Lab 3
Recursion

Quang D. C.
dcquang@it.tdt.edu.vn

September 07, 2020

In this tutorial, we will approach Recursion thinking. After completing this tutorial, you can apply recursive algorithm to solve some problems.

1. Introduction

A **recursive algorithm** is an algorithm which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input. Generally, if a problem can be solved utilizing solutions to smaller versions of the same problem, and the smaller versions reduce to easily solvable cases, then we can use a recursive algorithm to solve that problem. [1]

Every recursive algorithm involves at least two cases [2]:

- **base case:** The simple case; an occurrence that can be answered directly; the case that recursive calls reduce to.
- **recursive case:** a more complex occurrence of the problem that cannot be directly answered, but can be described in terms of smaller occurrences of the same problem

```
recursiveFunction(){
    if(simpleCase){
        Calculate answers directly
    }
    else{
        Call recursiveFunction() on each subproblem
    }
}
```

In the next section, we will make some recursive function to calculate **Factorials** and **Fibonacci series**.

2. Recursive function

2.1. Recursively calculating factorials

The factorial of a non-negative integer number, denoted by n, is defined as follows:

$$n! = n(n-1)(n-2)...1 = n(n-1)!$$

In the above formula, the second section defines how-to calculate factorial of n in an iterative version, thus we can use the for statement as follows:

```
float factorial = 1;
for(i = 2; i <= n; i++)
{
    factorial = factorial * i;
}
```

From the for statement, we can easily see that each time the statement inside the loop executes, it takes the last factorial result and multiplies it with the new i value, thus we can define the factorial of n as the third part in the formula, that is, $5! = 5 * 4 * 3 * 2 * 1 = 5 * 4! = 120$. The evaluation of 5! would proceed as in *Figure 1*, in the left figure, the recursive call executes until 1!, and it returns 1 (the base case) and terminates the recursion. In the left figure, it shows the values returned from each recursive call to its caller until the final value is calculated and returned.

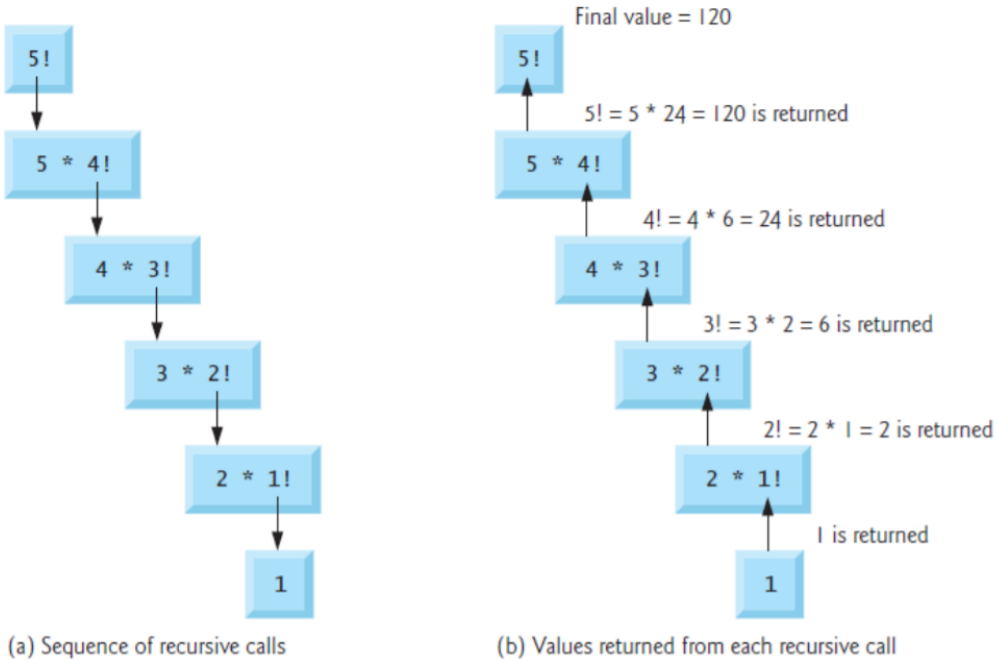

Figure 1: Recursive evaluation of 5!

The program below will illustrate how-to program a recursive function.

```
public class Recursion {
    public static double computeFactorial(int n)
    {
        if(n == 0 || n == 1) return 1;
        return n * computeFactorial(n - 1);
    }
    public static void main(String[] args)
    {
        System.out.println(computeFactorial(5));
    }
}
```

2.2. Fibonacci series

The Fibonacci series begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers. The Fibonacci series may be defined recursively as follows:

$$fibonacci(0) = 0$$
$$fibonacci(1) = 1$$
$$fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)$$

From the above formula, calculates the n^{th} Fibonacci number recursively using function fibonacci. Notice that Fibonacci numbers tend to become large quickly, so we use data type float for the parameter type and the return type also.

```
1  public class Recursion {
2      public static double fibonacci(int n)
3      {
4          if(n == 0) return 0;
5          if(n == 1) return 1;
6          return fibonacci(n - 1) + fibonacci(n - 2);
7      }
8      public static void main(String[] args)
9      {
10         System.out.println(fibonacci(10));
11     }
12 }
```

Each time the fibonacci function is invoked, it immediately tests for the base case - n is equal to 0 or 1. If this is true, n is returned. Interestingly, if n is greater than 1, the recursion step generates two recursive calls, each solves a slightly simpler problem than the original call to fibonacci. *Figure 2* shows how the function fibonacci evaluates when n = 3.

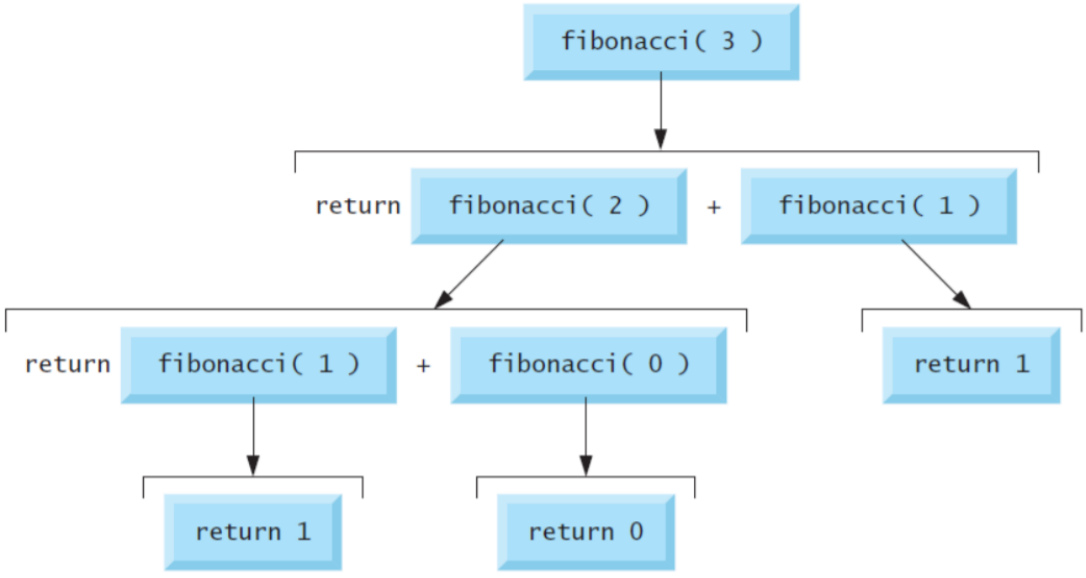

Figure 2: Evaluating fibonacci(3)

3. Recursion vs. Iteration

We have studied two kinds of functions which implemented in either recursive or iteratively. In this section, we compare the two approaches and discuss why you might choose one approach over the other in a particular situation.

- Both iteration and recursion are based on a control structure: Iteration uses a *repetition structure*; recursion uses a *selection structure*.
- Both iteration and recursion involve *repetition*: Iteration explicitly uses a *repetition statement*; recursion achieves repetition through *repeated function calls*.
- Iteration and recursion each involve a*termination test*: Iteration terminates when the *loop-continuation condition fails*; recursion when a *base case is recognized.*
- *Iteration with counter-controlled repetition and recursion each gradually approach termination: Iteration keeps modifying a counter until the counter assumes a value that makes the loop-continuation condition fail; recursion keeps producing simpler versions of the original problem until the base case is reached.*
- *Both iteration and recursion can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem each time in a manner that converges on the base case.*

4. Excercise

Exercise 1

Using iteration, define the functions to:

(a) Compute factorial of n.

(b) Compute x^n.

(c) Count the number of digits of a positive integer number.

(d) Check an positive integer number is prime or not.

(e) Find the Greatest Common Divisor (GCD) of 2 positive integer numbers. *(Using Euclid algorithm)*

Exercise 2

Using recursion, define the functions to:

(a) Compute factorial of n.

(b) Compute x^n.

(c) Count the number of digits of a positive integer number.

(d) Find the Greatest Common Divisor (GCD) of 2 positive integer numbers. *(Using Euclid algorithm)*

Exercise 3

Define a recursive function **public static boolean checkPrime(int n, int d)** that check whether a number is prime *(d is current divisor to check)*

Exercise 4

Define a recursive function to calculate the following expressions:

(a) $\sum_{i=1}^{n}(2i+1)$

(b) $\sum_{i=1}^{n}(i!)$

(c) $\prod_{i=1}^{n}(i!)$

(d)

$$C(n,k) = \begin{cases} \text{n(n-1)(n-2)}\cdots\text{(n-r+1)}, & \text{n >= r > 0} \\ 1, & \text{otherwise} \end{cases}$$

(e)

$$P(n) = \begin{cases} 2^n + n^2 + P(n-1), & \text{n} > 1 \\ 3, & \text{n} = 1 \end{cases}$$

Exercise 5

Define a recursive function to convert a Decimal number to Binary. *(The output must be an integer number. Example with input is 8, output must be an integer number 1000).*

Exercise 6

Using iteration, define the functions to:

(a) Find and return the minimum element in an array. The array and its size are given as parameters.

(b) Find and return the minimum element in an array. The array and its size are given as parameters.

(c) Count how many even numbers are there in an array. The array and its size are given as parameters.

Exercise 7

Using recursion, define the functions to:

(a) Find and return the minimum element in an array. The array and its size are given as parameters.

(b) Find and return the minimum element in an array. The array and its size are given as parameters.

(c) Count how many even numbers are there in an array. The array and its size are given as parameters.

Exercise 8

Using Linked List in **Lab 1** for this exercise.

(a) Implement method *addSortedList(E item)* to insert new element to a sorted linked list, that means we have to find the first node whose value is bigger than item and insert before it.

(b) Suppose we have a linked list contains integer numbers, do the following requirements:

- Count all even numbers.
- Sum all numbers.

5. Reference

[1]. https://www.cs.odu.edu/ toida/nerzic/content/recursive$_a lg/rec_a lg.html$

[2]. https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1178/lectures/7-IntroToRecursion/7-IntroToRecursion.pdf