

## Introduction to Algorithms

Any program comes created to resolve a problem and we should know how to approach a problem to obtain its solution with a program. To this point the concept of algorithm is fundamental.

An **algorithm** is any well-defined computational procedure that takes one value or a set of values, called **input** and produces some value, or set of values, called **output**. An algorithm is thus a sequence of computational steps that transform the input into the output. The algorithm concept represents a systematic approach to obtain the solution of a problem. The approach consists in the description of a series of passages that you do not involve conjectures (that is suppositions or reasoning based on probability). Like general rules, an algorithm:

1. does not have to be ambiguous. That is the steps have to be immediately obvious to who will be called to apply the algorithm, that is to say its processor. Besides in the algorithm one should indicate that it does for each step and when a step is ended where goes;
2. have to be executable that is to say each passage is able to perform concretely;
3. should arrive at a conclusion. It is not an algorithm the description of passages that follow themselves infinitely. Having an algorithm, then we can arrive at the solution also with the computer.

Conclusion: Algorithm is an essential prerequisite to be able to solve a problem: it is a tool for solving a well-specified **computational problem**.

The statement of the problem specifies in general terms the desired input/output relationship. In other words the algorithm describes a specific computational procedure for achieving that input/output relationship.

If we need to sort a sequence of numbers into non-decreasing order we can formally define the **sorting problem**: as in the followings

**Input:** A sequence of  $n$  numbers  $\langle a_1, a_2, \dots, a_n \rangle$ .

**Output:** A permutation of the input sequence such that  $a_1 \leq a_2 \leq \dots \leq a_n$

For example, given the input sequence  $\langle 45, 32, 52, 22, 99, 46 \rangle$ , a sorting algorithm returns as output the sequence  $\langle 22, 32, 45, 46, 52, 99 \rangle$ . Such an input sequence is called an **instance** of the sorting problem.

Sorting is a fundamental operation in computer science. This is why a large number of sorting algorithms exist. Selection of the best algorithms for a given application depends on the number of items to be sorted, if the items are already somewhat sorted, possible restrictions on the item values, and the kind of storage device to be used for storing the instance.

An algorithm is said to be **correct** if, for every input instance, it halts with the correct output.

We say that a correct algorithm **solves** the given computational problem.

## Recursion

Recursion is a algorithmic technique in which, to resolve its task and to deliver a result, a function calls itself in order to accomplish a part of the comprehensive task. It is noted that the fundamental concept in the recursion setting is that of to ascertain that a call to the same function, if done on a simpler task, is correct and takes us to get the result in finite time. A solution recursive, that is to say based on an algorithm recursive, requires two parts or cases. The second part also was three appearance distinguished:

- **Base Part**, in which special or base cases are identifies, where the problem is sufficiently simple to offer an immediate solution;

- **Recursive Part** with the three aspects:

1. the problem is divided in simpler or smaller parts of the problem,
2. the function is recursively called on each identified part and, finally,
3. the solutions of the varied parts are combined in a comprehensive solution of the problem.

The schema of a recursive algorithm is like to the following procedure:

```
public void calculates (list of variables);  
    if (base, simple, or trivial case)
```

```

    {some processing of the results}
else {
    recursive calls of calculates for smaller values of the variables;
    eventually other calls
}

```

## Factorial

Factorial is a mathematic function mathematics. Factorial of a number  $n$  is equal to the product of all integer from 1 to  $n$ . The function factorial of  $n$  is denoted with  $n! = 1 \times 2 \times 3 \dots \times n$  and it can be calculated in a cycle like this:

```

factorial = 1;
for (int i = 1; i <= n; i++)
    factorial *= i;
System.out.println(n+"! = " + factorial);

```

It is possible to propose a recursive definition of the function Factorial:

$$n! = \begin{cases} 1, & n = 1; \\ n * (n-1)!, & n > 1 \end{cases}$$

The Java code which implements the recursive algorithm Factorial is the following:

```

public static int fattoriale(int n) {
    if (n==1)
        return 1;
    else
        return fattoriale(n-1) * n;
}

```

An important thing that we should have always present when we plan a function recursive is that of the final condition, that is the condition that allows us to go out from execution rather than going to infinite.

They exist more manners to know when to finish l' execution. The simplest manner is to use an instruction `if` like in the previous example that captures the event that  $n$  became 1. It is observed that if  $n$  was smaller of 1, the method would have been performed to infinite.

Therefore, the condition in the instruction `if` changes in:  $n \leq 1$ .

## Final Recursion

The final recursion is a special case of recursion in which the last operation of the algorithm is a recursive call of the same algorithm. An algorithm with final recursion could be optimized to save memory.

Let consider an algorithm that calculates the utmost in a list of elements having like elements numbers greater of a certain number `maxIni`. The algorithm uses a recursive definition of the maximum of several numbers:

$$\text{maxList}(\text{list}, \text{maxIni}) = \begin{cases} \text{maxList}, & \text{if the list is empty} \\ \text{maxList}(\text{list}', a) \\ \text{maxList}(\text{list}', \text{maxIni}) \end{cases}$$

where `a` is the first element in `list` and `list'` is an identical list with `list` but without the first element. The following algorithm calculates the maximum in a Java list of Integer objects:

```

public static int maxList(List list, int maxIni) {
    if (null == list) {
        return maxIni;
    }
    if (maxIni < ((Integer)list.getFirst()).intValue()) {
        return maxList(list.sublist(1, list.length()),
            ((Integer)list.getFirst()).intValue());
    }else {

```

```

        return maxList(list.sublist(1, list.length()), maxIni);
    }
}

```

A more complex case is that of the recursive computation of the greater common divisor (MDC) of two positive integers. The algorithm is designed on basis of the following rules.

```

MDC(2m, 2n) = 2 * MDC (m, n)
MDC(2m, 2n+1) = MDC (m, 2n+1)
MDC(2m+1, 2n+1) = MDC(n-m, 2m+1), if m < n
MDC(m, m) = m

```

The following algorithm computes the MDC of two positive integers and uses the final recursion.

```

public int mdc(int m, int n) {
    if (m==n)
        return m;
    switch (m%2 + n%2) {
        case 0:
            return 2 * mdc(m/2, n/2);
        case 1:
            if (m%2 == 0)
                return mdc(m/2, n);
            else
                return mdc(m, n/2);
        case 2:
            return m>n?mdc(m-n, n):mdc(m, n-m);
    }
    return 1;
}

```

## Multiple Recursion

When a method calls itself in recursive manner more than of one time the method is said to contain a *multiple recursion*. The more known example is that of the numbers of Fibonacci that come definite ricorsivamente so: The numbers of Fibonacci generate themselves in a sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34,.

To calculate a number Fibonacci is necessary to know its index in the sequence of the numbers and apply recursively the same algorithm:

## Advanced algorithm

The following program introduces a class SortedList that defines an ArrayList with the possibility of to order its elements for means of a method sort (). The method realizes sorting using a complex, recursive algorithm, quickSort () :

```

class SortedList extends ArrayList {
    private Compare compare;
    public SortedListVector(Compare compare){
        this.compare = compare;
    }
    public void sort() {
        quickSort(0, size()-1);
    }
    private void quickSort(int left, int right) {
        if(right > left) Object o = get(right);
        int i = left - 1;
        int j = right;
        while(true) {
            while(compare.lessThan(get(++i), o)) ;

```

```
        while(j > 0)
            if(compare.lessThanOrEqualTo(get(--j), o))
                break; // exit while(j > 0)
            if(i >= j) break; // exit while(true)
            swap(i, j);
        }
        swap(i , right);
        quickSort(left, i-1);
        quickSort(i+1, right);
    }
private void swap(int loc1, int loc2) {
    Object tmp = get(loc1);
    set(get(loc2), loc1);
    set(tmp, loc2);
}
}
}
```