function complexity is thus, in the worst case,

$$C(n) = C_1 + C_2(n) + C_5 = a + (n-1)(c+a) = (n-1)c + n.a$$

and in the average case

$$C(n) = a + (n-1)(c + \frac{a}{2}) = (n-1)c + \left(\frac{n+1}{2}\right) a$$

1.5 Introduction to recurrence equations

We saw (see section 1.4) how to evaluate the complexity of an iterative algorithm. Rules described so far are however insufficient in a recursive case. One has then to work in two steps:

1. Determine a recurrence equation giving *(i)* the complexity of the algorithm in the simplest case(s) – the initial states of the system – and *(ii)* the process to compute the complexity in a given state when we know it in a state slightly closer to one of the simple cases previously determined. It's the same approach as proof by induction, be it about integers or about structures like trees. We get a system of the form

$$\begin{cases} C_{E_{init}} &= \exp_{init} \\ C_E &= f\left(E, C_{E'}\right) \end{cases}$$

   where E' is a state "closer" to initial state E_{init} than E is.

2. Solve this system of (recurrence) equations.

In the general case, the second step is very complex and beyond the scope of this course. Nonetheless, we give in this section several examples representing simple cases that can pop up when studying classical algorithms, in which the "state" is often represented by only one or two numerical values.

1.5.1 Example of logarithmic complexity

We can frequently expect a logarithmic complexity when each recursive call (supposed unique inside the function) divides the size of the search space by a constant factor. It is for instance the case in the binary search inside a

sorted array (see 1.1). Another example is that of the indian exponancial. This way of computing an exponential relies on the following equations:

$$\begin{cases} \exp(x,0) & = & 1 \\ \exp(x,2n) & = & \exp(x,n)^2 \\ \exp(x,2n+1) & = & x.\exp(x,n)^2 \end{cases}$$

We take the product as fundamental operation and n as the parameter. It is clear that this choice leads to the following recurrence equations:

$$\begin{cases} C_0 & = & 0 \\ C_{2n} & = & C_n + 1 \\ C_{2n+1} & = & C_n + 2 \end{cases}$$

Since we can always represent a natural integer by the sequence of its coefficients in radix 2

$$n = \sum_{i=0}^{k} a_i 2^i$$

where $a_i \in \{0,1\}$ and $k \leq \log_2 n$, we clearly have $n < 2^{k+1}$
$C_n$being an increasing function of n we get:

$$\begin{aligned} C_n \leq C_{2^{k+1}} &= C_{2^k} + 1 \\ &= (C_{2^{k-1}} + 1) + 1 \\ &= C_{2^{k-1}} + 2 \\ &\vdots \\ &= C_0 + k + 1 \\ &= k + 1 \end{aligned}$$

and so $C_n \in O(\log n)$.

1.5.2 Example of linear complexity

We can mainly expect a linear complexity in two cases:

- There are n recursive calls in the function body and each of them divides the search space by a factor n.
- There is only one function call and it reduces the search space by a constant amount.

The first situation occurs for instance when traversing a binary tree in a depth-first manner. For the sake of simplicity, we will suppose the tree as complete. The fundamental operation here is the processing of a particular node in the tree (for instance, the display of the node key). In such a case, each recursive call (there are two of them) divides the search space by two (each subtree contains roughly half the number of nodes in the tree), which gives the following recurrence equations (the parameter is the size of the tree):

$$\begin{cases} C_1 &= 1 \\ C_n &= 2C_{n/2} + 1 \end{cases}$$

Let's solve this equation in the simple case where n is a power of *2* (if it is not the case we consider - as in the previous example - the power of *2* immediately greater than the size of the tree): $n = 2^k$. We get

$$\begin{aligned} C_{2^k} &= 2C_{2^{k-1}} + 1 \\ &= 2\left(2C_{2^{k-2}} + 1\right) + 1 \\ &= 2^2 C_{2^{k-2}} + 2 \\ &\vdots \\ &= 2^{k-1} C_1 + k - 1 \\ &= 2^{k-1} + k - 1 \end{aligned}$$

and so $C(n) \in O(n)$.

1.5.3 Example of quadratic complexity

An important class of compexity intermediate between linear and quadratic is the set of algorithms for which $C_n = O(n \log n)$. We'll see an important instance of this category at the end of this course.

There are few classical recursive algorithms of quadratic complexity. Such an algorithm would contain, for instance, 4 recursive calls in the function code, each of them dividing the search space by 2:

$$\begin{cases} C_1 &= 1 \\ C_n &= 4C_{n/2} \end{cases}$$

Choosing again a power of 2 as the value for n ($n = 2^k$), we can write

$$C_{2^k} = 4C_{2^{k-1}}$$

$$
\begin{aligned}
&= 4^2 C_{2^{k-2}} \\
&\vdots \\
&= 4^k C_1 \\
&= (2^k)^2
\end{aligned}
$$

and so $C_n \in O(n^2)$.

Notice: by extending this example using different values for coefficients, we can obtain polynomial complexity of arbitrary degree.

1.5.4 Example of exponential complexity

The programs that fall into this category are not practically usable, but in the case where the search space is extremely small.

Let's take the example of a naive version of a function than computes the terms of the Fibonacci sequence:

```
int fibo(int n) {
  if (n<2) return n;
  return fibo(n-1) + fibo(n-2);
}
```

We choose n as the parameter and the addition as the unique fundamental operation. The recurrence equation is pretty straightforward:

$$
\begin{cases}
C_0 &= 0 \\
C_1 &= 0 \\
C_n &= 1 + C_{n-1} + C_{n-2}
\end{cases}
$$

We can see that, thanks to a very simple variable change ($F_n = C_n + 1$), the last equation becomes $F_n = F_{n-1} + F_{n-2}$, which is precisely the value of the nth term of the Fibonacci sequence. We'll admit that this sequence grows exponentially, the asymptotic behaviour being described by $F_n \in O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$ (the complete proof is beyond the scope of this course). Let us give an idea of the irrealistic aspect of such a way to program the function: if we suppose that the time needed to execute the fundamental operation (adding two integers) is $100ns\,(10^{-7}s)$, we get the following order of magnitude for the execution time of F_n:

n	time for F_n
20	1,3 ms
30	1,9 s
50	40 min
100	2 millions years

2 Abstract Data Types

A data type is used in a program to represent domain concepts and to manipulate them as easily and efficiently as possible. It is of primary importance to think, as early as during the design phase of the type, about the legal *operations* that are to be defined on the objects of the type. The definition of those operations must be written in such a way that it does not let any room for human interpretation (particularly from the development team).

In order to limit human interpretation, it is a good idea to use a symbolic formalism to define and represent the concepts to manipulate. The following sections describe the Abstract Data Type (noted ADT from now on) formalism, used in this course to represent the classical data structures (lists, trees, graphs ...) and their associated algorithms.

2.1 ADT definition

When studying the basics of logic, we use the notion of truth value, often called boolean. This notion is defined using several possible values (namely true and false) and operations that one can apply to those values (and, or, not...). It is possible to define those operations in a constructive manner using a logical table. Another approach consists in describing the *profile* of the operation (syntax), then to define its *behaviour* (semantics). For instance, the profile (or *signature*) for the `and` operation (which takes two booleans and returns a boolean) is:

$$\texttt{and: boolean} \times \texttt{boolean} \rightarrow \texttt{boolean}$$

Let's try to generalize this approach to type characterization:

An ADT is a tuple *(T,U,O,P,A)* where

- T is the name of the ADT we are defining.

- U is a set of (already defined) ADTs *used* in the definition of T. Notice that this implies a notion of *hierarchy* of ADTs.
- O is a set of *operation* signatures that defines the syntax for legal use of T objects.
- P is a set of *pre-conditions* (logical expressions that always must hold true) to using some of the operations in O.
- A is a set of *axioms* (logical expressions that are admittedly true) defining the behaviour of the operations in O.

The notions of signature, pre-condition and axiom demand to be more precisely defined.

2.1.1 Operation signature

The signature of an operation is made of

1. its name;
2. a (possibly empty) set of ADT describing the domains for parameter values for the operation;
3. an ADT describing the domain of the result of the operation.

The ADT specifying the parameter types are called input ADT (or source ADT), that specifying the result is called output (or target) ADT.

Example The following partial example is a excerpt from the definition of a `boolean` ADT with operation `and`:

```
ADT boolean
Operations
  and: boolean × boolean → boolean
  ...
```

This expresses the fact that this operation needs two parameters of type `boolean`, and returns a `boolean` result. If the operation does not need any parameter, it is called a constant. In this case there is no ADT on the left size of the operation signature arrow.

It is important to understand that the signature by itself is insufficient to characterize the operation, since it only gives the *syntax* of the calls. To make things clear, if we had used the term S instead of *boolean* for the ADT, and e instead of *and* for the operation, the reader would probably not be able to infer (by interpretation of the context) any semantics from this purely syntactic declaration:

```
ADT S
Operations
  e: S × S → S
  ...
```

2.1.2 Operation typology - terminology

We saw that existing ADT can help define a new ADT. For instance, it is clear that we need ADT `boolean` if we want to define ADT `Set` (for instance to define the `includes` predicate). In an ADT definition,

- We call *defined ADT* the ADT being currently defined,
- we call *predefined ADT* an existing ADT involved in the definition of the defined ADT
- we call *internal operation* an operation whose result is of the defined ADT
- we call *observer operation* an operation which has at least one parameter of the defined ADT, and returning a result of one of the predefined ADT
- we call *constructors* the minimum set of internal operations needed to generate any value of the type.

2.1.3 pre-condition

Operations are not always defined for all possible values of their input types. Some of them are only defined on a strict subset of the domain. For instance, if we consider the `Natural` ADT and the `successor` / `predecessor` operations, it is clear that the former is defined on the wole set of natural numbers but the second is undefined for zero...

Such incompletely defined operations are called partial (noted in the ADT definition using $\rightharpoonup$ instead of $\rightarrow$ to separate parameters from the return type in the signature), and for each of them we must specify the conditions of application of the operation. They are expressed in terms of logical invariants (expressions that must always hold true). For instance, the definition of a `PreNatural` ADT could contain:

```
ADT preNatural
Uses boolean
Operations
  0   : → preNatural
  succ: preNatural → preNatural
  prec: preNatural⇀ preNatural
  _=_ : preNatural× preNatural → boolean
  ...
pre-conditions
  prec(i) ⇔ i ≠ 0
```

an "iff" (*if and only if*) can replace the $\Leftrightarrow$ in the pre-condition.

2.1.4 axiom

As we have seen previously, the definition of the ADT is complete if and only if we provide a semantic for all operations, so as to avoid any misleading interpretation. This semantic can be given by completely define the code for the operation (but then we get an *implementation*, leading to what we'll call a *concrete type*), or by specifying a set of axioms that rule the operations' behaviour and the relations between them.

For example we could define the $=$ predicate for ADT PreNatural using the following axioms:

$0 = 0$
$succ(i) = succ(j) \equiv i = j$
$0 \neq succ(i)$ `// or else "not(0=succ(i))"`

The meaning of $\equiv$ is roughly "is equivalent to", "is the same as", "can be rewriten as"...

It is important to realize that the two approaches to defining operations lead to two very different kinds of specifications, commonly refered to as

declarative approach *vs imperative* approach. For instance, if we want to define the sum of two naturals, the latter looks like

$n + m \equiv succ(succ(...(succ(n))...)$, repeating m times operation *succ*.

whereas the declarative approach would consist in writing

$n + 0 \equiv n$

$n + succ(m) \equiv succ(n + m)$

In the rest of the document we systematically use declarative approach for operation definition.

2.1.5 An example: booleans

We give here a simple but consistent definition for boolean ADT. When operations are represented using operators (such as here for *and* and *or* operations) we represent the parameters by position markers. This allows for using notation $a \wedge b$ which is more "natural" than $\wedge(a, b)$

```
ADT boolean
operations
  true : → boolean
  false : → boolean
  !_    : boolean → boolean
  _∧_   : boolean × boolean → boolean
  _∨_   : boolean × boolean → boolean
axioms
  !true ≡ false
  !!a ≡ a
  a ∧ true ≡ a
  a ∧ false ≡ false
  a ∨ b = !(!a ∧ !b)
```

N.B. (important): The set of operations should be as light as possible, while remaining consistent and complete: some properties can be deduced from axioms (those properties are called *theorems*) by mixing axioms (and already defined theorems). We will *admit* that this will be the case when axioms can describe the effect of applying *any* observer on *any* constructor.

3 Linear data structures

Linear data structures are the simplest form of data organisation. Basically, they can be thought of as associating with each element they contain a position in the collection.

Stacks for instance are very widely used in computer science, if only in program execution (operating system code) or translation from recursive to iterative code.

Queues are used to store information in such a way that it can be processed in the same order than it became available.

General lists form the basis for functional or logical programming languages (Lisp, Caml, Prolog,...).

The section is organized as follows: we start with the presentation of the simplest structure, which is the stack. Then we define simple queues (i.e., without priority), and the List ADT. The section concludes on several examples of possible extensions to those specifications.

3.1 The Stack ADT

A stack in computer science can be seen as a homogeneous collection of elements for which only the topmost element is accessible. For instance, a stack of plates is almost always accessed by its top, be it to take plates before lunch or dinner, or to put them back after wash up. One commonly uses the term LIFO (for *Last In, First Out*) to refer to that kind of data structure: the last element that enteres the stack is the first to leave it.

Any formal specification for a stack must at least define the following operations:

- Tell wether the stack is empty.
- Push a new element on top of the stack.
- Pop the top element out of the stack.
- Tell what the top element is.

Of course (as will always be the case) one has to add to those explicit functionalities the classical operation for object creation. We then get the following ADT definition:

```
ADT Stack
Uses Element, boolean
operations
  create   : → Stack
  isEmpty: Stack → boolean
  push: Stack × Element → Stack
  pop: Stack ⇀ Stack
  top  : Stack ⇀ Element
preconditions
  pop(p) iff !isEmpty(p)
  top(p) iff !isEmpty(p)
axioms
  isEmpty(create) ≡ true // A1
  isEmpty(push(p,e)) ≡ false // A2
  top(push(p,e)) ≡ e // A3
  pop(push(p,e)) ≡ p // A4
```

Operations *pop* and *top,* being not defined everywhere, need pre-conditions.

A few notes about axioms:

- A1: we make shure that a newly created stack must be empty
- A2: as soon as we push an element on top of the stack, it is not empty any more
- A3: when we push an element on top of the stack, the (new) top is the element we just pushed
- A4: if we pop the stack just after pushing an element, we go back to the initial stack (before the push / pop couple)

3.2 The Queue ADT

A queue is an homogeneous collection of elements accessed by one of the two ends of the structure: insertion at the one end, removals at the other end. Intuitively, it mirrors the behaviour of a real queue, such as the one that can appear at the entrance of a cinema for instance. It can be called FIFO, for *First In, First Out*: the first element that entered the queue is the first to leave it.

Expected functionalities are as follows: