Complexité

Nous avons vérifié qu'un algorithme terminait bien, puis qu'un algorithme terminait correctement : on fait ce que l'on veut faire dans un temps fini. Ces deux points ne sont pas suffisants : à quoi sert un algorithme qui termine et qui est correct, si son temps d'exécution est trop long ? Nous allons, dans ce chapitre, apprendre à évaluer le temps d'exécution d'un algorithme. Pour être tout à fait correct, on n'évalue pas le *temps* d'exécution, mais plutôt le nombre d'opérations effectuées. Une fois le nombre d'opérations connu, on peut donner une évaluation du temps *en fonction des performances de la machine*. Pour parler de temps, il faut donc connaître l'ordinateur utilisé. Nous nous focalisons, comme d'habitude, sur la partie algorithmique, donc pas de machine en vue : on se contente donc du nombre d'opérations.

I Cas introductif

Dans ce cours, les notations $O()$ et $o()$ ont la même signification qu'en mathématiques. On considère l'algorithme suivant :

Entrées : une liste $L[1..n]$ et un élément a
Sorties : un booléen
$i \leftarrow 1$
tant que $i < n$ *et* $L[i] \neq a$ **faire**
  $i \leftarrow i + 1$
fin
si $i<n$ **alors**
  retourner *Vrai*
sinon
  retourner *Faux*
fin

Algorithme 1 : Recherche d'un élément dans une liste

On considère $L = [5, 1, 2, 4, 6, 7]$.

1 – Quel est le nombre d'opérations effectuées par l'algorithme de recherche avec les entrées L et 5 ?

2 – Quel est le nombre d'opérations effectuées par l'algorithme de recherche avec les entrées L et 7 ?

3 – Quel est le nombre d'opérations effectuées par l'algorithme de recherche avec les entrées 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 de taille n.
On appelle *complexité dans le pire cas* le nombre maximal d'opérations effectuées par l'algorithme sur une entrée de taille n.
On appelle *complexité en moyenne* le nombre moyen d'opérations effectuées par l'algorithme sur l'ensemble des entrées de taille n.

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

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

6 – On suppose que l'on sait que l'élément se trouve dans la liste. Quel est le nombre moyen d'opérations effectuées par l'algorithme pour des listes de taille n ? (on suppose les positions équiprobables)

7 – On suppose maintenant que l'élément a une chance sur deux de se trouver dans la liste. Quel est le nombre moyen d'opérations effectuées pour des listes de taille n ?

La complexité dans le pire cas donne une majoration du nombre d'opérations effectuées. La complexité en moyenne donne une évaluation du nombre moyen d'opérations. La complexité dans le meilleur cas donne un minorant... et n'apporte donc aucune information utile.

Définition 2 :

Le calcul de complexité et le calcul du nombre d'opérations sont la même tâche. En général, on donnera un résultat de complexité par une relation de comparaison, sous la forme d'un o ou d'un O.

II Calcul de complexité

Nous allons ici mettre en place la 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 classique...

Propriété 1

La complexité d'une instruction simple est un $O(1)$: on dit qu'une instruction simple s'effectue en temps constant.

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

On insiste sur le terme *simple* car une instruction d'une ligne peut prendre beaucoup de temps : par exemple la multiplication matricielle demande plus d'opérations que l'addition, et ceci est aussi vrai sur les entiers. Un résultat de complexité est toujours soumis à interprétation.

8 – 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 le nombre d'opérations de la séquence $P;Q$ est $T(P) + T(Q)$.

9 – Quel est le nombre d'opérations dans le pire cas de l'algorithme suivant ? Donner une évaluation de sa complexité.

```
x ← x + y;
y ← x − y;
x ← x − y;
```

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. Cela est différent lorsque l'on s'attaque dans la suite à la complexité d'une instruction conditionnelle ou d'une boucle, pour lesquelles on se focalise sur l'étude de la complexité dans le pire cas.

III Complexité d'une instruction conditionnelle

Le cas de l'instruction conditionnelle se traite comme pour la correction : 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 :

$$T(P) = \max(T(Q_1), T(Q_2)).$$

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

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

Lorsque les deux membres sont dans la même classe de complexité, l'ordre de gandeur de l'instruction conditionnelle n'est pas très intéressant. En termes de temps de calcul, dans l'exemple précédent, la seconde branche demande environ deux fois plus de temps.

IV Complexité d'une boucle

On ne s'intéresse ici qu'aux boucles Pour, plus simples à mettre en oeuvre. On rappelle qu'il y a équivalence entre les boucles **pour** et les boucles **Tant que**, la seule différence se faisant dans les commodités d'implémentation.

Propriété 4

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

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

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

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

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

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

Ce sont principalement les boucles qui font apparaître des temps de calculs démesurés. Nous avons donné ici les grandes lignes, mais chaque cas d'étude est différent. On essaye souvent d'évaluer le nombre d'opérations, puis on donne la complexité.

V Échelle de complexité

Les algorithmes se regroupent en plusieurs familles, dont nous donnons ici les principales (liste non-exhaustive).

- le temps constant, noté $O(1)$: quelque soit l'argument en entrée, le nombre d'opérations est constant. C'est le cas de l'échange de deux variables par exemple ;
- le temps sous-linéaire, noté $O(\log(n))$: le nombre d'opérations présente une dépendance logarithmique par rapport à la taille de l'argument. C'est le cas de la recherche dichotomique ;
- le temps linéaire, noté $O(n)$: le nombre d'opérations est proportionnel à la taille de l'argument. C'est le cas de la recherche du minimum ou du maximum dans un tableau ;
- le temps quasi-linéaire, noté $O(n\log(n))$: le log qui apparaît est en base quelconque puisque le résultat diffère d'une constante multiplicative près. C'est le cas du crible d'Ératosthène ;
- le temps quadratique, noté $O(n^2)$: 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 ;
- le temps polynomial, noté $O(n^k)$: le nombre d'opérations possède une dépendance polynomiale par rapport à la taille de l'argument. Par exemple la multiplication matricielle requiert de l'ordre de n^3 opérations ;
- le temps exponentiel, noté $0(c^n)$ où c est une constante : la dépendance par rapport à la taille de l'argument est exponentielle. Ces algorithmes sont très rapidement inutilisables.

Le tableau suivant donne une équivalence en temps, approximative car cela dépend évidemment de la machine.

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
$O(n)$	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 d'un type de complexité (source : Wikipedia)

13 – Les algorithmes en temps polynomial énoncés précédemment ont tous été déjà vu. Pour chacun d'entre eux, étudier sa complexité.

14 – Expliquer pourquoi la complexité de la recherche dichotomique est en $O(\log(n))$.

15 – 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. On notera $H_n = \sum_{k=1}^{n} \frac{1}{k}$ on on se rappellera que $H_n \sim \ln(n)$.

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.