Analyse d'un algorithme

Jean-Pierre Becirspahic
Lycée Louis-Le-Grand

Analyse d'un algorithme

- Prouver sa terminaison.
- Prouver sa correction.
- Évaluer son coût spatial et temporel.

Analyse d’un algorithme

- Prouver sa terminaison.
- Prouver sa correction.
- Évaluer son coût spatial et temporel.

Cas d’une fonction inductive. Terminaison et correction se prouvent par induction. Par exemple :

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

Analyse d'un algorithme

- Prouver sa terminaison.
- Prouver sa correction.
- Évaluer son coût spatial et temporel.

Cas d'une fonction inductive. Terminaison et correction se prouvent par induction. Par exemple :

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

- Terminaison. On pose $E = \mathbb{N}^*$, $A = \{1\}$, $\varphi : E \setminus A \to E$ est défini par : $\varphi(n) = \left\lfloor \frac{n}{2} \right\rfloor$ et vérifie $\varphi(n) < n$.

Analyse d'un algorithme

- Prouver sa terminaison.
- Prouver sa correction.
- Évaluer son coût spatial et temporel.

Cas d'une fonction inductive. Terminaison et correction se prouvent par induction. Par exemple :

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

- Terminaison. On pose $E = \mathbb{N}^*$, $A = \{1\}$, $\varphi : E \setminus A \to E$ est défini par : $\varphi(n) = \left\lfloor \frac{n}{2} \right\rfloor$ et vérifie $\varphi(n) < n$.
- Correction. On prouve par induction que $f(n) = |n|_2$ (nb de bits en base 2).
  - Si $n = 1$, $f(1) = 1 = |1|_2$.
  - Si $n \geqslant 2$, on pose $k = \left\lfloor \frac{n}{2} \right\rfloor$ et on suppose $f(k) = |k|_2$. Alors $n = 2k$ ou $n = 2k+1$ et dans les deux cas $|n|_2 = |k|_2 + 1 = f(k) + 1 = f(n)$.

Analyse d'un algorithme

- Prouver sa terminaison.
- Prouver sa correction.
- Évaluer son coût spatial et temporel.

Cas d'un programme impératif. Si B est un bloc d'instructions, son *contexte* est l'ensemble des données manipulées (en lecture et en écriture) à l'intérieur de B.
Si d désigne l'état du contexte à l'entrée de B et d' son état à la sortie, analyser B consiste à déterminer une fonction f vérifiant $d' = f(d)$.

Analyse d'une boucle inconditionnelle

```
for k = 1 to n do (* bloc B *) done
```

Si d_0 est l'état du contexte à l'entrée de la boucle, l'état à la sortie sera égal à d_n, obtenu par l'itération de la relation $d_{k+1} = f(d_k)$.

Analyse d'une boucle inconditionnelle

`for k = 1 to n do (* bloc B *) done`

Si d_0 est l'état du contexte à l'entrée de la boucle, l'état à la sortie sera égal à d_n, obtenu par l'itération de la relation $d_{k+1} = f(d_k)$.

Exemple.

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

Contexte : l'indice de boucle k et la référence x.

Analyse d'une boucle inconditionnelle

for k = 1 to n do (* bloc B *) done

Si d_0 est l'état du contexte à l'entrée de la boucle, l'état à la sortie sera égal à d_n, obtenu par l'itération de la relation $d_{k+1} = f(d_k)$.

Exemple.

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

Contexte : l'indice de boucle k et la référence x.

La suite $d_k = (k, x_k)$ est définie par $d_0 = (0, 1)$ et la relation $d_{k+1} = f(d_k)$, avec $f(u, v) = \big(u + 1, (u + 1)v\big)$.

Forme close : $d_k = (k, k!)$; cette fonction calcule $n!$.

Analyse d'une boucle inconditionnelle

```
for k = 1 to n do (* bloc B *) done
```

Si d_0 est l'état du contexte à l'entrée de la boucle, l'état à la sortie sera égal à d_n, obtenu par l'itération de la relation $d_{k+1} = f(d_k)$.

Exemple.

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

Contexte : les références x et y.

Analyse d'une boucle inconditionnelle

```
for k = 1 to n do (* bloc B *) done
```

Si d_0 est l'état du contexte à l'entrée de la boucle, l'état à la sortie sera égal à d_n, obtenu par l'itération de la relation $d_{k+1} = f(d_k)$.

Exemple.

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

Contexte : les références x et y.

La suite $d_k = (x_k, y_k)$ est définie par $d_0 = (1, 1)$ et la relation $d_{k+1} = f(d_k)$, avec $f(u, v) = (v, u + v)$.

Forme close : $d_k = (f_k, f_{k+1})$, où $(f_n)_{n\in\mathbb{N}}$ est la suite de FIBONACCI ; cette fonction calcule f_n.

Analyse d'une boucle conditionnelle

```
while (* condition *) do (* bloc B *) done.
```

La condition est une fonction à valeurs booléennes c définie sur le contexte d. Si d_k est l'état du contexte après k passages par le bloc B, l'état à la sortie de la boucle sera d_n défini par : $n = \min\Big\{k \in \mathbb{N} \;\Big|\; c(d_k) = \mathit{false}\Big\}$.

Analyse d'une boucle conditionnelle

while (* condition *) **do** (* bloc B *) **done**.

La condition est une fonction à valeurs booléennes c définie sur le contexte d. Si d_k est l'état du contexte après k passages par le bloc B, l'état à la sortie de la boucle sera d_n défini par : $n = \min\left\{k \in \mathbb{N} \mid c(d_k) = \mathit{false}\right\}$.

Exemple.

```
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 ;;
```

Contexte : les trois références x, y et z. On note $d_k = (x_k, y_k, z_k)$. Alors :

$$x_0 = 1,\ y_0 = a,\ z_0 = n, \quad x_{k+1} = x_k y_k^{z_k \bmod 2}, \quad y_{k+1} = y_k^2, \quad z_{k+1} = \left\lfloor \frac{z_k}{2} \right\rfloor.$$

Analyse d'une boucle conditionnelle

while (* condition *) **do** (* bloc B *) **done**.

La condition est une fonction à valeurs booléennes c définie sur le contexte d. Si d_k est l'état du contexte après k passages par le bloc B, l'état à la sortie de la boucle sera d_n défini par : $n = \min\left\{k \in \mathbb{N} \,\middle|\, c(d_k) = \mathit{false}\right\}$.

Exemple.

```
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 ;;
```

Contexte : les trois références x, y et z. On note $d_k = (x_k, y_k, z_k)$. Alors :

$$x_0 = 1,\ y_0 = a,\ z_0 = n, \quad x_{k+1} = x_k y_k^{z_k \bmod 2}, \quad y_{k+1} = y_k^2, \quad z_{k+1} = \left\lfloor \frac{z_k}{2} \right\rfloor.$$

$y_k = a^{(2^k)}$, et si $n = (b_p b_{p-1} \cdots b_1 b_0)_2$, alors $z_k = (b_p b_{p-1} \cdots b_k)_2$.

Cet algorithme se termine et retourne la valeur de x_{p+1}.

Analyse d'une boucle conditionnelle

while (* condition *) **do** (* bloc B *) **done**.

La condition est une fonction à valeurs booléennes c définie sur le contexte d. Si d_k est l'état du contexte après k passages par le bloc B, l'état à la sortie de la boucle sera d_n défini par : $n = \min\left\{k \in \mathbb{N} \,\middle|\, c(d_k) = false\right\}$.

Exemple.

```
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 ;;
```

Contexte : les trois références x, y et z. On note $d_k = (x_k, y_k, z_k)$. Alors :

$$x_0 = 1,\ y_0 = a,\ z_0 = n, \quad x_{k+1} = x_k y_k^{z_k \bmod 2}, \quad y_{k+1} = y_k^2, \quad z_{k+1} = \left\lfloor \frac{z_k}{2} \right\rfloor.$$

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

Un itérateur générique

Sachant qu'une boucle se ramène à l'itération du contexte, on peut utiliser un itérateur générique :

- Boucles inconditionnelles

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

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

Par exemple, les fonctions g et h peuvent être définies par :

```
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) ;;
```

Le paramètre **d** de la version récursive terminale est souvent appelé un accumulateur.

Un itérateur générique

Sachant qu'une boucle se ramène à l'itération du contexte, on peut utiliser un itérateur générique :

- Boucles conditionnelles

```
let rec tant_que c f d = match (c d) with
  | true  -> tant_que c f (f d)
  | false -> d ;;
```

Par exemple, la fonction *m* peut 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 ;;
```

Un itérateur générique

Sachant qu'une boucle se ramène à l'itération du contexte, on peut utiliser un itérateur générique :

- Boucles conditionnelles

```
let rec tant_que c f d = match (c d) with
  | true  -> tant_que c f (f d)
  | false -> d ;;
```

Par exemple, la fonction *m* peut 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 ;;
```

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

Dérécursivation

Toute fonction récursive terminale possède une version itérative.

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

est équivalente à :

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

Application : certains compilateurs détectent la récursivité terminale et optimisent son exécution en transformant la récursion en itération.

Dérécursivation

Toute fonction récursive possède une version itérative.

Exemple.

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

Version itérative :

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

Dérécursivation

Toute fonction récursive possède une version itérative.

Exemple.

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

Version itérative :

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

Application. Conversion en version récursive terminale.

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

Dérécursivation

Toute fonction récursive possède une version itérative.

Exemple.

Version non terminale :

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

$$n! = \underleftarrow{n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1}.$$

Version terminale :

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

$$n! = \underrightarrow{n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1}.$$

Complexité spatiale et temporelle

Analyser un algorithme, c'est :

- Prouver sa terminaison.
- Prouver sa correction.
- Évaluer son coût spatial et temporel.

Complexité spatiale et temporelle

Analyser un algorithme, c'est :

- Prouver sa terminaison.
- Prouver sa correction.
- Évaluer son coût spatial et temporel.

Complexité spatiale : évaluation de la quantité de mémoire utilisée ;

Complexité temporelle : évaluation du temps d'exécution.

Complexité spatiale et temporelle

Analyser un algorithme, c'est :

- Prouver sa terminaison.
- Prouver sa correction.
- Évaluer son coût spatial et temporel.

Complexité spatiale : évaluation de la quantité de mémoire utilisée ;

Complexité temporelle : évaluation du temps d'exécution.

Analyse *algorithmique* : ne pas prendre en compte les performances matérielles et logicielles, en utilisant une mesure indépendante de tout aspect technique conjoncturel et qui soit pertinente au regard du problème posé.

Analyse asymptotique

On ne cherche en général qu'un *ordre de grandeur* de la complexité.

Exemple : calcul du n^e terme de la suite de FIBONACCI.

Analyse asymptotique

On ne cherche en général qu'un *ordre de grandeur* de la complexité.

Exemple : calcul du n^e terme de la suite de Fibonacci.

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

Mesure de la complexité temporelle : nombre c_n d'additions effectuées.

$c_0 = c_1 = 0$ et $c_n = c_{n-1} + c_{n-2} + 1$ donc $c_n = f_{n+1} - 1 \sim \dfrac{\varphi^{n+1}}{\sqrt{5}}$ avec $\varphi \approx 1{,}618$. Un tel coût est exponentiel.

Mesure de la complexité spatiale : nombre d_n d'appels récursifs non terminaux.

$d_0 = d_1 = 1$ et $d_n = d_{n-1} + d_{n-2} + 1$ donc $d_n = 2f_{n+1} - 1 \sim \dfrac{2\varphi^{n+1}}{\sqrt{5}}$; le coût spatial est aussi exponentiel.

Analyse asymptotique

On ne cherche en général qu'un *ordre de grandeur* de la complexité.

Exemple : calcul du n^{e} terme de la suite de FIBONACCI.

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

Mesure de la complexité temporelle : nombre c_n d'additions effectuées.
$c_n = n - 1 \sim n$. Un tel coût est linéaire.
Autre coût temporel : la création du vecteur **t** est aussi linéaire.

Mesure de la complexité spatiale : quantité de mémoire allouée au vecteur **t**. Elle est aussi linéaire.

Analyse asymptotique

On ne cherche en général qu'un *ordre de grandeur* de la complexité.

Exemple : calcul du n^e terme de la suite de FIBONACCI.

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

Mesure de la complexité temporelle : nombre c_n d'additions effectuées.
$c_n = n$, le coût est linéaire.

Mesure de la complexité spatiale : coût constant en mémoire (appels récursifs terminaux).

Notations de Landau

- $u_n = \mathrm{O}(v_n) : \exists M > 0 \mid u_n \leqslant M v_n$ (u_n est *dominée* par v_n).
- $u_n = \Omega(v_n) : \exists M > 0 \mid u_n \geqslant M v_n$ (u_n *domine* v_n).
- $u_n = \Theta(v_n) : u_n = \mathrm{O}(v_n)$ et $u_n = \Omega(v_n)$ (u_n et v_n ont même ordre de grandeur).

Objectif : obtenir l'ordre de grandeur de la complexité.

Notations de Landau

- $u_n = \mathrm{O}(v_n) : \exists M > 0 \mid u_n \leqslant M v_n$ (u_n est *dominée* par v_n).
- $u_n = \Omega(v_n) : \exists M > 0 \mid u_n \geqslant M v_n$ (u_n *domine* v_n).
- $u_n = \Theta(v_n) : u_n = \mathrm{O}(v_n)$ et $u_n = \Omega(v_n)$ (u_n et v_n ont même ordre de grandeur).

Objectif : obtenir l'ordre de grandeur de la complexité.

Un coût c_n est dit :

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

Notations de LANDAU

	$\log n$	n	$n\log n$	n^2	n^3	2^n
10^2	0,7 ns	1 ns	5 ns	0,1 µs	10 µs	$4\cdot10^{11}$ a
10^3	1 ns	10 ns	0,7 µs	10 µs	10 ms	10^{290} a
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 j	

Temps requis pour effectuer n opérations suivant le coût

$\log n$	n	$n\log n$	n^2	n^3	2^n
$10^{1.8\,10^{12}}$	$6\cdot10^{12}$	$1{,}6\cdot10^{11}$	$2{,}4\cdot10^{6}$	18171	42

valeur atteinte en une minute

Un coût c_n est dit :

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

Différents types de complexité

On note $\mathscr{D}_n$ l'ensemble des données de taille n et $c(d)$ la complexité pour la donnée d. On définit :

- la complexité *dans le meilleur des cas* $C_{\min} = \min\limits_{d \in \mathscr{D}_n} c(d)$;
- la complexité *dans le pire des cas* $C_{\max} = \max\limits_{d \in \mathscr{D}_n} c(d)$;
- la complexité *en moyenne* $C_{\text{moy}} = \sum\limits_{d \in \mathscr{D}_n} p(d)c(d)$;

où $p(d)$ est la probabilité d'apparition de la donnée d.

Différents types de complexité

On note $\mathscr{D}_n$ l'ensemble des données de taille n et $c(d)$ la complexité pour la donnée d. On définit :

- la complexité *dans le meilleur des cas* $C_{\text{min}} = \min\limits_{d \in \mathscr{D}_n} c(d)$;
- la complexité *dans le pire des cas* $C_{\text{max}} = \max\limits_{d \in \mathscr{D}_n} c(d)$;
- la complexité *en moyenne* $C_{\text{moy}} = \sum\limits_{d \in \mathscr{D}_n} p(d)c(d)$;

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

Différents types de complexité

On note $\mathscr{D}_n$ l'ensemble des données de taille n et $c(d)$ la complexité pour la donnée d. On définit :

- la complexité *dans le meilleur des cas* $C_{\min} = \min\limits_{d \in \mathscr{D}_n} c(d)$;
- la complexité *dans le pire des cas* $C_{\max} = \max\limits_{d \in \mathscr{D}_n} c(d)$;
- la complexité *en moyenne* $C_{\text{moy}} = \sum\limits_{d \in \mathscr{D}_n} p(d)c(d)$;

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

Dans le meilleur des cas : $C_{\min} = 0$.
Dans le pire des cas : $C_{\max} = n$.
Pour une liste quelconque l de taille n, la complexité $C(l)$ vérifie donc

$$C(l) = \Omega(1) \quad \text{et} \quad C(l) = \mathrm{O}(n).$$

Différents types de complexité

On note $\mathscr{D}_n$ l'ensemble des données de taille n et $c(d)$ la complexité pour la donnée d. On définit :

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

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

Coût en moyenne : on suppose les entiers dans $[\![0,9]\!]$, équiprobables.

Alors $C_{\text{moy}} = \dfrac{1}{10^n}\Big(\sum_{k=0}^{n-1} k \times 9^k \cdot 10^{n-1-k} + n \times 9^n\Big) = 9\Big(1 - \big(\frac{9}{10}\big)^n\Big) = \Theta(1)$.