CHAPTER 3

# Algorithms

## 3.1. Algorithms

Consider the following list of instructions to find the maximum of three numbers $a, b, c$:

1. Assign variable $x$ the value of $a$.
2. If $b > x$ then assign $x$ the value of $b$.
3. If $c > x$ then assign $x$ the value of $c$.
4. Output the value of $x$.

After executing those steps the output will be the maximum of $a, b, c$.

In general an algorithm is a finite list of instructions with the following characteristics:

1. *Precision.* The steps are precisely stated.
2. *Uniqueness.* The result of executing each step is uniquely determined by the inputs and the result of preceding steps.
3. *Finiteness.* The algorithm stops after finitely many instructions have been executed.
4. *Input.* The algorithm receives input.
5. *Output.* The algorithm produces output.
6. *Generality.* The algorithm applies to a set of inputs.

Basically an algorithm is the idea behind a program. Conversely, programs are implementations of algorithms.

**3.1.1. Pseudocode.** Pseudocode is a language similar to a programming language used to represent algorithms. The main difference respect to actual programming languages is that pseudocode is not required to follow strict syntactic rules, since it is intended to be just read by humans, not actually executed by a machine.

Usually pseudocode will look like this:

```
procedure ProcedureName(Input)
  Instructions...
end ProcedureName
```

For instance the following is an algorithm to find the maximum of three numbers $a, b, c$:

```
1: procedure max(a,b,c)
2:   x := a
3:   if b>x then
4:     x := b
5:   if c>x then
6:     x := c
7:   return(x)
8: end max
```

Next we show a few common operations in pseudocode.

The following statement means "assign variable $x$ the value of variable $y$:

```
x := y
```

The following code executes "action" if condition "p" is true:

```
if p then
  action
```

The following code executes "action1" if condition "p" is true, otherwise it executes "action2":

```
if p then
  action1
else
  action2
```

The following code executes "action" while condition "p" is true:

```
while p do
  action
```

The following is the structure of a for loop:

```
for var := init to limit do
  action
```

If an action contains more than one statement then we must enclose them in a block:

```
begin
  Instruction1
  Instruction2
  Instruction3
  ...
end
```

Comments begin with two slashes:

```
// This is a comment
```

The output of a procedure is returned with a return statement:

```
return(output)
```

Procedures that do not return anything are invoked with a call statement:

```
call Procedure(arguments...)
```

As an example, the following procedure returns the largest number in a sequence $s_1, s_2, \ldots s_n$ represented as an array with $n$ elements: `s[1]`, `s[2]`,..., `s[n]`:

```
1: procedure largest_element(s,n)
2:   largest := s[1]
3:   for k := 2 to n do
4:     if s[k] > largest then
5:       largest := s[k]
6:   return(largest)
7: end largest_element
```

### 3.1.2. Recursiveness.

*Recursive Definitions.* A definition such that the object defined occurs in the definition is called a *recursive definition.* For instance,

consider the *Fibonacci sequence*

$$0, 1, 1, 2, 3, 5, 8, 13, \ldots$$

It can be defined as a sequence whose two first terms are $F_0 = 0$, $F_1 = 1$ and each subsequent term is the sum of the two previous ones: $F_n = F_{n-1} + F_{n-2}$ (for $n \geq 2$).

Other examples:

- Factorial:
  1. $0! = 1$
  2. $n! = n \cdot (n-1)!$ $\qquad (n \geq 1)$
- Power:
  1. $a^0 = 1$
  2. $a^n = a^{n-1}\, a$ $\qquad (n \geq 1)$

In all these examples we have:

1. A *Basis*, where the function is explicitly evaluated for one or more values of its argument.
2. A *Recursive Step*, stating how to compute the function from its previous values.

*Recursive Procedures.* A *recursive procedure* is a procedure that invokes itself. Also a set of procedures is called *recursive* if they invoke themselves in a circle, e.g., procedure $p_1$ invokes procedure $p_2$, procedure $p_2$ invokes procedure $p_3$ and procedure $p_3$ invokes procedure $p_1$. A *recursive algorithm* is an algorithm that contains recursive procedures or recursive sets of procedures. Recursive algorithms have the advantage that often they are easy to design and are closer to natural mathematical definitions.

As an example we show two alternative algorithms for computing the factorial of a natural number, the first one iterative (non recursive), the second one recursive.

```
1: procedure factorial_iterative(n)
2:   fact := 1
3:   for k := 2 to n do
4:     fact := k * fact
5:   return(fact)
6: end factorial_iterative
```

```
1: procedure factorial_recursive(n)
2:   if n = 0 then
3:     return(1)
4:   else
5:     return(n * factorial_recursive(n-1))
6: end factorial_recursive
```

While the iterative version computes $n! = 1 \cdot 2 \cdot \ldots n$ directly, the recursive version resembles more closely the formula $n! = n \cdot (n-1)!$

A recursive algorithm must contain at least a basic case without recursive call (the case $n = 0$ in our example), and any legitimate input should lead to a *finite* sequence of recursive calls ending up at the basic case. In our example $n$ is a legitimate input if it is a natural number, i.e., an integer greater than or equal to 0. If $n = 0$ then `factorial_recursive(0)` returns 1 immediately without performing any recursive call. If $n >$ then the execution of

```
factorial_recursive(n)
```

leads to a recursive call

```
factorial_recursive(n-1)
```

which will perform a recursive call

```
factorial_recursive(n-2)
```

and so on until eventually reaching the basic case

```
factorial_recursive(0)
```

After reaching the basic case the procedure returns a value to the last call, which returns a value to the previous call, and so on up to the first invocation of the procedure.

Another example is the following algorithm for computing the $n$th element of the Fibonacci sequence:

```
1: procedure fibonacci(n)
2:   if n=0 then
3:     return(0)
4:   if n=1 then
5:     return(1)
6:   return(fibonacci(n-1) + fibonacci(n-2))
7: end fibonacci
```

In this example we have two basic cases, namely $n = 0$ and $n = 1$.

In this particular case the algorithm is inefficient in the sense that it performs more computations than actually needed. For instance a call to `fibonacci(5)` contains two recursive calls, one to `fibonacci(4)` and another one to `fibonacci(3)`. Then `fibonacci(4)` performs a call to `fibonacci(3)` and another call to `fibonacci(2)`, so at this point we see that `fibonacci(3)` is being called twice, once inside `fibonacci(5)` and again in `fibonacci(4)`. Hence sometimes the price to pay for a simpler algorithmic structure is a loss of efficiency.

However careful design may yield efficient recursive algorithms. An example is *merge_sort*, and algorithm intended to sort a list of elements. First let's look at a simple non recursive sorting algorithm called *bubble_sort*. The idea is to go several times through the list swapping adjacent elements if necessary. It applies to a list of numbers $s_i, s_{i+1}, \ldots, s_j$ represented as an array `s[i], s[i+1],..., s[j]`:

```
1: procedure bubble_sort(s,i,j)
2:   for p:=1 to j-i do
3:     for q:=i to j-p do
4:       if s[q] > s[q+1] then
5:         swap(s[q],s[q+1])
6: end bubble_sort
```

We can see that `bubble_sort` requires $n(n-1)/2$ comparisons and possible swapping operations.

On the other hand, the idea of `merge_sort` is to split the list into two approximately equal parts, sort them separately and then merge them into a single list:

```
 1: procedure merge_sort(s,i,j)
 2:   if i=j then
 3:     return
 4:   m := floor((i+j)/2)
 5:   call merge_sort(s,i,m)
 6:   call merge_sort(s,m+1,j)
 7:   call merge(s,i,m,j,c)
 8:   for k:=i to j do
 9:     s[k] := c[k]
10: end merge_sort
```

The procedure `merge(s,i,m,j,c)` merges the two increasing sequences $s_i, s_{i+1}, \ldots, s_m$ and $s_{m+1}, s_{m+2}, \ldots, s_j$ into a single increasing sequence $c_i, c_{i+1}, \ldots, c_j$. This algorithm is more efficient than `bubble_sort` because it requires only about $n \log_2 n$ operations (we will make this more precise soon).

The strategy of dividing a task into several smaller tasks is called *divide and conquer*.

**3.1.3. Complexity.** In general the *complexity* of an algorithm is the amount of time and space (memory use) required to execute it. Here we deal with *time complexity* only.

Since the actual time required to execute an algorithm depends on the details of the program implementing the algorithm and the speed and other characteristics of the machine executing it, it is in general impossible to make an estimation in actual physical time, however it is possible to measure the length of the computation in other ways, say by the number of operations performed. For instance the following loop performs the statement `x := x + 1` exactly $n$ times,

```
1: for i := 1 to n do
2:   x := x + 1
```

The following double loop performs it $n^2$ times:

```
1: for i := 1 to n do
2:   for j := 1 to n do
3:     x := x + 1
```

The following one performs it $1 + 2 + 3 + \cdots + n = n(n+1)/2$ times:

```
1: for i := 1 to n do
2:   for j := 1 to i do
3:     x := x + 1
```

Since the time that takes to execute an algorithm usually depends on the input, its complexity must be expressed as a function of the input, or more generally as a function of the *size* of the input. Since the execution time may be different for inputs of the same size, we define the following kinds of times:

1. *Best-case time*: minimum time needed to execute the algorithm among all inputs of a given size $n$.
2. *Wost-case time*: maximum time needed to execute the algorithm among all inputs of a given size $n$.
3. *Average-case time*: average time needed to execute the algorithm among all inputs of a given size $n$.

For instance, assume that we have a list of $n$ objects one of which is colored red and the others are colored blue, and we want to find the one that is colored red by examining the objects one by one. We measure time by the number of objects examined. In this problem the minimum time needed to find the red object would be 1 (in the lucky event that the first object examined turned out to be the red one). The maximum time would be $n$ (if the red object turns out to be the last one). The average time is the average of all possible times: $1, 2, 3, \ldots, n$, which is $(1+2+3+\cdots+n)/n = (n+1)/2$. So in this example the best-case time is 1, the worst-case time is $n$ and the average-case time is $(n+1)/2$.

Often the exact time is too hard to compute or we are interested just in how it grows compared to the size of the input. For instance and algorithm that requires exactly $7n^2 + 3n + 10$ steps to be executed on an input of size $n$ is said to be or *order* $n^2$, represented $\Theta(n^2)$. This justifies the following notations:

*Big Oh Notation.* A function $f(n)$ is said to be of order at most $g(n)$, written $f(n) = O(g(n))$, if there is a constant $C_1$ such that

$$|f(n)| \leq C_1|g(n)|$$

for all but finitely many positive integers $n$.

*Omega Notation.* A function $f(n)$ is said to be of order at least $g(n)$, written $f(n) = \Omega(g(n))$, if there is a constant $C_2$ such that

$$|f(n)| \geq C_2|g(n)|$$

for all but finitely many positive integers $n$.

*Theta Notation.* A function $f(n)$ is said to be of order $g(n)$, written $f(n) = \Theta(g(n))$, if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

*Remark*: All logarithmic functions are of the same order: $\log_a n = \Theta(\log_b n)$ for any $a, b > 1$, because $\log_a n = \log_b n / \log_b a$, so they always differ in a multiplicative constant. As a consequence, if the execution time of an algorithm is of order a logarithmic function, we can just say that its time is "logarithmic", we do not need to specify the base of the logarithm.

The following are several common growth functions:

| Order | Name |
|---|---|
| $\Theta(1)$ | Constant |
| $\Theta(\log \log n)$ | Log log |
| $\Theta(\log n)$ | Logarithmic |
| $\Theta(n \log n)$ | n log n |
| $\Theta(n)$ | Linear |
| $\Theta(n^2)$ | Quadratic |
| $\Theta(n^3)$ | Cubic |
| $\Theta(n^k)$ | Polynomial |
| $\Theta(a^n)$ | Exponential |

Let's see now how we find the complexity of algorithms like `bubble_sort` and `merge_sort`.

Since `bubble_sort` is just a double loop its complexity is easy to find; the inner loop is executed

$$(n-1) + (n-2) + \cdots + 1 = n(n-1)/2$$

times, so it requires $n(n-1)/2$ comparisons and possible swap operations. Hence its execution time is $\Theta(n^2)$.

The estimation of the complexity of `merge_sort` is more involved. First, the number of operations required by the merge procedure is $\Theta(n)$. Next, if we call $T(n)$ (the order of) the number of operations

required by `merge_sort` working on a list of size $n$, we see that roughly:

$$T(n) = 2T(n/2) + n\,.$$

Replacing $n$ with $n/2$ we have $T(n/2) = 2T(n/4) + n/2$, hence

$$T(n) = 2T(n/2) + n = 2(2T(n/4) + n/2) + n = 4T(n/4) + 2n\,.$$

Repeating $k$ times we get:

$$T(n) = 2^k T(n/2^k) + kn\,.$$

So for $k = \log_2 n$ we have

$$T(n) = nT(1) + n\log_2 n = \Theta(n \log n)\,.$$