4 Analyse des algorithmes

Deux questions peuvent se poser pour juger de la validité d'un algorithme :

- L'algorithme s'arrête-t-il ? (problème de terminaison)
- L'algorithme renvoie-t-il le résultat attendu ? (problème de correction).

La terminaison et la correction d'un algorithme ne suffisent pas à juger de l'utilisabilité d'un algorithme. En effet, un algorithme qui se finit en théorie, mais qui en pratique a un temps de réponse de plusieurs années, voire plusieurs milliards d'années, est assez inutilisable en pratique. Il est donc important de pouvoir estimer le temps de réponse d'un algorithme, afin de juger de sa qualité. La recherche d'algorithmes les plus rapides possibles dans un contexte donné est un des enjeux majeurs de l'informatique.
C'est l'étude de la complexité qui nous permettra de savoir si un algorithme répond suffisamment rapidement pour pourvoir être utilisable dans les conditions qu'on lui impose (notamment suivant la taille des données).

I Validité d'un algorithme

I.1 Terminaison d'un algorithme

Dans un algorithme **non récursif** correctement structuré, et sans erreur de programmation (par exemple modification manuelle d'un compteur de boucle `for`), les seules structures pouvant amener une non terminaison d'un algorithme sont les boucles `while` et `repeat`. Nous n'évoquerons pas dans ce cours la terminaison et la correction d'un algorithme récursif (étude se ramenant le plus souvent à une récurrence).
Pour l'étude de la terminaison, partons de l'exemple de l'algorithme d'Euclide (algorithme 4.1).

Algorithme 4.1 : Euclide

Entrée : a, b : entiers positifs
Sortie : d : entier

tant que $b > 0$ **faire**
  $(a, b) \leftarrow (b, a \mod b)$
fin tant que
renvoyer a

La preuve mathématique classique de la terminaison de l'algorithme d'Euclide passe par le principe de descente infinie : les restes successifs non nuls (c'est-à-dire les valeurs successives de b) forment une séquence strictement décroissante d'entiers positifs (sauf éventuellement à la première étape si la valeur initiale de b est négative ; mais dans ce cas, le premier reste est positif). Le principe de descente infinie assure alors que la séquence est finie, donc que l'algorithme finit par s'arrêter.

Plus précisément, si on veut couler cette preuve dans le moule de la démonstration par descente infinie de

Fermat, on peut partir d'un couple (a, b) tel que $a \geqslant 0$ et tel que l'algorithme d'Euclide ne s'arrête pas. Alors l'algorithme d'Euclide ne s'arrête pas non plus pour $(a', b') = (b \mod a, a)$, et on a $0 \leqslant a' < a$. On a donc trouvé une situation similaire pour une valeur entière positive strictement plus petite de a, ce qui amène la contradiction voulue par le principe de descente infinie.

Cet exemple est l'archétype d'une preuve de terminaison. On prouve la terminaison d'une boucle en exhibant un variant de boucle :

Définition 4.1.1 (Variant de boucle)

On appelle variant de boucle une quantité v définie en fonction des variables $(x_1, \ldots, x_k)$ constituant l'état de la machine, et de n, le nombre de passages effectués dans la boucle et telle que :

- v ne prenne que des valeurs entières ;
- la valeur de v en entrée de boucle soit toujours positive ;
- v prenne des valeurs strictement décroissantes au fur et à mesure des passages dans la boucle : ainsi, si v_n désigne la valeur de v au n-ième passage dans la boucle, si les passages n et $n+1$ ont lieu, on a $v_{n+1} < v_n$.

Le principe de descente infini permet alors d'affirmer que :

Théorème 4.1.2 (Terminaison d'une boucle, d'un algorithme)

1. *Si, pour une boucle donnée, on peut exhiber un variant de boucle, alors le nombre de passages dans la boucle est finie.*
2. *Si, pour un algorithme donné, on peut exhiber, pour toute boucle de l'algorithme, un variant de boucle, alors l'algorithme s'arrête en temps fini.*

Avertissement 4.1.3

Il faut bien justifier la validité du variant de boucle pour toute valeur possible d'entrée.

Exemple 4.1.4

Dans le cas de l'algorithme d'Euclide, un variant de boucle simple est donné simplement par la variable a, à partir du rang 1 (le premier passage permet de se ramener au cas où $a \geqslant 0$, ce qui n'est pas nécessairement le cas initialement).

Remarque 4.1.5

La terminaison assure un arrêt théorique de l'algorithme. En pratique, un deuxième paramètre entre en jeu : le temps de réponse. Ce temps de réponse sans être infini, peut être très grand (certains algorithmes exponentiels peuvent avoir un temps de réponse de l'ordre de milliards d'années pour des données raisonnablement grandes : c'est le cas par exemple du calcul récursif de la suite de Fibonacci donné ci-dessus, pour des valeurs de n restant raisonnables). Évidemment dans ce cas, même si l'algorithme s'arrête en théorie, c'est d'assez peu d'intérêt, et quand il finit par s'arrêter, on ne se souvient plus de la question posée...

Remarque 4.1.6

Dans le cas d'une boucle `for`, on peut toujours construire un variant simple. Si la boucle est donnée par la structure :
`Pour` $i \leftarrow a$ `à` b,
un variant simple est $b - i$

Évidemment, ceci ne fait que traduire un fait intuitivement évident : le nombre de passages dans une boucle `for` est évidemment fini, égal au nombre de valeurs que le compteur de boucle peut prendre. Dans la pratique, cela nous dispense d'un argument de terminaison pour ce type de boucles.

En guise d'exemple développé, nous traitons l'étude de la terminaison de l'algorithme 4.2.

Algorithme 4.2 : Mystère

Entrée : a,b : entiers
Sortie : q, r : entiers

$r \leftarrow a$;
$q \leftarrow 0$;
si $b = 0$ **alors**
  Erreur
sinon
  tant que $r \geqslant b$ **faire**
    $r \leftarrow r - b$;
    $q \leftarrow q + 1$;
  fin tant que
  renvoyer q,r
fin si

Le variant de boucle est à contruire à partir du test de continuation de la boucle. Puisqu'on s'arrête dès que $r - b < 0$, que $r - b$ prend des valeurs entières, on peut considérer la quantité $v = r - b$.

- Toutes les opérations étant entières, v est bien un entier
- Par définition même d'une boucle `while`, à l'entrée dans une boucle, la valeur de v est positive
- Il reste à étudier la décroissance stricte. Si on note v_n la valeur de v à l'entrée dans la n-ième boucle, et si on passe effectivement dans les boucles d'indice n et $n + 1$, alors $v_{n+1} = v_n - b$.

À ce stade, on se rend compte qu'il y a un problème : v est bien un variant de boucle pour une valeur initiale de b strictement positive, mais pas pour une valeur négative. Un petit examen de la situation nous fait comprendre qu'effectivement, si b est négatif, les valeurs de r seront de plus en plus grandes, donc le test $r \geqslant b$ sera « de plus en plus vrai ». Il y a peu de chance que l'algorithme s'arrête.
Il est donc nécessaire de rectifier notre algorithme, soit en excluant les valeurs négatives en même temps que la valeur nulle, soit en adaptant l'algorithme. À ce stade, vous aurez bien sûr compris ce que calcule l'algorithme `Mystère` : il ne s'agit de rien de plus que le quotient et le reste de la division euclidienne. Or, ces quantités sont aussi définies pour b strictement négatif, avec la petite adaptation suivant : le quotient q et le reste r de la division euclidenne de a par b sont les uniques entiers tels que

$$a = bq + r \qquad \text{où} \qquad r \in [\![0, |b| - 1]\!].$$

On a fait cette adaptation dans l'algorithme 4.3.
Revenons à notre preuve de terminaison. Maintenant la quantité $v = r - |b|$ est bien entière, positive en entrée de boucle (si on entre dans une boucle, dans le cas contraire, la terminaison est assurée !), et strictement décroissante d'un passage à l'autre dans la boucle (puisqu'on lui retire la quantité strictement positive $|b|$ à chaque étape, le cas $b = 0$ ayant été écarté initialement).
Ceci prouve la terminaison de l'algorithme 4.3, mais pas sa correction qui nous réserve encore des surprises.

I.2 Correction d'un algorithme

La terminaison d'un algorithme n'assure pas que le résultat obtenu est bien le résultat attendu, c'est-à-dire que l'algorithme répond bien à la question posée initialement. L'étude de la validité du résultat qu'on obtient fait l'objet de ce paragraphe. On parle de *l'étude de la correction de l'algorithme.*

Algorithme 4.3 : Division euclidienne ?

Entrée : a,b : entiers
Sortie : q, r : entiers

$r \leftarrow a$;
$q \leftarrow 0$;
si $b = 0$ **alors**
  Erreur
sinon
  tant que $r \geqslant |b|$ **faire**
    $r \leftarrow r - |b|$;
    $q \leftarrow q + 1$;
  fin tant que
  si $b < 0$ **alors**
    $q \leftarrow -q$
  fin si
  renvoyer q,r
fin si

Reprenons l'exemple de l'algorithme d'Euclide 4.1. Cet algorithme a pour objet le calcul du pgcd de a et b. La preuve mathématique classique de la correction de l'algorithme repose sur la constatation suivante : si r est le reste de la division euclidienne de a par b, alors $a \wedge b = b \wedge r$. La preuve mathématique de ce point ne pose pas de problème.
On constate alors que la quantité $w = a \wedge b$ prend toujours la même valeur à l'entrée d'une boucle (correspondant aussi à la valeur à la sortie de la boucle précédente). C'est le fait d'avoir une valeur constante qui nous assure que la valeur initiale recherchée $a \wedge b$, est égale à $a \wedge b$ pour les valeurs finales obtenues pour a et b. Or, en sortie de boucle $b = 0$, donc $a \wedge b = a$. Ainsi, le pgcd des valeurs initiales de a et b est égal à la valeur finale de a.

De façon générale, l'étude de la correction d'un algorithme se fera par la recherche d'une quantité invariante d'un passage à l'autre dans la boucle :

Définition 4.1.7 (Invariant de boucle)

On appelle invariant de boucle une quantité w dépendant des variables $x_1, \ldots, x_k$ en jeu ainsi éventuellement que du compteur de boucle n, telle que :

- la valeur prise par w en entrée d'itération, ainsi que la valeur qu'aurait pris w en cas d'entrée dans l'itération suivant la dernière itération effectuée, soit constante.
- Si le passage initial dans la boucle est indexé 0 et que la dernière itération effectuée est indexée $n-1$, l'égalité $w_0 = w_n$ permet de prouver que l'algorithme retourne bien la quantité souhaitée.

Exemple 4.1.8

Dans le cas de l'algorithme d'Euclide, la quantité $w = a \wedge b$ est un invariant de boucle : il est constant, et à la sortie, puisque $b = 0$, il prend la valeur a. Ainsi, la valeur retournée par l'algorithme est bien égal à $a \wedge b$, pour les valeurs initiales de a et b, passées en paramètres.

Étudions encore une fois le cas de l'algorithme de la division euclidenne 4.3. On sait que le quotient et le reste doivent satisfaire à la relation $a = bq + r$. Il semble donc naturel de considérer $w = \varepsilon bq + r = |b|q + r$ comme invariant de boucle, où ε est le signe de b (cela permet de gérer le changement final de signe effectué sur q à la fin de l'algorithme). On note w_k la valeur de w à l'entrée dans l'itération d'indice k, la boucle initiale étant d'indice 0. On note de même q_k et r_k les valeurs de q et r à l'entrée dans l'itération k.

- D'après les initialisations de q et r, on a $w_0 = a$
- Si le passage dans l'itération k est effectué, on ajoute 1 à q et on retranche $|b|$ à r. Ainsi :

$$w_{k+1} = |b|q_{k+1} + r_{k+1} = |b|(q_k + 1) + r_k - |b| = |b|q_k + r_k = w_k.$$

  Ainsi, (w) est bien constant.
- À la sortie de structure, en appelant n l'indice de la première boucle non exécutée, on a $r_n < |b|$ et donc :

$$w_n = q_n|b| + r_n = qb + r, \qquad \text{avec } r < |b|$$

  où q et r sont les valeurs retournées (avec changement de signe éventuel). On n'a pas encore tout à fait répondu au problème, puisqu'on doit aussi s'assurer que $r \geqslant 0$. Cette dernière propriété peut être obtenue facilement si on peut justifier que $r_{n-1} \geqslant |b|$. Dans ce cas, $r_n = r_{n-1} - |b|$. On peut aussi l'obtenir facilement lorsque $a \in [\![0, |b|-1]\!]$ (dans ce cas, il n'y a pas de passage dans la boucle, et $n = 0$). En revanche, si r peut se retrouver à avoir des valeurs négatives, on reste coincé. Or, c'est ce qu'il se passe si la valeur initiale de r est négative, donc si $a \leqslant 0$.

La recherche de l'invariant de boucle nous a donc permis de détecter l'erreur faite dans l'algorithme 4.3 : celui n'est valable que pour les valeurs positives de a. Nous pouvons donc maintenant donner une version corrigée de l'algorithme de la division euclidienne (algorithme 4.4)

Algorithme 4.4 : Division euclidienne, version corrigée

```
Entrée : a,b : entiers
Sortie : q, r : entiers

r ← a ;
q ← 0 ;
si b = 0 alors
    Erreur
sinon
    si a ⩾ 0 alors
        tant que r ⩾ |b| faire
            r ← r − |b| ;
            q ← q + 1 ;
        fin tant que
        si b < 0 alors
            q ← −q
        fin si
        renvoyer q,r
    sinon
        tant que r < 0 faire
            r ← r + |b| ;
            q ← q − 1 ;
        fin tant que
        si b < 0 alors
            q ← −q
        fin si
        renvoyer q,r
    fin si
fin si
```

On pourrait tout de même regrouper les deux cas en combinant les deux tests (en se rendant compte que dans les deux cas, on tombera forcément dans le « trou »), et en gérant le changement de signe par un ε égal à 1 et à -1, mais, si cela raccourcit le code, cela augmente en revanche le nombre d'opérations et de tests. Nous en resterons donc à la version de l'algorithme 4.4.

Nous reprenons donc avec ce nouvel algorithme l'étude de la terminaison et de la correction correction. Nous avons maintenant deux boucles à étudier.

1. Étude de la boucle du cas $a \geqslant 0$:
   - L'étude de la terminaison de cette boucle a déjà été faite.
   - Nous terminons l'étude de correction précédente en constatant que si $a \in [\![0, |b| - 1]\!]$, il n'y a pas de passage dans la boucle, et la valeur retournée est $q = 0$ et $r = a \in [\![0, |b| - 1]\!]$ qui répond bien au problème, et que si $a \geqslant |b|$, il y a au moins un passage dans la boucle. Ainsi, avec les notations précédentes, $n \geqslant 1$, et $r_{n-1} \geqslant |b|$ (sinon on ne serait pas passé dans la boule $n-1$, d'où $r_n = r_{n-1} - |b| \geqslant 0$. Cela fournit donc le point qui manquait précédemment pour conclure que (w) est un invariant de boucle
2. Étude de la boucle du cas $a < 0$.
   - Cette fois, $-r$ est clairement un variant de boucle assurant la terminaison (il est entier, positif tant qu'on entre dans la boucle, et décroissant puisqu'on ajoute à r une valeur strictement positive à chaque étape)
   - Nous prouvons de même que ci-dessus que $w = |b|q + r$ est un invariant pour la boucle dans le cas $a < 0$ (chaque fois qu'on ajoute $|b|$ à r, on retranche 1 à q par compensation). À la sortie multiplie q par le signe de b, donc on a la relation

     $$a = |b|q_0 + r_0 = |b|q_n + r_n = bq + r,$$

     et de plus, vu le test initial, on passe au moins une fois dans la boucle, et $r_{n-1} < 0$ (sinon on ne serait pas passé dans la boucle $n-1$, donc $r_n < |b|$. Comme on ne passe pas dans la boucle n, on a de plus $r_n \geqslant 0$. Ainsi, les valeurs finales de q et r, vérifient bien les propriétés attendues caractérisant la division, ce qui assure la correction de l'algorithme.

Retenons de cette étude que non seulement, l'étude de la terminaison et de la correction d'un algorithme permet de prouver sa validité, mais que par ailleurs, cette étude permet de détecter des erreurs dans l'algorithme le cas échéant.

Remarque 4.1.9

Un invariant de boucle peut très bien être un booléen, c'est-à-dire la valeur de vérité d'une propriété. Ainsi, on peut définir comme invariant de boucle, le fait qu'une propriété $\mathcal{P}(n)$ dépendant du rang n de passage dans la boucle soit toujours vraie à l'entrée de boucle.

Nous illustrons cette possibilité sur l'exemple suivant (algorithme 4.5).

Algorithme 4.5 : Selection du minimum

Entrée : T : tableau
Sortie : T :tableau

pour $i \leftarrow 1$ **à** `Taille` *(T) - 1* **faire**
  si $T[i] < T[0]$ **alors**
    $T[0], T[i] \leftarrow T[i], T[0]$
  fin si
fin pour

La terminaison de cet algorithme ne pose pas de problème (il n'y a pas de boucle `repeat` ou `while`). Pour la correction, on se convainc sans peine que l'algorithme va rechercher la valeur minimale (ou une des valeurs minimales) du tableau T et la placer en tête de tableau, en modifiant éventuellement l'ordre des autres éléments. On le prouve en considérant l'invariant de boucle $\mathcal{P}(n)$: « à l'entrée dans la boucle de rang $n \geqslant 1$, $T[0]$ est le minimum des $T[i]$ pour $i \in [\![0, n-1]\!]$ ».
L'invariant $\mathcal{P}(n)$ est vrai pour $n = 1$ (initialement, $T[0]$ est le minimum du sous-tableau constitué de l'unique case $T[0]$), et si $\mathcal{P}(n)$ est vraie, alors $\mathcal{P}(n+1)$ aussi, car à l'entrée dans la boucle n, soit $T[0]$ est le

minimum de $T[i]$, $i \in [\![0, n]\!]$, et il le reste alors à la sortie (pas de modification faite), donc à l'entrée dans la boucle suivante ; soit $T[0]$ n'est pas le minimum des $T[i]$, $i \in [\![0, n]\!]$, mais comme il est le minimum des $T[i]$, $i \in [\![0, n-1]\!]$, on a alors $T[n] < T[0]$, et c'est $T[n]$ le minimum recherché. On fait alors l'échange de $T[0]$ et $T[n]$, ce qui place à l'issue de la boucle le minimum en place 0 du tableau, assurant que $\mathcal{P}(n+1)$ est vérifié.
Ainsi, $\mathcal{P}(n)$ est toujours vrai, par principe de récurrence, et pour la dernière valeur de n (taille t du tableau, c'est la valeur d'entrée dans l'itération suivante, celle qui n'a pas lieu), on récupère que $T[0]$ est le minimum des $T[i]$, pour $i \in [\![0, t-1]\!]$, ce qui prouve la correction de l'algorithme.

Même si le fait d'utiliser un « invariant » qui ait l'air plus compliqué que précédemment (une proposition), il ne s'agit que d'un cas particulier de la définition, l'invariant étant ici tout simplement le booléen $\mathcal{P}(n)$.
Dans cette situation, montrer la correction d'un algorithme se fait en 3 étapes :

- initialisation : montrer que l'invariant est vrai avant la première itération.
- conservation : montrer que si l'invariant est vrai avant une itération, il reste vrai avant l'itération suivante.
- terminaison : déduire de l'invariant final la propriété voulue.

II Complexité en temps (modèle à coûts fixes)

II.1 Première approche

Chaque opération effectuée nécessite un temps de traitement. Les opérations élémentaires effectuées au cours d'un algorithme sont :

- l'affectation
- les comparaisons
- les opérations sur des données numériques.

Évaluer la complexité en temps revient alors à sommer tous les temps d'exécution des différentes opérations effectuées lors de l'exécution d'un algorithme.
La durée d'exécution des différentes opérations dépend de la nature de l'opération : une multiplication sur un flottant demande par exemple plus de temps qu'une addition sur un entier. Ainsi, si on veut une expression précise du temps de calcul connaissant la durée d'exécution de chaque opération, il nous faut toute une série de constantes donnant ces différents temps.

Définition 4.2.1 (Modèle à coût fixe)

Le modèle à coût fixe consiste à considérer que la durée des opérations ne dépend pas de la taille des objets (notamment des entiers), et que les opérations sur les flottants sont de durée similaire aux opérations semblables sur les entiers.

Évidemment, suivant la situation, le modèle à coût fixe peut être acceptable ou non : si on est amené à manipuler de très longs entiers, il est évident que le modèle à coût fixe n'est pas bon : multiplier deux entiers à 1.000.000.000 chiffres est plus long que multiplier deux entiers à 1 chiffre !

Nous supposons ici que nous sommes dans une situation ou le modèle à coût fixe est acceptable. Nous nous donnons alors des constantes C_+, C_*, $C_/$ etc correspondant au temps des différentes opérations, ainsi que $C_\leftarrow$ pour l'affectation et $C_{==}$, $C_{<=}$ etc. pour les tests. Ces différentes valeurs peuvent être distinctes, mais sont toutes strictement positives.

Définition 4.2.2 (Complexité en temps)

La complexité en temps d'un algorithme est une fonction C dépendant de la taille n des données (éventuellement de plusieurs variables s'il y a en entrée des objets pouvant être de tailles différentes) et estimant le temps de réponse en fonction de n et des constantes $C_\bullet$ définies ci-dessus.

Étudions par exemple l'algorithme suivant :

Algorithme 4.6 : Évaluation naïve d'un polynôme

Entrée : T : tableau, coefficients d'un polynôme P, a : réel
Sortie : $P(a)$: réel.

$S \leftarrow 0$;
pour $i \leftarrow 0$ **à** n **faire**
  $b \leftarrow 1$;
  pour $j \leftarrow 1$ **à** i **faire**
    $b \leftarrow b \times a$
  fin pour
  $S \leftarrow S + T[i] \times b$
fin pour
renvoyer S

Exercice 2

1. Justifier la correction de l'algorithme ci-dessus.
2. En considérant que l'incrémentation de l'indice des boucles nécessite une addition et une affectation, montrer que pour un polynôme de degré n (donc un tableau indexé de 0 à n), la complexité en temps est :

$$C(n) = n^2 \left(\frac{C_+}{2} + \frac{C_*}{2} + C_\leftarrow \right) + n \left(4C_\rightarrow + \frac{3}{2}C_+ + \frac{3}{2}C_* \right) + (4C_\leftarrow + C_*)\,.$$

Une telle expression aussi précise peut être importante si on a à estimer de façon très précise le temps d'exécution pour des tableaux de taille connue. Cependant, dans l'étude du comportement asymptotique, on n'est souvent intéressé que par l'ordre de grandeur de la complexité. Ici, du fait que le coefficient précédant le terme en n^2 est strictement positif, on obtient :

$$C(n) = \Theta(n^2).$$

On dit que l'algorithme a un coût, ou une complexité, quadratique. Les différents comportements de référence sont :

Définition 4.2.3 (Complexités de référence)

- Coût constant : $C(n) = \Theta(1)$
- Coût logarithmique : $C(n) = \Theta(\ln(n))$
- Coût linéaire : $C(n) = \Theta(n)$
- Coût quasi-linéaire : $C(n) = \Theta(n \ln(n))$
- Coût quadratique : $C(n) = \Theta(n^2)$
- Coût cubique : $C(n) = \Theta(n^3)$
- Coût polynomial : $C(n) = \Theta(n^\alpha)$, $\alpha > 0$
- Coût exponentiel : $C(n) = \Theta(x^n)$, $x > 1$.

On utilisera la même terminologie (en précisant éventuellement « au plus ») si on ne dispose que d'un O au lieu du Θ.

Pour information, voici les temps approximatifs de calcul pour un processeur effectuant 1 milliard d'opérations par seconde, et pour une donnée de taille $n = 10^6$, suivant les coûts :

- $O(1)$: 1 ns
- $O(\ln(n))$: 15 ns
- $O(n)$: 1 ms

- $O(n \ln n)$: 15 ms
- $O(n^2)$: 15 min
- $O(n^3)$: 30 ans
- $O(2^n)$: 10^{300000} milliards d'années !

Ainsi, si on est amené à traiter des grandes données, l'amélioration des complexités est un enjeu important !

Remarque 4.2.4

En pratique, se rendre compte de l'évolution de la compexité n'est par très dur : pour des données de taille suffisamment grande, en doublant la taille des données :

- on n'augmente pas le temps de calcul pour un algorithme en temps constant
- on augmente le temps de calcul d'une constante (indépendante de n) pour un coût logarithmique
- on double le temps de calcul pour un algorithme linéaire
- on quadruple le temps de calcul pour un algorithme quadratique
- on multiplie le temps de calcul par 8 pour un algorithme cubique
- ça ne répond plus pour un algorithme exponentiel (sauf si on a pris une valeur initiale n trop ridiculement petite)

II.2 Simplification du calcul de la complexité

Le calcul de la complexité effectué ci-dessus nécessite de distinguer les différentes opérations. De plus, les valeurs des différentes constantes ne sont pas des données absolues : elles dépendent de façon cruciale du processeur utilisé ! Nous voyons dans ce paragraphe comment s'affranchir de la connaissance de ces constantes.

Dans la recherche de la complexité asymptotique sous forme d'un Θ, la connaissance précise des $C_\bullet$ n'est pas indispensable : ces constantes interviennent, comme dans l'exemple traité, dans les coefficients de l'expression obtenu, coefficients qu'il n'est pas utile de connaître explicitement pour obtenie une expression en Θ (seule leur stricte positivité est indispensable)

Théorème 4.2.5

À condition de garder des valeurs strictement positives, on peut modifier la valeur des constantes $C_\bullet$ comme on veut, sans changer le comportement en $\Theta(u_n)$ de la complexité.

◁ **Éléments de preuve.**

Utiliser le fait que la fonction temps de réponse est croissante en fonction du temps de chacune des opérations, ainsi que sa linéarité : si on multiplie tous les temps d'opérations par λ, le temps total aussi (cela revient à faire un changement d'échelle temporelle).

Étant donnés deux jeux $(t_1, \dots, t_k)$ et $(t'_1, \dots, t'_k)$ de constantes, et en considérant le minimum et le maximum des t_i et des t_i, il est alors possible de trouver deux constantes strictement positives λ et μ telles que

$$\lambda C(t_1, \dots, t_k, n) \leqslant C(t'_1, \dots, t'_k, n) \leqslant \mu C(t_1, \dots, t_k, n).$$

Pour le première inégalité, s'arranger par exemple pour que le maximum des λt_i soit inférieur au minimum des t'_j. ▷

Corollaire 4.2.6 (Calcul simplifié de la complexité asymptotique)

On peut calculer le comportement asymptotique en Θ de la complexité en faisant la supposition que toutes les constantes de temps $C_\bullet$ sont égales à 1.

Autrement dit, le calcul de complexité asymptotique peut se faire sous la supposition que toutes les opérations ont le même temps d'exécution et on peut prendre ce temps d'exécution comme unité temporelle. On peut même considérer, pour simplifier, l'incrémentation d'un compteur de boucle comme nécessitant une seule unité de temps.

Évidemment, ce principe est aussi valable pour obtenir juste un O ou un Ω.

Exemple 4.2.7

Reprendre le calcul de la complexité asymptotique de l'évaluation naïve d'un polynôme sous cet angle.

Remarque 4.2.8

On peut alors considérer une succession d'un nombre fini fixé d'opérations élémentaires comme une nouvelle opération plus complexe, et compter alors pour 1 cette succession. Par exemple, si une boucle contient un nombre déterminé d'instructions ayant elles-même un coût ne dépendant pas de l'itération, on peut compter ce bloc d'instructions comme une seule instruction. Il suffit dans ce cas de compter le nombre de passages dans la boucle Cet argument de simplification est à préciser un peu dans la rédaction.

II.3 Amélioration de la complexité de l'exemple donné

Un même problème peut souvent être résolu par différents algorithmes pouvant avoir des complexités différentes. Cette recherche de la performance est un enjeu capital de l'informatique. Nous l'illustrons sur notre exemple de l'évaluation polynomiale, en donnant deux améliorations possibles, et en recherchant leur complexité.

La première amélioration que nous proposons est le calcul rapide des puissances de a, basé sur la remarque que $a^{16} = (((a^2)^2)^2)^2$, permettant de la sorte de passer de 15 opérations à 4. De façon plus générale, il faut tenir compte de l'imparité possible de l'exposant.

Fonction 4.7 : exp-rapide(a,n)

Entrée : a entier ou réel, n entier
Sortie : a^n
si $n = 0$ **alors**
  renvoyer *1* et sortir
fin si
$y \leftarrow a$;
$z \leftarrow 1$;
tant que $n > 1$ **faire**
  si n *est impair* **alors**
    $n \leftarrow n - 1$;
    $z \leftarrow z * y$
  sinon
    $n \leftarrow \frac{n}{2}$;
    $y \leftarrow y * y$
  fin si
fin tant que
renvoyer $y * z$

Algorithme 4.8 : Évaluation polynomiale par exponentiation rapide

Entrée : T : tableau des coefficients d'un polynôme P, a réel
Sortie : $P(a)$
$S \leftarrow 0$;
pour $i \leftarrow 0$ **à** n **faire**
  $S \leftarrow S + T[i] \times$ exp-rapide(a, i)
fin pour

Exercice 3

1. Justifier la terminaison et la correction de l'algorithme d'exponentiation rapide (on pourra considérer l'invariant $y^n z$).
2. Soit $C(n)$ le coût de l'exponentiation rapide pour un exposant n. Montrer que pour tout $k \in \mathbb{N}$ et tout $n \in [\![2^k, 2^{k+1}]\!]$,
$$C(2^k) \leqslant C(n) < C(2^{k+1}) + 6k.$$
3. En déduire que $C(n) = \Theta(\ln(n))$.
4. En déduire que l'algorithme d'évaluation polynomial obtenu ainsi a un coût en $\Theta(n \ln(n))$. Pour la minoration, on pourra se restreindre dans un premier temps à des indices entre $\frac{n}{2}$ et n.

Mais il est évidemment possible de faire mieux : ayant déjà calculé a^{i-1} au passage précédent dans une boucle, il est inutile de reprendre le calcul de a^i depuis le début : il suffit de multiplier la puissance obtenue précédemment par a, ce qui ne nécessite qu'une opération supplémentaire.
Cette remarque peut se traduire par l'égalité suivante, dans laquelle on a mis dès que possible les termes x en facteur :

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n))).$$

Cette factorisation est à la base de l'algorithme de Hörner :

Algorithme 4.9 : Hörner

Entrée : T, coefficients d'un polynôme P, a : réel
Sortie : $P(a)$
$S \leftarrow T[n]$;
pour $i \leftarrow n-1$ **descendant à** 0 **faire**
  $S \leftarrow S * a + T[i]$
fin pour
renvoyer (S)

Exercice 4

1. Montrer la correction de l'algorithme de Hörner.
2. Montrer que l'algorithme de Hörner a un coût linéaire ($\Theta(n)$).

III Complexité dans le meilleur ou le pire des cas, en moyenne

III.1 Complexité dans le meilleur et le pire des cas

Il n'est parfois pas possible d'exprimer la complexité (que ce soit par une expression exacte, ou par une estimation asymptotique) pour une entrée quelconque. En effet, le nombre d'opérations effectuées, est parfois très dépendante des valeurs fournies en entrée : des valeurs de même taille peuvent parfois nécessiter des temps de calcul très différents. Un exemple très simple est fourni par le test le plus élémentaire de primalité :

Algorithme 4.10 : Test de primalité

Entrée : p : entier > 1
Sortie : b booléen, résultat du test de primalité
$i \leftarrow 2$;
$m = \sqrt{p}$;
tant que $i \leqslant m$ **faire**
  si $p \mod i = 0$ **alors**
    renvoyer *False* et sortir
  fin si
  $i \leftarrow i + 1$
fin tant que
renvoyer *True*

Exercice 5

1. Justifier la terminaison et la correction de cet algorithme
2. Montrer que si p est pair (ce qui peut arriver pour p grand !) le temps de réponse est en $\Theta(1)$ (en négligeant le coût du calcul de $\sqrt{p}$)
3. Montrer que si p est premier (Euclide m'a affirmé un jour qu'on peut aussi trouver des p aussi grand qu'on veut vérifiant cela), le temps de réponse est en $\Theta(\sqrt{p})$ (dans le modèle à coût constant, qui devient discutable ici pour des grandes valeurs de p).

Définition 4.3.1 (Complexité dans le pire et le meilleur des cas)

- Pour des objets de taille n, la complexité dans le meilleur des cas est le nombre minimal $C_-(n)$ d'opérations à effectuer pour des objets dont la taille est n.
- Pour des objets de taille n, la complexité dans le pire des cas est le nombre maximal $C_+(n)$ d'opérations à effectuer pour des objets dont la taille est n.

Pour un objet T donnée de taille n, on a donc toujours :

$$C_-(n) \leqslant C(T) \leqslant C_+(n),$$

où $C(T)$ désigne le nombre d'opérations à effectuer pour l'objet T.

Exemple 4.3.2

En considérant que la taille des entiers est donnée par le nombre de leur chiffres (donc en gros par $\log(p)$), et en remarquant que pour tout n, il existe des nombres pairs à n chiffres (ça, c'est trivial), et également des nombres premiers à n chiffres (ça l'est un peu moins, ça peut être vu comme une conséquence du postulat de Bertrand affirmant qu'il y a toujours un nombre premier entre m et $2m$), on obtient, pour l'algorithme de primalité :

$$C_-(n) = \Theta(1) \qquad \text{et} \qquad C_+(n) = \Theta(10^{\frac{n}{2}}).$$

Dans ce cas, même si le nombre d'opérations $C(n)$ pour des objets de taille n est impossible à définir explicitement (car dépendant de l'objet donné en entrée), on dira que la complexité vérifie :

$$C(n) = O(C_+(n)) \qquad \text{et} \qquad C(n) = \Omega(C_-(n)).$$

Ainsi, pour l'algorithme de primalité, $C(n) = \Omega(1)$ et $C(n) = O(10^{\frac{n}{2}})$.

Exercice 6

On propose une amélioration au tri à bulle vu dans un exercice du chapitre précédent, consistant à compter le nombre d'échanges effectués lors d'un passage du début à la fin du tableau, et à s'arrêter lorsqu'aucun échange n'est effectué (ce qui signifie que les termes sont dans l'ordre). Déterminer la complexité dans le meilleur et dans le pire des cas, en donnant à chaque fois un exemple de configuration initiale associée.

Remarque 4.3.3

De nombreux tris (mais pas tous) sont plus rapides sur des tableaux presque triés (obtenus par exemple en ajoutant un petit nombre d'éléments à un tableau initialement trié, c'est une situation très fréquente, correspondant par exemple à l'ajout de quelques éléments dans une base qu'on maintient triée). Certains algorithmes, réputés plus lents sur des tableaux généraux, peuvent être plus rapides dans cette situation que des algorithmes récursifs réputés plus rapides dans des situations générales. Ainsi, avant de se précipiter vers un algorithme réputé rapide, il est important d'analyser le contexte dans lequel l'algorithme doit être utilisé.

III.2 Complexité en moyenne

La complexité dans le pire et dans le meilleur des cas n'est pas forcément une donnée très représentative du comportement général d'un algorithme : il s'agit souvent de situations bien particulières, qui, certes, dans certains situations, peuvent se produire souvent (voir les tris), mais dans d'autres, peuvent rester exceptionnelles. Dans ces situations, la notion de complexité en moyenne est plus intéressante.

Définition 4.3.4

La complexité en moyenne $C_m(n)$ est la moyenne du nombre d'opérations à effectuer pour chacun des objets de taille n. Si certains objets sont plus probables que d'autres, il faut en tenir compte en pondérant convenablement. Ainsi, de façon plus formelle, cette moyenne correspond à l'espérance de la variable aléatoire $C(n)$ donnant le nombre d'opérations à effectuer, définie sur l'espace probabilité Ω_n des objets de taille n.

Très souvent, on considère la mesure de probabilité uniforme sur l'ensemble des objets.

Exercice 7

On remplit un tableau de taille N, qu'on remplit aléatoirement de valeurs entières comprises entre 1 et k. Étant donnée une valeur $a \in [\![1, k]\!]$, on effectue la recherche de a dans le tableau en considérant les éléments les uns après les autres. Déterminer la complexité moyenne de cet algorithme, en fonction des deux variables N et k vouées à devenir grandes.

III.3 Algorithmes randomisés

Il est fréquent d'introduire un aléa artificiel de sorte à éviter les cas extrêmes. En effet, les cas les meilleurs, mais aussi les pires, correspondent dans certains situations, à des situations fréquentes. Ainsi est-on parfois amené à trier des données en sens inverse, à les remettre dans le bon sens, à trier un tableau presque trié, dans un sens ou dans l'autre. Ces différentes actions nous amènent souvent à fleurter avec le meilleur des cas (ce qui n'est pas trop gênant), mais aussi avec le pire des cas (ce l'est plus).
Introduire un aléa permet parfois de se ramener (avec une probabilité forte) à un cas général, dont la complexité sera plutôt de l'ordre moyen.

Définition 4.3.5 (Algorithme randomisé)
On parle d'algorithme randomisé lorsqu'on a introduit dans l'algorithme un aléa dont le but est d'éviter les cas extrêmes.

Nous introduisons rapidement le tri rapide dont le principe général (et très vague) est le suivant :

Fonction 4.11 : trirapidenonrandomisé(T)

Entrée : T : tableau à trier
Sortie : T : tableau trié
Comparer tous les éléments autres que $T[0]$ à $T[0]$ et séparer en 2 tableaux ;
U : tableau des plus petits ;
V : tableau des plus grands ;
$U \leftarrow$ trirapidenonrandomisé(U) ;
$V \leftarrow$ trirapidenonrandomisé(V) ;
renvoyer concaténation de U, $[T[0]]$ et V.

On peut montrer, mais ce n'est pas complètement évident, que la complexité en moyenne du tri rapide est en $O(n\ln(n))$. C'est aussi la complexité dans le meilleur des cas, correspondant au cas où on divise à chaque fois le tableau en deux sous-tableaux de même taille (donc que la valeur pivot choisie se trouve au milieu).

Exercice 8
Montrer que le temps de calcul du tri rapide sur un tableau trié (dans un sens ou dans l'autre) est en $O(n^2)$.

Il s'agit d'ailleurs là de la complexité dans le pire des cas. Ainsi, le tri rapide est particulièrement mauvais sur les listes triées et presque triées, dans un sens ou dans l'autre. Comme dit plus haut, il s'agit de situations fréquentes dans la vie courante. Une façon d'y remédier est d'introduire un aléa dans le choix du pivot :

Fonction 4.12 : trirapiderandomisé(T)

Entrée : T : tableau à trier
Sortie : T : tableau trié
Choisir i aléatoirement dans $[\![0, n-1]\!]$ (indices du tableau) ;
Comparer tous les éléments autres que $T[i]$ à $T[i]$ et séparer en 2 tableaux ;
U : tableau des plus petits ;
V : tableau des plus grands ;
$U \leftarrow$ trirapiderandomisé(U) ;
$V \leftarrow$ trirapiderandomisé(V) ;
renvoyer concaténation de U, $[T[i]]$ et V.

De cette façon, on subit moins les effets de bord, et on peut bénéficier (sauf très grande malchance) d'un algorithme efficace dans toutes circonstances.

IV Limitations du modèle à coûts fixes

Si on est amené à manipuler des grands nombres, il est évident qu'on ne peut plus considérer le coût des opérations comme indépendant des entrées. On exprime alors le coût des opérations en fonction de $\lg(N)$, le nombre de chiffres de N en base 2 (ou choix d'une autre base à convenance). Les différentes opérations ont alors un coût qu'il convient de ne pas négliger, et qui diffère suivant les opérations. Par exemple, par l'algorithme standard, le coût de l'addition $m+n$ va être de l'ordre de $\max(\lg(m), \lg(n))$, alors que le

coût du produit mn va être de l'ordre de $\lg(m) \times \lg(n)$. Le coût du produit peut être amélioré en utilisant des algorithmes plus sophistiqués, comme l'algorithme de Karatsuba, ou encore un algorithme basé sur la transformée de Fourier rapide.

Dans certaines situations, il est possible d'éviter les grands nombres. C'est le cas par exemple si on travaille modulo K, où K est un entier fixé. Attention à la maladresse consistant dans cette situation à réduire modulo K à la fin du calcul. Comme exemple, reprenons l'algorithme de Hörner, adapté pour une évaluation modulo K. Que pensez-vous des deux algorithmes ci-dessous ? Lequel est meilleur ?

Algorithme 4.13 : Hörner 2

Entrée : T, coefficients d'un polynôme P, a : réel, K base du modulo
Sortie : $P(a)$
$S \leftarrow T[n]$;
pour $i \leqslant n-1$ **descendant à** 0 **faire**
  $S \leftarrow S * a + T[i]$
fin pour
renvoyer $(S \mod K)$

Algorithme 4.14 : Hörner 3

Entrée : T, coefficients d'un polynôme P, a : réel, K base du modulo
Sortie : $P(a)$
$S \leftarrow T[n] \mod K$;
pour $i \leftarrow n-1$ **descendant à** 0 **faire**
  $S \leftarrow ((S * a + T[i]) \mod K$
fin pour
renvoyer (S)

V Exercices

Les trois premiers exercices doivent être considérés comme du cours.

Exercice 9 (Maximum)

Étudier la correction, la terminaison et la complexité de l'algorithme de recherche du maximum décrit dans le chapitre précédent.

Exercice 10 (Recherche)

Étudier la correction, la terminaison et la complexité dans le pire et dans le meilleur des cas de l'algorithme de recherche d'un élément dans une liste.

Exercice 11 (Recherche dans une liste triée)

Étudier la correction, la terminaison et la complexité de l'algorithme de recherche dichotomique dans une liste triée.

On consigne les résultats des exercices précédents dans cette proposition, à connaître comme résultat du cours :

Proposition 4.5.1 (Complexité de quelques recherches)

(i) La recherche du maximum (ou du minimum) d'une liste se fait en temps linéaire

(ii) La recherche d'un élément dans une liste (première position, ou test d'appartenance) se fait en temps au plus linéaire.

(iii) La recherche d'un élément (ou de la position d'insertion d'un élément) dans une liste triée se fait en temps logarithmique.

Exercice 12 (Tri à bulle)

Étudier la correction et la terminaison de l'algorithme 35 : Proposer une amélioration repérant lors du passage dans la boucle interne, le plus grand indice pour lequel une inversion a été faite. Jusqu'où suffit-il d'aller pour la boucle suivante ? Étudier la correction et la terminaison du nouvel algorithme obtenu.

Algorithme 4.15 : Tri-bulle

Entrée : T :tableau de réels, n :taille du tableau
Sortie : T : tableau trié
pour $i \leftarrow 1$ **à** $n-1$ **faire**
  pour $j \leftarrow 1$ **à** $n-i$ **faire**
    si $T[j] < T[j-1]$ **alors**
      $T[j-1], T[j] \leftarrow T[j], T[j-1]$
    fin si
  fin pour
fin pour

Exercice 13 (Tri par sélection)

Donner le pseudo-code de l'algorithme de tri par sélection consistant à rechercher le minimum et à le mettre à sa place, puis à rechercher le deuxième élément et à le mettre à sa place etc. Étudier la terminaison et la correction de cet algorithme.

Exercice 14 (Décomposition)

Que fait l'algorithme 4.16 ? Justifier sa terminaison et sa correction.

Algorithme 4.16 : Décomposition

Entrée : n : entier positif
Sortie : L : liste de couples d'entier ; à quoi correspond-elle ?
$p \leftarrow 2$;
$L \leftarrow []$;
tant que $p \leqslant \sqrt{n}$ **faire**
  $i \leftarrow 0$;
  tant que $n \mod p = 0$ **faire**
    $i \leftarrow i + 1$;
    $n \leftarrow n/p$
  fin tant que
  si $i > 0$ **alors**
    Ajouter (p, i) à la liste L
  fin si
  si $p = 2$ **alors**
    $p \leftarrow p + 1$
  sinon
    $p \leftarrow p + 2$
  fin si
fin tant que
si $n > 1$ **alors**
  Ajouter $(n, 1)$ à la liste L
fin si