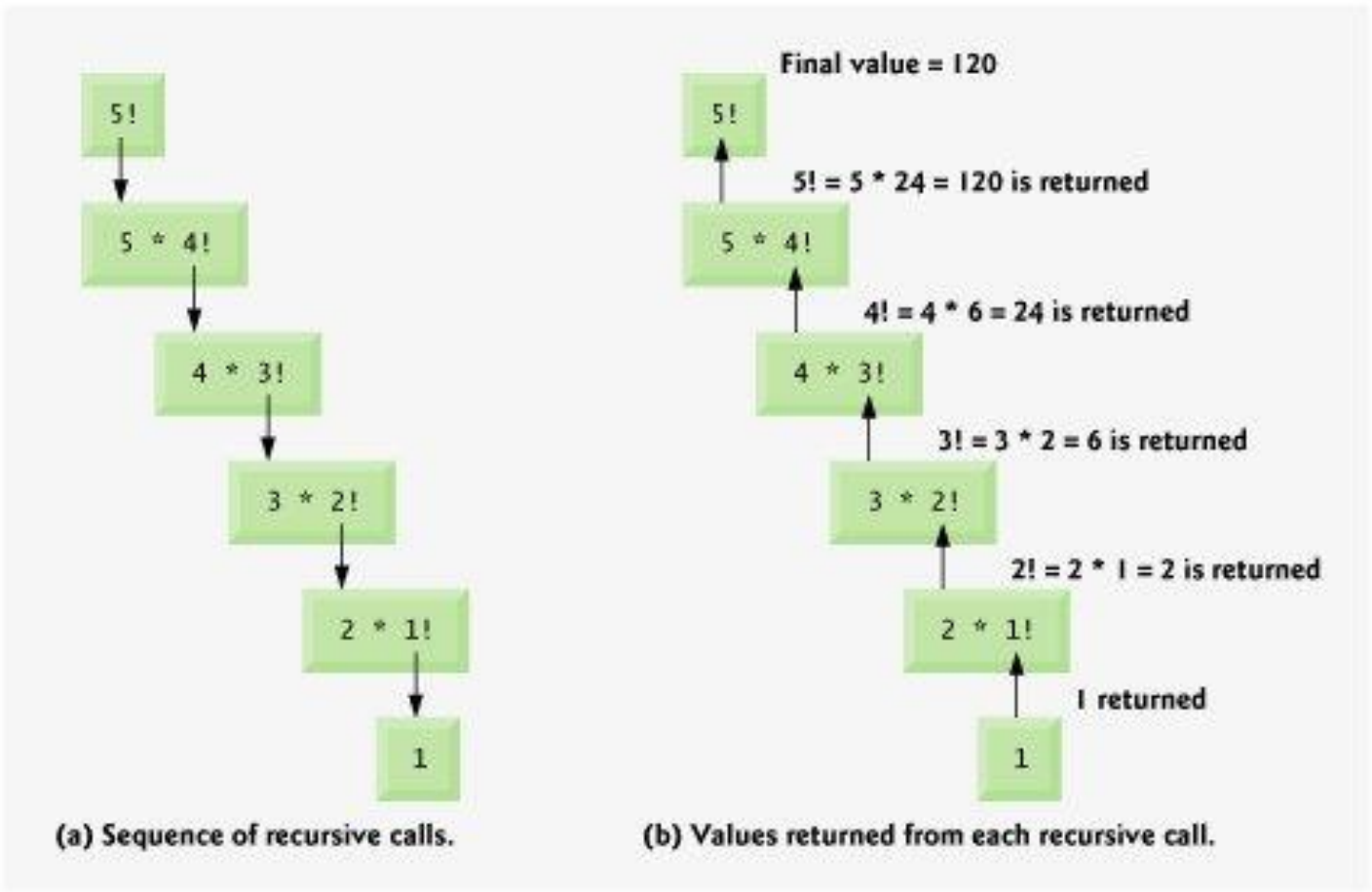
- For an example, 5! can be broken down as follows.

  5! = 5 * 4!
  4! = 4 * 3!
  3! = 3 * 2!
  2! = 2 * 1!

  1! = 1 which is given as a base condition.

Example : Factorial Contd.

- Recursive evaluation of 5!.



Example : Factorial Contd.

- Factorial recursive function pseudo code.

```
Fact(n)
Begin
  if n == 0 or 1 then
    Return 1;
  else
    Return n*Call Fact(n-1); // recursive call
  endif
End
```

Example : Factorial Contd.

- Factorial - recursive function java code.

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

4.4. Implementation in Recursion

- There are several implementations of recursions based on how the recursive call is placed in the function.
  - Tail Recursion
  - NonTail Recursion
  - Indirect Recursion
  - Nested Recursion
  - Excessive Recursion

4.4.1. Tail Recursion

- All recursive definitions contain a reference to a set or function being defined.
- There are, however, a variety of ways such a reference can be implemented.
- This reference can be done in a straightforward manner or in an intricate fashion, just once or many times.
- There may be many possible levels of recursion or different levels of complexity.

4.4.1. Tail Recursion Contd.

- Tail recursion is characterized by the use of only one recursive call at the very end of a method implementation.

- In other words, when the call is made, there are no statements left to be executed by the method; the recursive call is not only the last statement but there are no earlier recursive calls, direct or indirect.

4.4.1. Tail Recursion Contd.

- Example : The method tail() defined as

```
void tail (int i) {
        if (i > 0) {
        System.out.print (i + " ");
        tail (i-1);
    }
}
```

4.4.1. Tail Recursion Contd.

- Tail recursion is simply a glorified loop and can be easily replaced by one.

- In this example, it is replaced by substituting a loop for the if statement and incrementing or decrementing the variable i in accordance with the level of recursion.

- In this way, tail ( ) can be expressed by an iterative method:

```
void iterativeEquivalentOfTail(int i) {
   for ( ; i > 0; i-- )
   System. out. print (i+ "");
}
```

4.4.1. Tail Recursion Contd.

- Is there any advantage in using tail recursion over iteration?
  - For languages such as Java, there may be no compelling advantage, but in a language such as Prolog, which has no explicit loop construct (loops are simulated by recursion), tail recursion acquires a much greater weight.
  - In languages endowed with a loop or its equivalents, such as an if statement combined with a goto statement or labeled statement, tail recursion should not be used.

4.4.2. NonTail Recursion

- Recursions that are not tail recursions are called non- tail recursions.

- Here is a simple non-tail recursive implementation:

```
void reverse() {
   char ch = getChar();
   if (ch != '\n') {
         reverse() ;
         System.out.print(ch);
   }
}
```

4.4.3. Indirect Recursion

- Direct recursion - where a method f( ) called itself.
- f( ) can call itself indirectly via a chain of other calls. For example, f( ) can call g(), and g( ) can call f( ) . This is the simplest case of indirect recursion.
- The chain of intermediate calls can be of an arbitrary length, as in:

  f() → f1() → f2() → …….. → fn() → f()

4.4.3. Indirect Recursion Contd.

- There is also the situation when f( ) can call itself indirectly through different chains.

- Thus, in addition to the chain just given, another chain might also be possible. For instance

  f() → g1() → g2() → …….. → gm() → f()

4.4.3. Indirect Recursion Contd.

- This situation can be exemplified by three methods used for decoding information.
  - **receive()** stores the incoming information in a buffer
  - **decode()** converts it into legible form
  - **store()** stores it in a file

- receive() fills the buffer and calls decode(), which in turn, after finishing its job, submits the buffer with decoded information to store().

- After store() accomplishes its tasks, it calls receive() to intercept more encoded information using the same buffer.

4.4.3. Indirect Recursion Contd.

- Therefore, we have the chain of calls

  receive() → decode() → store() → receive() → ……..

- Above three methods work in the following manner:

```
receive (buffer)
        while buffer is not filled up
                if information is still incoming
                        get a character and store it in buffer;
                else exit( );
        decode (buffer);
```

4.4.3. Indirect Recursion Contd.

```
decode (buffer)
    decode information in buffer;
    store (buffer);

store (buffer)
    transfer information from buffer to file;
    receive (buffer);
```

- As usual in the case of recursion, there has to be an anchor in order to avoid falling into an infinite loop of recursive calls.

4.4.4. Nested Recursion

- A more complicated case of recursion is found in definitions in which a function is not only defined in terms of itself, but also is used as one of the parameters. The following definition is an example of such a nesting:

$$h(n) = \begin{cases} 0 & \text{if } n = 0 \\ N & \text{if } n > 4 \\ h(2 + h(2n)) & \text{if } n <= 4 \end{cases}$$

4.4.4. Nested Recursion Contd.

- Function h has a solution for all n >= 0.

  This fact is obvious for all n > 4 and n = 0, but it has to be proven for n = 1,2,3, and 4.

  Thus, h(2) = h(2 + h(4)) = h(2 + h(2 + h(8))) = 12.

  What are the values of h(n) for n = 1,3, and 4?

4.4.4. Nested Recursion Contd.

- Another example of nested recursion is a very important function originally suggested by Wilhelm Ackermann in 1928 and later modified by Rozsa Peter:

$$A(n,m) = \begin{cases} m+1 & \text{if } n = 0 \\ A(n-1,1) & \text{if } n > 0, m = 0 \\ A(n-1, A(n,m-1)) & \text{otherwise} \end{cases}$$

4.4.4. Nested Recursion Contd.

- Above function is interesting because of its remarkably rapid growth.
- It grows so fast that it is guaranteed not to have a representation by a formula that uses arithmetic operations such as addition, multiplication, and exponentiation.

4.4.4. Nested Recursion Contd.

- To illustrate the rate of growth of the Ackermann function, we need only show that

$$A(3,m) = 2^{m+3} - 3$$

$$A(4,m) = 2^{2^{:2}}{}^{16} - 3$$

- with a stack of m 2s in the exponent;

$$A(4,1) = 2^{2^{16}} - 3 = 2^{65536} - 3$$

  which exceeds even the number of atoms in the universe (which is 10^{80} according to current theories).

- The definition translates very nicely into Java, but the task of expressing it in a non-recursive form is truly troublesome.

4.4.5. Excessive Recursion

- Logical simplicity and readability are used as an argument supporting the use of recursion.

- The price for using recursion is slowing down execution time and storing on the run-time stack more things than required in a non-recursive approach.

- If recursion is too deep (for example, computing $5.6^{100'000}$), then we can run out of space on the stack and our program terminates abnormally by raising an unrecoverable StackOverflowError.

4.4.5. Excessive Recursion Contd.

- But usually, the number of recursive calls is much smaller than 100,000, so the danger of overflowing the stack may not be imminent. However, if some recursive function repeats the computations for some parameters, the run time can be prohibitively long even for very simple cases.

- Consider Fibonacci numbers. A sequence of Fibonacci numbers is defined as follows:

$$\text{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{if } n >= 2 \end{cases}$$

4.4.5. Excessive Recursion Contd.

- The definition states that if the first two numbers are 0 and 1, then any number in the sequence is the sum of its two predecessors. But these predecessors are in turn sums of their predecessors, and so on, to the beginning of the sequence. The sequence produced by the definition is

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

4.4.5. Excessive Recursion Contd.

- How can this definition be implemented in Java? It takes almost term-by-term translation to have a recursive version, which is

```
int Fib(int n) {
        if (n < 2)
                return n;
        else return Fib(n-2) + Fib(n-l);
}
```

- The method is simple and easy to understand but extremely inefficient fibonacci heap.

Summary

Introduction to Recursion	Any function which calls itself is called recursive.
Recursion versus Iteration	Solutions to some problems are easier to formulate recursively than iteratively.
How Recursion Works	Recursion solve a problem by reducing it to a simpler problem of the same kind.
Implementation in Recursion	Tail recursion, Nontail recursion, Indirect recursion, Nested Recursion, Excessive recursion