Complexité des algorithmes

Nous savons montrer qu'un algorithme termine et qu'il retourne un résultat correct. Mais à quoi sert un algorithme si son temps d'exécution est *trop* long ? Dans ce chapitre, nous allons apprendre à *estimer* le temps d'exécution d'un algorithme à partir de son code. Pour être tout à fait correct, on n'évalue pas le *temps d'exécution*, mais plutôt le *nombre d'opérations effectuées*. Une fois ce nombre connu, on peut donner une évaluation du temps *en fonction des performances de la machine*. Pour parler de temps, il faut en effet connaître l'ordinateur utilisé. Comme d'habitude, nous nous focalisons sur la partie algorithmique et nous cherchons un *ordre de grandeur* exprimé à l'aide des notations $O()$ et $\Theta()$ du nombre d'opérations effectuées en fonction de la taille des données.

I Exemple introductif

On considère l'algorithme suivant :

Entrées : un tableau d'entiers $L[1..n]$ avec $n \geq 1$ et un entier a
Sorties : un booléen, vrai ssi $a \in L$
i : *entier* $\leftarrow 1$
tant que $i \leq n$ *et* $L[i] \neq a$ **faire**
  $i \leftarrow i + 1$
fin
retourner $i \leq n$

Algorithme 1 : Recherche d'un élément dans une liste

1 – On pose $L = [5, 1, 7]$. Quel est le nombre d'opérations effectuées par l'algorithme de recherche avec les entrées L et 5 ? Avec L et 7 ? Avec L et 3 ?

Définition 1 : Types de complexité

Pour un algorithme d'entrée E (E peut éventuellement être un ensemble d'entrées), on appelle :
- *complexité dans le meilleur cas* le nombre minimal d'opérations effectuées par l'algorithme sur une entrée E de taille n ;
- *complexité dans le pire cas* le nombre maximal d'opérations effectuées par l'algorithme sur une entrée E de taille n ;
- *complexité en moyenne* le nombre moyen d'opérations effectuées par l'algorithme sur l'ensemble des entrées E de taille n.

2 – Dans l'algorithme précédent, quel est le nombre minimal d'opérations effectuées avec en entrée un tableau de taille n ? Dans quels cas est-il atteint ?

3 – Dans l'algorithme précédent, quel est le nombre maximal d'opérations effectuées avec en entrée un tableau de taille n ? Dans quels cas est-il atteint ?

4 – On suppose que l'on sait que l'élément se trouve dans le tableau. On suppose de plus que les positions sont équiprobables, i.e., la probabilité que a se trouve en première position est $1/n$, en deuxième position $1/n$, etc. Quel est le nombre moyen d'opérations effectuées par l'algorithme pour des tableaux de taille n ?

5 – On suppose maintenant que l'élément a une chance sur deux de se trouver dans le tableau et que s'il s'y trouve, les position sont équiprobables. Quel est le nombre moyen d'opérations effectuées pour des listes de taille n ?

La complexité dans le pire cas donne un *majorant* du nombre d'opérations effectuées. La complexité en moyenne donne une *évaluation du nombre moyen* d'opérations effectuées. La complexité dans le meilleur cas donne un *minorant* du nombre d'opérations effectuées. Dans ce chapitre, nous nous concentrons sur la complexité dans le pire des cas.

Définition 2 :

Le calcul de complexité et le calcul du nombre d'opérations sont la même tâche. On donne un résultat de complexité par une relation de comparaison, sous la forme d'un O ou d'un Θ fonction de la taille des données.

II Calcul de complexité

Nous allons ici mettre en place une technique de calcul de complexité. Comme dans le cas de la correction, nous décomposons le calcul de complexité sur des instructions atomiques : nous allons nous intéresser à la complexité d'une affectation, d'une séquence d'instructions, d'une instruction conditionnelle, d'une boucle.

Dans la suite, on note $T(P)$ la complexité dans le pire cas d'un bloc d'instructions P.

II.1 Complexité d'une instruction simple

On s'intéresse ici à la complexité d'une instruction simple : incrémentation, affectation, retour d'une valeur ...

Propriété 1:

Une instruction simple s'effectue en temps constant.

On a donc $T(X \leftarrow a) = 1$, que a soit une incrémentation ou une expression simple.

Un calcul de complexité s'appuie toujours sur une *modélisation* du processeur qui réalise effectivement le calcul. Si on s'intéresse à un ordre de grandeur du temps de calcul (et ce sera notre cas), on peut se permettre certaines simplifications. Certains contextes industriels peuvent imposer de descendre plus finement dans la description du calcul, par exemple au niveau assembleur, pour donner des informations plus précises sur les temps de calcul.

6 – Quelle est la complexité de l'algorithme suivant ?

```
x ← x + 1
```

II.2 Complexité d'une séquence d'instructions

On considère deux blocs d'instructions P et Q.

Propriété 2:

Si $T(P)$ est le nombre d'opérations du bloc P et si $T(Q)$ est le nombre d'opérations du bloc Q, alors $T(P;Q)$, le nombre d'opérations de la séquence $P;Q$, est :

$$T(P\,;\,Q) = T(P) + T(Q).$$

7 – Quel est le nombre d'opérations l'algorithme suivant ? Donner une évaluation de sa complexité.

```
x, y : entier;
x ← x + y;
y ← x − y;
x ← x − y;
```

II.3 Complexité d'une instruction conditionnelle

Dans les exemples de complexité d'instructions simples ou de séquences, nous n'avons pas eu besoin de faire de différence entre les complexités dans le meilleur ou pire cas, ou cas moyen. Ce n'est plus le cas lorsque l'on s'attaque à la complexité d'une instruction conditionnelle pour laquelle on se focalise sur l'étude de la complexité dans le pire cas. Le cas de l'instruction conditionnelle se traite comme suit : on évalue chaque bloc et on regroupe.

Propriété 3:

Si P est une instruction conditionnelle du type **si** b **alors** Q_1 **sinon** Q_2 **finsi**, si le nombre d'opérations dans le pire cas de Q_1 est $T(Q_1)$ et si celui de Q_2 est $T(Q_2)$, alors on a le résultat suivant :

$$T(P) = T(\textbf{si } b \textbf{ alors } Q_1 \textbf{ sinon } Q_2 \textbf{ finsi}) = 1 + \max(T(Q_1), T(Q_2)).$$

8 – Quel est le nombre d'opérations dans le pire cas de l'algorithme suivant ? Quelle est sa complexité ?

```
si Syr : entier est pair alors
|  Syr ← Syr/2
sinon
|  Syr ← 3 * Syr
|  Syr ← Syr + 1
fin
```

II.4 Complexité d'une boucle pour

Dans un premier temps, on s'intéresse aux boucles **pour**.

Propriété 4:

Si P est une instruction de la forme $P :=$ **pour** i **de** 1 **à** p **faire** $P(i)$ **fin**, alors le nombre d'opérations dans le pire cas de P s'exprime :

$$T(P) = T(\textbf{pour } i \textbf{ de } 1 \textbf{ à } p \textbf{ faire } P(i) \textbf{ fin}) = 1 + 2p + \sum_{i=1}^{p} T(P(i)).$$

9 – Quels sont le nombre d'opérations et la complexité dans le pire cas de l'algorithme suivant :

```
Entrées : i un entier
somme : entier ← 0
pour j : entier de 1 à i faire
|  somme ← somme + 1
fin
retourner somme
```

10 – Quels sont le nombre d'opérations et la complexité dans le pire cas de l'algorithme suivant :

```
Entrées : n un entier positif
somme : entier ← 0
pour i : entier de 1 à n faire
|  pour j : entier de 1 à n faire
|  |  somme ← somme + 1
|  fin
fin
retourner somme
```

11 – Quels sont le nombre d'opérations et la complexité dans le pire cas de l'algorithme suivant :

```
Entrées : n un entier positif
somme : entier ← 0
pour i : entier de 1 à n faire
    pour j : entier de 1 à i faire
        somme ← somme + 1
    fin
fin
retourner somme
```

II.5 Complexité d'une boucle tant que

Ce sont principalement les boucles **pour** et **Tant que** qui génèrent des temps de calcul importants. Pour les boucles **Tant que**, on suppose que l'on dispose d'une fonction de rang à valeur dans $\mathbb{N}$. On commence par déterminer la valeur n de la fonction de rang la *première fois* qu'on entre dans la boucle. Cette valeur n correspond par définition de la fonction de rang à un majorant du nombre de passages dans la boucle. On estime en suite le coût c d'un passage dans la boucle. Le nombre d'opérations exécutées par la boucle **Tant que** est alors majoré par le produit $n*(1+c)$.

Propriété 5:

Si P est une instruction de la forme **Tant que** C **faire** B **fin** et si le nombre d'opérations dans le pire des cas pour l'exécution de B est $T(B)$ alors le nombre d'opérations dans le pire cas de P s'exprime par :

$$T(P) = 1 + \texttt{valeur initiale de la fonction de rang} * (1 + T(B))$$

Par exemple, considérons l'algorithme suivant :

```
Entrées : n, un entier naturel
i : entier ← 0
Tant que i < n faire
    i ← i + 1
```

On vérifie que la fonction $f(n,i) = n - i$ est bien une fonction de rang à valeur dans $\mathbb{N}$. Le coût du corps de la boucle est unitaire. La première fois qu'on entre dans la boucle, i vaut 0. La valeur initiale de la fonction de rang est donc $f(n,0) = n$. Le nombre d'opérations dans le pire des cas est $1+n*(1+1) = 1+2n$. La complexité en temps de l'algorithme est $O(n)$. On peut argumenter qu'elle est même en $\Theta(n)$ car le corps de la boucle sera exécuté n fois.

12 – Reprendre les algorithmes des fiches précédentes et pour chacun d'entre eux, étudier la complexité dans le pire des cas (par exemple les exercices 15, 19, 23, 29, 30, 17 et 37 de la fiche *Terminaison des programmes*).

13 – Le crible d'Ératosthène consiste à écrire les nombres entiers de 2 à n. On prend ensuite le premier nombre non barré (au début il s'agit de 2) puis on barre tous les nombres multiples de celui-ci. On prend alors le nombre non barré suivant et on recommence. À la fin, seuls les nombres premiers entre 2 et n restent non barrés. Écrire un algorithme mettant en place le crible d'Ératosthène, prenant en argument un nombre entier n et renvoyant la liste des entiers premiers inférieurs à n. Donner le nombre d'opérations effectuées et évaluer sa complexité en fonction de n. On notera $H_n = \sum_{k=1}^{n} \frac{1}{k}$ et on se rappellera que $H_n \sim \ln(n)$. Si on mesure la taille de l'entrée non par l'entier n, mais par le nombre de bits nécessaire pour coder n en binaire, quelle est la complexité de l'algorithme ?

III Échelle de complexité

En fonction de leur complexité, les algorithmes se regroupent en plusieurs familles, dont voici les principales :

- algorithmes en **temps constant**, noté $\Theta(1)$: quelque soit l'argument en entrée, le nombre d'opérations est constant. C'est le cas de l'échange de deux variables par exemple ;
- algorithmes en **temps logarithmique**, noté $\Theta(log(n))$: le nombre d'opérations présente une dépendance logarithmique par rapport à la taille de l'argument. Par exemple, la recherche dichotomique d'un élément dans un tableau trié de taille n est en $O(log(n))$;
- algorithmes en **temps linéaire**, noté $\Theta(n)$: le nombre d'opérations est proportionnel à la taille de l'argument. C'est le cas de la recherche du minimum d'un tableau de taille n ;

— algorithmes en **temps quasi-linéaire**, noté $\Theta(n\log(n))$: le log qui apparaît est en base quelconque (≥ 2) puisque le résultat s'établit à une constante multiplicative près. On peut montrer que le tri fusion, *mergesort*, trie un tableau de taille n en $\Theta(n\ log(n))$;
— algorithmes en **temps quadratique**, noté $\Theta(n^2)$: le nombre d'opérations possède une dépendance quadratique par rapport à la taille de l'argument. C'est le cas des algorithmes de tri simples, comme le tri par recherche itérée du minimum d'un tableau de taille n ;
— algorithmes en **temps polynomial**, noté $\Theta(n^k)$ avec $k > 1$: le nombre d'opérations possède une dépendance polynomiale par rapport à la taille de l'argument. Par exemple l'algorithme élementaire de multiplication de deux matrices carrées de taille n requiert de l'ordre de n^3 opérations ;
— algorithmes en **temps exponentiel**, noté $\Theta(c^n)$ où c est une constante > 1 : la dépendance par rapport à la taille de l'argument est exponentielle, comme par exemple le calcul d'une table de vérité pour une formule booléenne de n variables qui s'effectue en $\Theta(2^n)$ étapes.

Les algorithmes en temps polynomial sont considérés **efficaces**. Les algorithmes en temps exponentiel sont très rapidement inutilisables. Le tableau suivant donne une équivalence en temps (approximative car cela dépend évidemment de la machine) en fonction de la taille n des données.

n	5	10	20	50	250	1 000	10 000	1 000 000
1	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns
$\log(n)$	10 ns	10 ns	10 ns	20 ns	30 ns	30 ns	40 ns	60 ns
n	50 ns	100 ns	200 ns	500 ns	2.5 μs	10 μs	100 μs	10 ms
$n\ log(n)$	50 ns	100 ns	200 ns	501 ns	2.5 μs	10 μs	100,5 μs	10 050 μs
n^2	250 ns	1 μs	4 μs	25 μs	625 μs	10 ms	1 s	2.8 heures
n^3	1.25 μs	10 μs	80 ms	1.25 ms	156 ms	10 s	2.7 heures	316 ans
2^n	320 ns	10 μs	10 ms	130 jours	10^{59} ans	...	...	...
$n!$	1.2 μs	36 ms	770 ans	10^{48} ans	...	...	...	...

TABLE 1 – Ordre de grandeur du temps nécessaire à l'exécution d'un algorithme de complexité donnée (source : Wikipedia)

On parle aussi de complexité en mémoire : il s'agit d'évaluer (toujours en ordre de grandeur), la place occupée en mémoire par l'exécution de l'algorithme.

IV D'autres exercices

14 – Déterminer la complexité de l'algorithme suivant :

Entrées : n et m deux entiers
i : *entier* $\leftarrow 1$
j : *entier* $\leftarrow 1$
Tant que $i \leq m \wedge j \leq n$ **faire**
    $i \leftarrow i+1$
    $j \leftarrow j+1$

15 – Déterminer la complexité de l'algorithme suivant :

Entrées : n et m deux entiers
i : *entier* $\leftarrow 1$
j : *entier* $\leftarrow 1$
Tant que $i \leq m \vee j \leq n$ **faire**
    $i \leftarrow i+1$
    $j \leftarrow j+1$

16 – Déterminer la complexité de l'algorithme suivant :

Entrées : n et m deux entiers
i : *entier* $\leftarrow 1$
j : *entier* $\leftarrow 1$
Tant que $j \leq n$ **faire**
    si $i \leq m$
     alors $i \leftarrow i+1$
     sinon $j \leftarrow j+1$

17 – Déterminer la complexité de l'algorithme suivant :

Entrées : n et m deux entiers
i : *entier* $\leftarrow 1$
j : *entier* $\leftarrow 1$
Tant que $j \leq n$ **faire**
    si $i \leq m$
     alors $i \leftarrow i+1$
     sinon $j \leftarrow j+1$; $i \leftarrow 1$

18 – Déterminer la complexité de l'algorithme suivant :

Entrées : n et m deux entiers
i : *entier* $\leftarrow 1$
j : *entier* $\leftarrow 1$
Tant que $m < i \Rightarrow j \leq n$ **faire**
    $i \leftarrow i+1$
    $j \leftarrow j+1$

19 – À quelles conditions sur m et n le programme suivant termine-t-il ? Dans ce cas, déterminer sa complexité.

Entrées : n et m deux entiers
i : *entier* $\leftarrow 1$
j : *entier* $\leftarrow 1$
Tant que $i \leq m \Rightarrow j \leq n$ **faire**
    $i \leftarrow i+1$
    $j \leftarrow j+1$