Chapitre 7

option informatique

Analyse d'un algorithme

1. Introduction

Analyser un programme ou un algorithme, c'est prouver sa *terminaison*, c'est à dire montrer qu'il se terminera en un temps fini, et prouver sa *correction*, c'est à dire s'assurer que le résultat fourni est bien solution au problème que le programme ou l'algorithme est destiné à résoudre.
Nous ne reviendrons que brièvement sur le cas d'une fonction inductive, car nous avons abordé le problème de la terminaison dans le chapitre précédent, et la preuve de correction est en général elle aussi établie par induction. À titre d'exemple, analysons la fonction suivante :

```
let rec f = function
  | 1 -> 1
  | n -> 1 + f (n / 2) ;
```

La fonction $\varphi : \mathbb{N}\setminus\{0,1\} \to \mathbb{N}$ définie par $\varphi(n) = \left\lfloor \frac{n}{2} \right\rfloor$ vérifie : $\varphi(n) < n$, ce qui assure la terminaison de la fonction définie ci-dessus. En outre, on peut affirmer qu'elle calcule l'*unique* fonction $f : \mathbb{N}^* \to \mathbb{N}$ vérifiant $f(1) = 1$ et $f(n) = 1 + f\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$. Il reste à trouver de quelle fonction il s'agit : il n'est guère difficile de postuler puis de prouver qu'elle calcule le nombre de bits nécessaire à l'écriture de l'entier n en base 2, à savoir : $f(n) = 1 + \lfloor \log_2 n \rfloor$.

Mais qu'on prenne garde cependant : il est facile de définir des fonctions récursives sortant du cadre étroit que nous avons donné au chapitre précédent, et pour de telles fonctions les preuve de terminaison et de correction sont loin d'être évidentes. On citera à titre d'exemple la fonction de Takeuchi, souvent utilisée pour tester la qualité de l'implémentation de la récursivité dans les langages fonctionnels :

```
let rec tak (x, y, z) =
  if y < x then tak (tak (x-1, y, z), tak (y-1, z, x), tak (z-1, x, y))
           else y ;;
```

Prouver sa terminaison et donner une expression simple de ce qu'elle calcule est un exercice difficile.

2. Analyse d'un programme impératif

Étant donné un bloc d'instructions B, on appelle *contexte* de B l'ensemble des données manipulées (en lecture et en écriture) par les instructions de B. Si on note d l'état du contexte à l'entrée de B et d' son état à la sortie, analyser le bloc d'instructions consiste à déterminer l'évolution du contexte, c'est à dire à déterminer une fonction f vérifiant $d' = f(d)$.

2.1 Cas d'une boucle inconditionnelle

Considérons la boucle suivante : `for k = 1 to n do (* bloc B *) done`. Si on note d_0 l'état du contexte à l'entrée de la boucle, on constate que l'état à la sortie sera égal au n^{e} terme d_n de la suite définie par la relation $d_{k+1} = f(d_k)$. Analyser la boucle revient donc souvent à déterminer une *forme close* pour d_n (c'est à dire une formule explicite), ou à défaut une propriété invariante par f, qu'on appelle un *invariant de boucle*.

À titre d'exemple, analysons la fonction suivante :

```
let g n =
  let x = ref 1 in
    for k = 1 to n do x := k * !x done ;
  !x ;;
```

Le contexte est ici constitué de l'indice de boucle k et de la référence x, cette fonction calcule donc la n^{e} itération de la suite $d_k = (k, x_k)$ définie par la condition initiale $d_0 = (0,1)$ et la relation $d_{k+1} = f(d_k)$, avec $f(u,v) = \big(u+1, (u+1)v\big)$, puis retourne la valeur de x_n.
On donnera alors l'invariant de boucle $x_k = k!$ pour justifier que cette fonction calcule $n!$.

Considérons maintenant la fonction suivante :

```
let h n =
  let x = ref 0 and y = ref 1 in
    for k = 1 to n do
      let z = !y in y := !x + !y ; x := z
    done ;
  !x ;;
```

Le contexte est ici constitué des deux références x et y (l'indice k de la boucle n'intervient pas dans le bloc d'instructions B délimité par **do** et **done**, et **z** est une variable locale à B). Cette fonction calcule donc la n^{e} itération de la suite $d_k = (x_k, y_k)$ définie par la condition initiale $d_0 = (0,1)$ et la relation $d_{k+1} = f(d_k)$, avec $f(u,v) = (v, u+v)$, puis retourne la valeur de x_n.
En notant $(f_n)_{n\in\mathbb{N}}$ la suite de Fibonacci, on donne l'invariant de boucle $(x_k, y_k) = (f_k, f_{k+1})$ pour justifier que cette fonction retourne la valeur de f_n.

2.2 Cas d'une boucle conditionnelle

Considérons maintenant une boucle conditionnelle : **while** `(* condition *)` **do** `(* bloc B *)` **done**.
La condition est constituée d'une fonction à valeurs booléennes c définie sur le contexte d. Si on note d_k l'état du contexte après k passages par le bloc B, l'état à la sortie de la boucle sera la valeur d_n définie par : $n = \min\Big\{k \in \mathbb{N} \;\Big|\; c(d_k) = \mathit{false}\Big\}$.
Prouver la terminaison consiste à prouver qu'un tel entier existe, analyser la boucle revient à déterminer une forme close pour d_n.

À titre d'exemple, analysons la fonction suivante :

```
let m a n =
  let x = ref 1 and y = ref a and z = ref n in
  while !z > 0 do
    if !z mod 2 = 1 then  x := !x * !y ;
    z := !z / 2 ;
    y := !y * !y done ;
  !x ;;
```

Le contexte est constitué des trois références x, y et z, on note donc $d_k = (x_k, y_k, z_k)$. Nous avons $x_0 = 1$, $y_0 = a$ et $z_0 = n$ et les relations :

$$x_{k+1} = \begin{cases} x_k y_k & \text{si } z_k \text{ est impair} \\ x_k & \text{sinon} \end{cases}, \qquad y_{k+1} = y_k^2, \qquad z_{k+1} = \left\lfloor \frac{z_k}{2} \right\rfloor.$$

On obtient facilement l'expression : $y_k = a^{(2^k)}$.
Pour déterminer z_k, nous allons introduire la décomposition en base 2 de l'entier n : $n = (b_p b_{p-1} \cdots b_1 b_0)_2$. Il apparaît alors que pour tout $k \in [\![0, p]\!]$, $z_k = (b_p b_{p-1} \cdots b_k)_2$.
À cette étape de l'analyse, on peut affirmer que $z_p = b_p \neq 0$ et $z_{p+1} = 0$, ce qui prouve déjà que cet algorithme se termine et retourne la valeur de x_{p+1}.
Remarquons enfin que la formule : $x_{k+1} = \begin{cases} x_k y_k & \text{si } b_k = 1 \\ x_k & \text{si } b_k = 0 \end{cases}$ peut se réduire à : $x_{k+1} = x_k y_k^{b_k}$ et donc :

$$x_{p+1} = \prod_{k=0}^{p} y_k^{b_k} = \prod_{k=0}^{p} a^{(b_k 2^k)} = a^{\sum_{k=0}^{p} b_k 2^k} = a^n.$$

Il s'agit donc d'un algorithme de calcul de a^n.

2.3 Un itérateur générique

Nous venons de constater qu'une boucle se ramène à l'itération d'une suite $(d_n)_{n\in\mathbb{N}}$ définie par la donnée de d_0 et de la relation $d_{k+1} = f(d_k)$. Nous pouvons donc définir un itérateur générique qui pourra être utilisé pour remplacer l'usage des boucles.
Dans le cas des boucles inconditionnelles, on définit :

```
# let rec itere f d = function
    | 0 -> d
    | n -> itere f (f d) (n-1) ;;
itere : ('a -> 'a) -> 'a -> int -> 'a = <fun>
```

Ainsi, les fonctions g et h peuvent être définies de la façon suivante :

```
let g n = snd (itere (function (k, x) -> (k+1, (k+1)*x)) (0, 1) n) ;;

let h n = fst (itere (function (x, y) -> (y, x+y)) (1, 1) n) ;;
```

Dans le cas des boucles conditionnelles, on définit :

```
# let rec tant_que c f d = match (c d) with
    | true  -> tant_que c f (f d)
    | false -> d ;;
tant_que : ('a -> bool) -> ('a -> 'a) -> 'a -> 'a = <fun>
```

et la fonction m peut alors être définie par :

```
let m a n =
  let (r, _, _) = tant_que
    (function (x, y, z) -> z > 0)
    (function (x, y, z) -> ((if z mod 2 = 1 then x*y else x), y*y, z/2))
    (1, a, n)
  in r ;;
```

Il nous reste à noter que les fonctions **`itere`** et **`tant_que`** sont récursives terminales pour pouvoir énoncer le résultat suivant :

Théorème. — *Tout algorithme utilisant une boucle conditionnelle ou inconditionnelle possède une version récursive terminale.*

Bien sur, les versions que nous avons obtenues peuvent être rendues plus lisibles en utilisant des fonctions auxiliaires s'inspirant de la définition des fonctions génériques. Par exemple :

```
let h =
  let rec aux (u, v) = function
    | 0 -> u
    | n -> aux (v, u + v) (n-1)
  in aux (1, 1) ;;

let m a =
  let rec aux (x, y) = function
    | 0                   -> x
    | z when z mod 2 = 1 -> aux (x * y, y * y) (z / 2)
    | z                   -> aux (x, y * y) (z / 2)
  in aux (1, a) ;;
```

les variables (u, v) ou (x, y) utilisées dans les deux fonctions précédentes sont souvent appelées des *accumulateurs* ; leur usage est fréquent dans les versions récursives terminales des itérations.

Attention. Nos deux fonctions génériques ont pour objet de calculer $d_n = f \circ f \circ \cdots \circ f(d_0)$. Elles sont rendues récursives terminales en interprétant cette égalité par : $d_n = f^{n-1} \circ f(d_0)$. Ça n'aurait pas été le cas si nous l'avions interprété par : $d_n = f \circ f^{n-1}(d_0)$, ce qui aurait conduit à la version *non terminale* suivante :

```
let rec itere f d = function
  | 0 -> d
  | n -> f (itere f d (n-1)) ;;
```

qui, contrairement à la précédente, a un coût spatial.

2.4 Dérécursivation

Nous venons de constater que tout algorithme itératif possède une version récursive (terminale). Nous allons maintenant nous poser la question réciproque : un algorithme récursif possède-t-il nécessairement une version itérative ? C'est le problème de la *dérécursivation*.

• Le cas de la récursivité terminale

Il est facile de répondre par l'affirmative à cette question dans le cas de la récursivité terminale. Considérons en effet le modèle de fonction suivant :

```
let rec f = function
  | x when dans_A x -> g x
  | x               -> f (phi x) ;;
```

Cette fonction possède la version itérative suivante :

```
let f x =
  let y = ref x in
    while not dans_A !y do y := phi !y done ;
  g !y ;;
```

On peut à ce sujet noter que certains compilateurs [1] détectent la récursivité terminale et optimisent son exécution en transformant la récursion en itération, de manière à économiser l'espace de la pile d'exécution.

• La cas général

Il est possible, notamment en simulant la récursivité à l'aide d'une pile, de prouver que tout algorithme récursif possède une version itérative, ceci permettant de conclure qu'itération et récursion ont le même pouvoir expressif. Nous nous contenterons cependant de ne traiter ici que quelques cas particuliers simples, pour lesquels il s'agira en général d'extraire un invariant de la définition récursive.
On pourra observer que cette démarche nous permettra en outre d'obtenir des versions récursives terminales de fonctions initialement non terminales, en utilisant le plus souvent un accumulateur.

Commençons par un exemple très simple : le calcul de la somme $s_n = \sum_{k=1}^{n} k$ (sans recourir à la formule de Gauss).

La première solution récursive consiste à écrire :

```
let rec sum1 = function
  | 0 -> 0
  | n -> n + sum1 (n-1) ;;
```

Cette solution n'est pas terminale.

En observant qu'il s'agit de l'itération de la suite (n, s_n), on obtient la version itérative suivante :

```
let sum2 n =
  let rec s = ref 0 in
  for k = 1 to n do s := !s + k done ;
  !s ;;
```

On peut alors convertir cette fonction en une version récursive terminale :

1. C'est le cas des compilateurs Caml, mais pas des interprètes Python.

```
let sum3 =
  let rec aux acc = function
    | 0 -> acc
    | n -> aux (acc + n) (n-1)
  in aux 0 ;;
```

Attention. Ne pas croire que la différence entre la récursivité terminale et non terminale soit anecdotique. En effet, les deux fonctions **sum1** et **sum3** n'ont pas les mêmes performances :

```
# sum1 200000 ;;
Exception non rattrapée: Out_of_memory
# sum3 200000 ;;
- : int = 20000100000
```

Autre exemple presque identique : celui de la fonction factorielle.

```
let rec fact = function
  | 0 -> 1
  | n -> n * fact (n-1) ;;
```

En considérant l'itération de la suite $(n, n!)$ on obtient la version récursive terminale suivante :

```
let fact =
  let rec itere acc = function
    | 0 -> acc
    | n -> itere (n * acc) (n - 1)
  in itere 1 ;;
```

3. Complexité spatiale et temporelle

Prouver la terminaison et la validité d'un algorithme n'est pas toujours suffisant, il faut encore que ce dernier soit utilisable dans la pratique, c'est à dire qu'il utilise une quantité « raisonnable » de mémoire, et que son temps d'exécution soit lui aussi « raisonnable ». Mais encore faut-il que l'on s'entende sur ce qualificatif ! Après tout, deux ordinateurs différents peuvent avoir des performances différentes (aussi bien en mémoire qu'en vitesse), et sur un ordinateur donné, un algorithme traduit en programme aura des performances dépendant du langage de programmation utilisé, voire même du compilateur utilisé.

Mesurer la *complexité spatiale* (l'encombrement en mémoire) ou la *complexité temporelle* (le temps d'exécution) va donc nécessiter de choisir une unité de mesure indépendante de tout aspect technique conjoncturel et qui soit pertinente au regard du problème posé. Par exemple, pour analyser la complexité d'un algorithme effectuant essentiellement des calculs numériques, on dénombrera les opérations coûteuses que sont les multiplications, les racines carrées, les logarithmes,... A contrario, l'évaluation de la complexité d'un algorithme de tri se fera par le biais du dénombrement des accès aux emplacements de mémoire et des comparaisons entre données.

3.1 Analyse asymptotique

En général, les algorithmes que nous allons étudier dépendent d'une ou plusieurs grandeurs (le plus souvent entières), et la complexité de ces algorithmes est fonction de ces grandeurs. Mais le calcul exact de la complexité n'est pas toujours possible ; en outre, analyser le coût d'un algorithme n'a d'intérêt que pour des grandeurs relativement élevées. On se contentera donc le plus souvent de rechercher un *équivalent* de la complexité lorsque les grandeurs dont dépendent l'algorithme tendent vers $+\infty$.

Exemple. Considérons de nouveau le calcul du n^e terme de la suite de Fibonacci $(f_n)_{n\in\mathbb{N}}$, et comparons le coût des trois algorithmes suivants :

```
let rec fib1 = function
  | 0 -> 0
  | 1 -> 1
  | n -> fib1 (n-1) + fib1 (n-1) ;;
```

Il parait raisonnable de choisir pour unité de mesure temporelle le nombre d'additions c_n effectuées pour calculer f_n. Cette suite vérifie les relations : $c_0 = c_1 = 0$ et $c_n = c_{n-1} + c_{n-2} + 1$ qui conduisent à $c_n = f_{n+1} - 1$. Sachant que $f_n \sim \frac{\varphi^n}{\sqrt{5}}$ avec $\varphi = \frac{1+\sqrt{5}}{2}$, nous avons $c_n \sim \frac{\varphi^{n+1}}{\sqrt{5}}$. Un tel coût est qualifié d'*exponentiel*.

Compte tenu de ce que l'on sait de la récursivité, il parait raisonnable de choisir pour unité de mesure de la complexité spatiale le nombre d_n d'appels à la fonction **`fib1`** effectués pour calculer f_n. Cette suite vérifie les relations $d_0 = d_1 = 1$ et $d_n = d_{n-1} + d_{n-2} + 1$ qui conduisent à $d_n = 2f_{n+1} - 1 \sim \frac{2\varphi^{n+1}}{\sqrt{5}}$; le coût spatial est lui aussi exponentiel.

```
let fib2 n =
  let t = make_vect (n+1) 0 in
  t.(1) <- 1 ;
  for k = 2 to n do t.(k) <- t.(k-1) + t.(k-2) done ;
  t.(n) ;;
```

Ce second algorithme est facile à analyser : le nombre d'additions effectuées est égal à $c_n = n - 1 \sim n$, et il est raisonnable de penser que la création du vecteur **`t`** va aussi avoir un coût temporel proportionnel à n. On dira que la complexité temporelle est *linéaire*.

Quant à la complexité spatiale, elle dépend clairement de l'emplacement en mémoire alloué au vecteur **`t`**, elle est donc aussi linéaire.

```
let fib3 =
  let rec aux (u, v) = function
    | 0 -> u
    | n -> aux (v, u+v) (n-1)
  in aux (0, 0) ;;
```

Enfin, ce troisième algorithme effectue à l'évidence $c_n = n$ additions, et n'a pas de coût spatial (ou plus exactement un coût spatial constant) car la récursivité est terminale. Il a donc une complexité temporelle linéaire et une complexité spatiale constante.

3.2 Notations de Landau

Ce qui importe en premier lieu lorsqu'on mesure la complexité d'un algorithme, c'est de connaître son ordre de grandeur. Voila pourquoi on utilise souvent les notations de Landau pour présenter les résultats des analyses de complexité. Outre le O que vous connaissez, on en utilise deux autres, Ω et Θ, pour traduire plus précisément cette idée :

- $u_n = O(v_n)$ (u_n est *dominée* par v_n) lorsqu'il existe $M > 0$ tel que : $\forall n \in \mathbb{N}$, $u_n \leqslant M v_n$;
- $u_n = \Omega(v_n)$ (u_n *domine* v_n) lorsque $v_n = O(u_n)$;
- $u_n = \Theta(v_n)$ (u_n et v_n ont *même ordre de grandeur*) lorsque $u_n = O(v_n)$ et $u_n = \Omega(v_n)$.

On peut alors classer les algorithmes suivant leur complexité c_n, qui sera qualifiée de :

- *logarithmique* [2] lorsque $c_n = \Theta(\log n)$;
- *linéaire* lorsque $c_n = \Theta(n)$;
- *quasi-linéaire* lorsque $c_n = \Theta(n \log n)$;
- *quadratique* lorsque $c_n = \Theta(n^2)$;
- *polynomiale* lorsque $c_n = \Theta(n^k)$ avec $k > 1$;
- *exponentielle* lorsqu'il existe $a > 1$ tel que $c_n = \Omega(a^n)$.

Pour avoir une idée de ce que peuvent représenter ces différents ordres de grandeurs lorsqu'on mesure une complexité temporelle, nous allons imaginer un ordinateur exécutant un programme dont la mesure de complexité temporelle est le nombre d'opérations numériques effectuées. En 2013, le nombre de FLOPS (FLoating point Operations Per Second) d'un ordinateur personnel est de l'ordre de 10^{11}. Nous allons calculer pour différentes valeurs de n le temps requis en fonction de la complexité du programme :

2. comme dans tout cours d'informatique, le logarithme utilisé ici est celui en base 2. Cela dit, pour cette définition en particulier ceci n'a pas d'importance.

	$\log n$	n	$n\log n$	n^2	n^3	2^n
10^2	0,7 ns	1 ns	5 ns	0,1 µs	10 µs	$4\cdot10^{11}$ années
10^3	1 ns	10 ns	0,7 µs	10 µs	10 ms	10^{290} années
10^4	1,3 ns	0,1 µs	0,9 µs	1 ms	10 s	
10^5	1,7 ns	1 µs	12 µs	0,1 s	2,8 h	
10^6	2 ns	10 µs	0,1 ms	10 s	116 jours	

Observons maintenant quelle valeur de n on peut atteindre en une minute :

$\log n$	n	$n\log n$	n^2	n^3	2^n
$10^{1800000000000}$	$6\cdot10^{12}$	$1,6\cdot10^{11}$	$2,4\cdot10^6$	18171	42

Ces résultats sont parlants : au delà d'une complexité quadratique, le coût temporel devient vite exorbitant, et un algorithme de complexité exponentielle n'est utilisable que pour de très faibles valeurs de n.
Notez qu'on peut aisément comprendre au vu de ce tableau pourquoi on parle de complexité quasi-linéaire lorsque $c_n = \Theta(n\log n)$; la différence avec un algorithme de coût linéaire reste minime.

3.3 Différents types de complexité

Il peut arriver que la complexité d'un algorithme dépende non seulement de la taille des données n, mais aussi des valeurs particulières de ses données. Si on note $\mathscr{D}_n$ l'ensemble des données de taille n et $c(d)$ la complexité pour la donnée d, on est amené à distinguer plusieurs types de complexité :

- la complexité *dans le meilleur des cas* $\mathrm{C}_{\min} = \min_{d\in\mathscr{D}_n} c(d)$;
- la complexité *dans le pire des cas* $\mathrm{C}_{\max} = \max_{d\in\mathscr{D}_n} c(d)$;
- la complexité *en moyenne* $\mathrm{C}_{\text{moy}} = \sum_{d\in\mathscr{D}_n} p(d)c(d)$;

où $p(d)$ désigne la loi de probabilité associée à l'apparition des données de taille n.
En général, déterminer le coût dans le meilleur des cas ne présente que peu d'intérêt : il vaut toujours mieux envisager le pire pour éviter de mauvaises surprises. La plus-part du temps, on se contentera donc de déterminer la complexité dans le pire des cas et dans de rares cas, la complexité en moyenne.

À titre d'exemple, considérons la fonction suivante, dont le rôle est le calcul du produit des termes d'une liste d'entiers de longueur n. On utilisera comme mesure du coût le nombre de multiplications effectuées.

```
let produit =
  let rec f acc = function
    | []   -> acc
    | 0::q -> 0
    | t::q -> f (t * acc) q
  in f 1 ;;
```

Le coût dans le meilleur des cas intervient lorsque la liste débute par un 0 : aucune multiplication n'est effectuée. En revanche, lorsque la liste ne comporte pas de 0, le nombre de multiplication est maximal et égal à n ; c'est le coût dans le pire des cas.

Pour déterminer le coût en moyenne, il nous faut un renseignement supplémentaire : la fréquence d'apparition de 0 dans les listes en question. Considérons par exemple que les entiers qui interviennent ne prennent que les valeurs de 0 à 9 de façon équiprobable et indépendante. On peut les classer suivant le rang d'apparition de 0 dans la liste :

- 10^{n-1} listes débutent par 0 et n'effectuent donc pas de multiplications ;
- $9\cdot10^{n-2}$ listes effectuent une multiplication avant de rencontrer un 0 ;
- …
- $9^k\cdot10^{n-1-k}$ listes effectuent k multiplications avant de rencontrer un 0 ;
- …
- 9^{n-1} listes effectuent $n-1$ multiplications avant de rencontrer un 0 ;
- et enfin 9^n listes ne comportent pas de 0 et effectuent donc n multiplications.

Le coût en moyenne est égal à : $\frac{1}{10^n}\left(\sum_{k=0}^{n-1} k\times 9^k\cdot 10^{n-1-k} + n\times 9^n\right) = 9\left(1-\left(\frac{9}{10}\right)^n\right)$. On constate que ce dernier est borné ; cet algorithme possède une complexité en moyenne constante.

4. Exercices

4.1 Analyse d'un algorithme

Exercice 1 Déterminer l'itérateur associé à la suite $(u_n)_{n\in\mathbb{N}}$ définie par la donnée de $u_0 = u_1 = 1$ et la relation de récurrence $u_{n+2} = u_{n+1} + (-1)^n u_n$, et en déduire les versions impérative et récursive terminale de la fonction permettant le calcul de u_n.

Exercice 2 Démontrer que l'algorithme de multiplication de deux entiers est correct :

```
let multiplie a b =
  let x = ref 0 and y = ref a and z = ref b in
  while !z > 0 do x := !x + !y * (!z mod 2) ;
                  y := 2 * !y ;
                  z := !z / 2
               done ;
  !x ;;
```

En donner ensuite une version récursive terminale.

Exercice 3 Prouver la terminaison et déterminer ce que calcule la fonction suivante :

```
let f a =
  let y = ref 0 and z = ref 1 and u = ref 0 in
  while !y <= a do
    y := !y + !z ;
    z := !z + 2 ;
    u := !u + 1 done ;
  !u - 1 ;;
```

Exercice 4 Déterminer ce que calcule la fonction suivante, et prouvez-le :

```
let f u =
  let x = ref 0 and y = ref u in
  let c = ref 1 and d = ref 1 in
  while (!y > 0) or (!c > 0) do
    let a = !y mod 2 in
    if (a + !c) mod 2 = 1 then x := !x + !d ;
    c := a * !c ;
    d := 2 * !d ;
    y := !y / 2 done ;
  !x ;;
```

Exercice 5 Rédiger une fonction récursive terminale qui :

a) détermine le nombre d'occurrences d'un élément dans une liste ;

b) calcule la moyenne des éléments d'une liste (de type *float list*) ;

c) calcule la liste des entiers compris entre p et q.

4.2 Complexité d'un algorithme

Exercice 6 Vous êtes face à un mur qui s'étend à l'infini dans les deux directions. Il y a une porte dans ce mur, mais vous ne connaissez ni la distance, ni la direction dans laquelle elle se trouve. Par ailleurs, l'obscurité vous empêche de voir la porte à moins d'être juste devant elle.
Décrire un algorithme vous permettant de trouver cette porte en un temps linéaire vis-à-vis de la distance qui vous sépare de celle-ci.

Exercice 7 Dans un groupe de n individus , une *star* est quelqu'un que tout le monde connait mais qui ne connait personne. Pour trouver une star, s'il en existe une, vous ne pouvez poser aux individus de ce groupe que des questions du type : « connaissez-vous x ? ».
Combien de stars au maximum peut-il exister dans un groupe ?
Donner un algorithme trouvant une star s'il en existe une (ou déterminant qu'il n'en existe pas) et de coût linéaire (en prenant comme mesure de la complexité le nombre de questions posées).

Exercice 8 Le problème est de déterminer à partir de quel étage d'un immeuble sauter par une fenêtre est fatal. Vous êtes dans un immeuble à n étages (numérotés de 1 à n) et vous disposez de k étudiants. Il n'y a qu'une opération possible pour tester si la hauteur d'un étage est fatale : faire sauter un étudiant par la fenêtre. S'il survit, vous pouvez le réutiliser ensuite, sinon vous ne pouvez plus.
Vous devez proposer un algorithme pour trouver la hauteur à partir de laquelle un saut est fatal en faisant le minimum de sauts.

a) Si $k \geqslant \lceil \log n \rceil$, proposer un algorithme en $O(\log n)$ sauts.

b) Si $k < \lceil \log n \rceil$, proposer un algorithme en $O\left(k + \frac{n}{2^{k-1}}\right)$ sauts.

c) Si $k = 2$, proposer un algorithme en $O(\sqrt{n})$ sauts.

Exercice 9 Expliquer comment trouver le deuxième plus grand élément d'un vecteur $(a_0, \ldots, a_{n-1})$ en effectuant au plus $n + \lceil \log n \rceil - 2$ comparaisons.
Vous pourrez procéder par analogie avec un tournoi à élimination directe en remarquant que le deuxième joueur le plus fort fait nécessairement partie des adversaires malheureux du vainqueur.

Exercice 10 Dans cet exercice, on s'intéresse au problème des tours de Hanoï lorsqu'on dispose de quatre tiges et de n disques. On note t_n le nombre minimal de mouvements nécessaire pour résoudre le problème.

a) Montrer que si $k \in [\![0, n]\!]$, alors $t_n \leqslant 2t_{n-k} + 2^k - 1$, et en déduire : $\forall p \in \mathbb{N}$, $t_{p(p+1)/2} \leqslant 2t_{p(p-1)/2} + 2^p - 1$.

b) Montrer que pour tout entier $p \in \mathbb{N}$, $t_{p(p+1)/2} \leqslant 2^p(p-1) + 1$.

c) Justifier la croissante de la suite $(t_n)_{n\in\mathbb{N}}$, et en déduire l'existence d'un algorithme résolvant le problème à quatre tiges avec un coût en $O\left(\sqrt{n}2^{\sqrt{2n}}\right)$.