Universidad Carlos III de Madrid

Algorithms and Data Structures (ADS)
Bachelor in Informatics Engineering
Computer Science Department

Algorithm Analysis.

Authors: Juan Perea
Isabel Segura Bedmar

April 2011
Departamento de Informtica,
Laboratorio de Bases de Datos Avanzadas (LaBDA)
http://labda.inf.uc3m.es/

Recursion

Most of the information has been sourced from the books [1].

Some algorithms and mathematical functions can be defined in a recursive way. This happens when the algorithm or function being defined is used in its own definition. For example, the factorial function of an integer number N can be defined in an iterative way as the product of all the positive integers that are less than or equal to N:

$$N! = 1 * 2 * 3 * \ldots * (N-1) * N \tag{1}$$

Since $(N-1)! = 1 * 2 * 3 * \ldots * (N-1)$, it turns out that:

$$N! = (N-1)!N \tag{2}$$

This definition, while correct, is still incomplete, as it doesn't say how to calculate the factorial of 0 (or 1). So we need to define a base case, at which the recursion will stop. The complete recursive definition of the factorial function is:

$$0! = 1 \tag{3}$$

$$N! = (N-1)!N, \quad for\ N > 0 \tag{4}$$

Another simple recursive algorithm is the Euclidean algorithm[1] to calculate the greatest common divisor of two numbers a, b (being a¿=b), which can be recursively defined as:

$$gcd(a,b) = a \qquad if\ b = 0 \tag{5}$$

$$gcd(a,b) = gcd(b, amodb), \quad otherwise \tag{6}$$

In this example, it's not so obvious that recursion reduces the problem towards the base case, but we still can see that the numbers become smaller in each iteration.

Finally, another typical example of a recursive function is the definition of the Fibonacci numbers[2], formed by a series of numbers, starting by 0, 1, in

[1] http://en.wikipedia.org/wiki/Euclidean_algorithm
[2] http://en.wikipedia.org/wiki/Fibonacci_number

which each number is obtained by summing the previous two numbers in the series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... In this case, the recursive definition is much clearer than an iterative one:

$$Fib(0) = 0 \tag{7}$$
$$Fib(1) = 1 \tag{8}$$
$$Fib(N) = Fib(N-2) + Fib(N-1), \quad for\ N > 1 \tag{9}$$

As a conclusion, the definition of a function of algorithm is recursive when it is formed by:

1. A set of one or more simple base cases, to stop recursion.
2. A set of one or more rules that reduce complex cases towards the base case(s).

It is important -and not always easy- to guarantee that the recursive algorithm will not end up in an infinite loop.

Loop equivalence and examples

Almost all recursive algorithms can be solved in an iterative way, using 'for' or 'while' loops. In some cases, the recursive solution will be more elegant and easier to understand and implement. In other cases, the iterative solution should be chosen (mostly due to the limitations exposed in the next section). Some code for the recursive algorithms mentioned in the introduction follows, along with its equivalent iterative algorithm:

Factorial

Both implementations are equally simple and elegant.

```
static long factorialRec(int n) {
            if (n ¡ 2) {
              return 1;
            }else {
              return n * factorialRec(n  1);
            }
}

static long factorialIt(int n) {
            long fact = 1;
            (for int i=2; i¡=n; i++) fact*=i;
            return fact;
}
```

Euclidean algorithm

In this case, the recursive implementation looks more elegant than the iterative solution, as the latter even needs an auxiliary variable to avoid problems, making the code more obfuscated:

```
static long euclideanIt(long a,long b) {
        while (b¿0){
                long aux=a;
                a=b;
                b=aux%b;
        }
}

static long euclideanRec(long a,long b) {
        if (b==0) {
          return a;
        }else {
          return euclideanRec(b,a%b);
        }
}
```

Limitations

However, being in most cases more elegant than the equivalent iterative solution, recursive algorithms must be used with care, as they have some limitations.

Stack usage and overflow

Computers use an execution stack (also known as call stack[3]) to store some necessary information related to all the running functions (from main to the currently running function). For each running function, this information includes the returning address (the address into the caller function to which execution should return when the function finishes) as well as local variables and parameters. Execution stack size is normally a quite limited resource (sometimes a few kB). This is normally enough for most applications, but an uncontrolled (or not too well estimated) recursive algorithm can easily cause a crash in the form of a stack overflow exception. If we take a look back at the examples related to the Euclidean algorithm, the iterative implementation will make a constant use of the execution stack. No matter the number of iterations, the stack will only hold one copy of the parameters 'a' and 'b' of the local variable 'aux' (as well, of course, as the returning address). But if we take the recursive algorithm, for each recursive call, the system will create a copy in the stack of the parameters parameters 'a' and 'b' and of the returning address. This is not a problem in the case of the Euclidean algorithm, is it's normally resolved in a few iterations,

[3] http://en.wikipedia.org/wiki/Call_stack

but can really be an issue in other recursive algorithms that need more recursive calls and/or more memory space for each iteration. For example, calculating 'fibonacciRec(5000)' will cause a stack overflow with a stack size of 32kB (4 bytes for 'n' 4 bytes for the returning address, in a 32 bit architecture, multiplied by 4999 calls, this makes 39992 bytes) issues. One typical example is the calculation of the first N Fibonacci numbers. The iterative implementation has a linear complexity (it's O(n)), while the recursive implementation has an exponential complexity (it's O(2n)). This happens because fibonacciRec(i) will be called from fibonacciRec(i+1) and from fibonacciRec(i+2). What's more, fibonacciRec(i+1) will be called from fibonacciRec(i+2) and from fibonacciRec(i+3), and so on, which means fibonnaciRec(i) will be called a total number of 2n-i times, causing the recursive implementation to be completely inefficient.

Cases of use

However, being aware of the mentioned limitations, there are cases in which the recursive solution should be considered. A complete discussion can be found here1. In this chapter, we will only mention some typical paradigms. There are no systematic approaches to neither of these paradigms, it takes some time and practice to understand and master them, however it's important to know they can be considered for some kinds of problems.

Divide and conquer strategy

A divide and conquer recursive algorithm[4] will break the problem down in several subproblems of the same type but with a reduced size, until the problem is so simple that it can be directly solved. Some examples of efficient divide and conquer algorithms are: Array sorting algorithms such as quicksort[5] and mergesort[6]. These algorithms split the array in several parts, and then call themselves recursively to sort each of these parts. The base case for both algorithms is an one-sized array. Gaming algorithms like the towers of Hanoi[7]. In this well-known game, we assume that, if we know how to move a tower formed by N disks from stack A to stack B using stack C as an auxiliary stack, then moving a tower formed by N+1 disks is as simple as moving the top N disks from stack A to the auxiliary stack C, then move disk N+1 from A to B, and then move again the top N disks from the auxiliary stack C over the disk already moved to stack B. The base case in this algorithm is when N=1, and we only need to move the disk from stack A to stack B.

[4] http://en.wikipedia.org/wiki/Fibonacci_number
[5] http://en.wikipedia.org/wiki/Quicksort
[6] http://en.wikipedia.org/wiki/Mergesort
[7] http://en.wikipedia.org/wiki/Hanoi_towers

Backtracking

Backtracking algorithms[8] are useful for solving certain computational problems when, at a given point of the problem, there are several ways to follow up, some of which may lead to a solution, and some of which may not. They are specially useful when the problem has several solutions, and we are looking for all them.

[8] http://en.wikipedia.org/wiki/Backtracking

Bibliography

[1] M. Goodrich and R. Tamassia, *Data structures and algorithms in Java*. Wiley-India, 2009.