Thème 3

Structures linéaires

A. Complexité d'un algorithme

Analyser un algorithme revient le plus souvent à évaluer les ressources nécessaires à son exécution (la quantité de mémoire requise) et le temps de calcul à prévoir. Bien évidemment, ces deux notions dépendent de nombreux paramètres matériels qui sortent du domaine de l'algorithmique : nous ne pouvons attribuer une valeur absolue ni à la quantité de mémoire requise ni au temps d'exécution d'un algorithme donné. En revanche, il est souvent possible d'évaluer l'ordre de grandeur de ces deux quantités de manière à identifier l'algorithme le plus efficace au sein d'un ensemble d'algorithmes résolvant le même problème.
Pour réaliser cette évaluation, il est nécessaire de préciser un modèle de la technologie employée ; en ce qui nous concerne, il s'agira d'une machine à processeur unique pour laquelle les instructions seront exécutées l'une après l'autre, sans opération simultanées. Il faudra aussi préciser les instructions élémentaires disponibles ainsi que leurs coûts. Ceci est particulièrement important lorsqu'on utilise un langage de programmation tel que Python pour illustrer ce cours car ce langage possède de nombreuses instructions de haut niveau qu'il serait irréaliste de considérer comme ayant un coût constant : par exemple, la fonction sort permet effectivement de trier un tableau en une instruction, mais il serait illusoire de croire que son temps d'exécution est indépendant de la taille du tableau. En outre, pour évaluer cette dépendance il n'y a guère d'autre solution que de se plonger dans le code source de Python ou sa documentation.

A - 1. Instructions élémentaires

Les instructions élémentaires (et qui seront considérées comme ayant un coût constant) sont présentes dans la plupart des langages de programmation :

- opérations arithmétiques (addition, soustraction, multiplication, division, modulo, partie entière, ...)
- comparaisons de données (relation d'égalité, d'infériorité, ...)
- transferts de données (lecture et écriture dans un emplacement mémoire)
- instructions de contrôle (branchement conditionnel et inconditionnel, appel à une fonction auxiliaire, ...)

mais là encore il est parfois nécessaire de préciser la portée de certaines de ces instructions. En arithmétique par exemple, il est impératif que les données représentant les nombres soient codées sur un nombre fixe de bits. C'est le cas en général des nombres flottants (la classe `float`) et des entiers relatifs (la classe `int`) représentés usuellement sur 64 bits, mais dans certains langages existe aussi un type entier long dans lequel les entiers ne sont pas limités en taille. C'est le cas en Python, où coexistaient jusqu'à la version 3.0 du langage une classe `int` et une classe `long`. Ces deux classes ont depuis fusionné, le passage du type `int` au type `long` étant désormais transparent pour l'utilisateur.
Dans le cas des nombres entiers, l'exponentiation peut aussi être source de discussion : s'agit-il d'une opération de coût constant ? En général on répond à cette question par la négative : le calcul de n^k nécessite un nombre d'opérations élémentaires (essentiellement des multiplications) qui dépend de k. Cependant, certains processeurs possèdent une instruction permettant de décaler de k bits vers la gauche la représentation binaire d'un entier, autrement dit de calculer 2^k en coût constant.
Les comparaisons entre nombres (du moment que ceux-ci sont codés sur un nombre fixe de bits) seront aussi considérées comme des opérations à coût constant, de même que la comparaison entre deux caractères. En revanche, la comparaison entre deux chaînes de caractères ne pourra être considérée comme une opération élémentaire, même s'il est possible de la réaliser en une seule instruction

Python. Il en sera de meme des opérations d'affectation : lire ou modifier le contenu d'un case d'un tableau est une opération élémentaire, mais ce n'est plus le cas s'il s'agit de recopier tout ou partie d'un tableau dans un autre, même si la technique du slicing en Python permet de réaliser très simplement ce type d'opération.

A - 2. Notations mathématiques

Une fois précisé la notion d'opération élémentaire, il convient de définir ce qu'on appelle la taille de l'entrée. Cette notion dépend du problème étudié : pour de nombreux problèmes, il peut s'agir du nombre d'éléments constituant les paramètres de l'algorithme (par exemple le nombre d'éléments du tableau dans le cas d'un algorithme de tri) ; dans le cas d'algorithmes de nature arithmétique (le calcul de n^k par exemple) il peut s'agir du nombre de bits nécessaire à la représentation des données. Enfin, il peut être approprié de décrire la taille de l'entrée à l'aide de deux entiers (le nombre de sommets et le nombre d'arêtes dans le cas d'un algorithme portant sur les graphes).
Une fois la taille n de l'entrée définie, il reste à évaluer en fonction de celle-ci le nombre $f(n)$ d'opérations élémentaires requises par l'algorithme. Mais même s'il est parfois possible d'en déterminer le nombre exact, on se contentera le plus souvent d'en donner l'ordre de grandeur à l'aide des notations de Landau.
La notation la plus fréquemment utilisée est le « grand $\mathcal{O}$ » :

$$f(n) = \mathcal{O}(\alpha_n) \iff \exists B > 0,\ f(n) \leqslant B\alpha_n .$$

Cette notation indique que dans le pire des cas, la croissance de $f(n)$ ne dépassera pas celle de la suite (α_n). L'usage de cette notation exprime l'objectif qu'on se donne le plus souvent : déterminer le temps d'exécution dans le cas le plus défavorable. On notera qu'un usage abusif est souvent fait de cette notation, en sous-entendant qu'il existe des configurations de l'entrée pour lesquelles $f(n)$ est effectivement proportionnel à (α_n).
D'un usage beaucoup moins fréquent, la notation Ω exprime une minoration du meilleur des cas :

$$f(n) = \Omega(\alpha_n) \iff \exists B > 0,\ f(n) \geqslant B\alpha_n .$$

L'expérience montre cependant que pour de nombreux algorithmes le cas « moyen » est beaucoup plus souvent proche du cas le plus défavorable que du cas le plus favorable. En outre, on souhaite en général avoir la certitude de voir s'exécuter un algorithme en un temps raisonnable, ce que ne peut exprimer cette notation.
Enfin, lorsque le pire et le meilleur des cas ont même ordre de grandeur, on utilise la notation Θ :

$$f(n) = \Theta(\alpha_n) \iff f(n) = \mathcal{O}(\alpha_n) \text{ et } f(n) = \Omega(\alpha_n).$$

Cette notation exprime le fait que quelle que soit le configuration de l'entrée, le temps d'exécution de l'algorithme sera *grosso-modo* proportionnel à α_n.

Ordre de grandeur et temps d'exécution

Nous l'avons dit, la détermination de la complexité algorithmique ne permet pas d'en déduire le temps d'exécution mais seulement de comparer entre eux deux algorithmes résolvant le même problème. Cependant, il importe de prendre conscience des différences d'échelle considérables qui existent entre les ordres de grandeurs usuels que l'on rencontre. En s'appuyant sur une base de 10^9 opérations par seconde, le tableau de la figure 3.1 est à cet égard significatif.
La lecture de ce tableau est édifiante : il faut autant que faire se peut éviter toute complexité temporelle supérieure à un coût quadratique.

B. Structures de données linéaires

Dans son acceptation la plus générale, une structure de données spécifie la façon de représenter en mémoire machine les données d'un problème à résoudre en décrivant :

- la manière d'attribuer une certaine quantité de mémoire à cette structure ;
- la façon d'accéder aux données qu'elle contient.

	$\log n$	n	$n \log n$	n^2	n^3	2^n
10^2	7 ns	100 ns	0.7 µs	10 µs	1 ms	$4 \cdot 10^{13}$ années
10^3	10 ns	1 µs	10 µs	1 ms	1 s	10^{292} années
10^4	13 ns	10 µs	133 µs	100 ms	17 min	
10^5	17 ns	100 µs	2 ms	10 s	11,6 jours	
10^6	20 ns	1 ms	20 ms	17 min	32 années	

FIGURE 3.1. – Temps nécessaire à l'exécution d'un algorithme en fonction de sa complexité

$\mathcal{O}(\log n)$	logarithmique
$\mathcal{O}(n)$	linéaire
$\mathcal{O}(n \log n)$	semi-linéaire
$\mathcal{O}(n^2)$	quadratique
$\mathcal{O}(n^k) \quad (k \geqslant 2)$	polynomiale
$\mathcal{O}(k^n) \quad (k > 1)$	exponentielle

FIGURE 3.2. – Qualifications usuelles des complexités

Dans certains cas, la quantité de mémoire allouée à la structure de donnée est fixée au moment de la création de celle-ci et ne peut plus être modifiée ensuite ; on parle alors de structure de données *statique*. Dans d'autres cas l'attribution de la mémoire nécessaire est effectuée pendant le déroulement de l'algorithme et peut donc varier au cours de celui-ci ; il s'agit alors de structure de données *dynamique*. Enfin, lorsque le contenu d'une structure de donnée est modifiable, on parle de structure de donnée *mutable*.
Par exemple, en PYTHON la classe `tuple` et la classe `str` sont des structures de données statiques et non mutables, contrairement à la classe `list` qui est une structure de donnée dynamique et mutable.

```
>>> l = [1, 2, 3]
>>> l.append(4)
>>> l[0] = 5
>>> l
[5, 2, 3, 4]
```

```
>>> t = (1, 2, 3)
>>> t.append(5)
AttributeError: 'tuple' object has no attribute 'append'
>>> t[0] = 5
TypeError: 'tuple' object does not support item assignment
```

FIGURE 3.3. – la classe `list` est dynamique et mutable, pas la classe `tuple`

Les structures de données classiques appartiennent le plus souvent aux familles suivantes :

- les *structures linéaires* : il s'agit essentiellement des structures représentables par des suites finies ordonnées ; on y trouve les listes, les tableaux, les piles, les files ;
- les *matrices* ou *tableaux multidimensionnels* ;
- les *structures arborescentes* (en particulier les arbres binaires) ;
- les *structures relationnelles* (bases de données ou graphes pour les relations binaires).

B - 1. Tableaux

Les tableaux forment une suite de variables de même type associées à des emplacements consécutifs de la mémoire.
Puisque tous les emplacements sont de même type, ils occupent tous le même nombre d de cases mémoire ; connaissant l'adresse a de la première case du tableau, on accède en coût constant à l'adresse de la case d'indice k en calculant $a + kd$. En revanche, ce type de structure est statique : une fois un

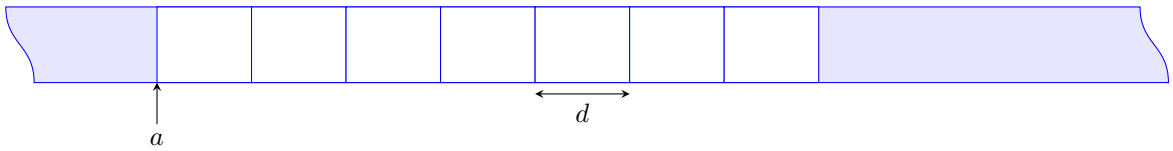

Figure 3.4. – Représentation d'un tableau en mémoire

tableau créé, la taille de ce dernier ne peut plus être modifiée faute de pouvoir garantir qu'il y a encore un espace mémoire disponible au delà de la dernière case. En résumé :

- un tableau est une structure de données statique ;
- les éléments du tableau sont accessibles en lecture et en écriture en temps constant $\mathcal{O}(1)$.

Les tableaux existent en Python : c'est la classe `array` fournie par la bibliothèque `numpy`.

B - 2. Listes chaînées

Les *listes* associent à chaque donnée (de meme type) un pointeur indiquant la localisation dans la mémoire de la donnée suivante (à l'exception de la dernière, qui pointe vers une valeur particulière indiquant la fin de la liste).

Figure 3.5. – Représentation d'une liste chaînée en mémoire

Dans une liste, il est impossible de connaître à l'avance l'adresse d'une case en particulier, à l'exception de la première. Pour accéder à la n-ième case il faut donc parcourir les $n-1$ précédentes : le coût de l'accès à une case est linéaire. En contrepartie, ce type de structure est dynamique : une fois la liste créée, il est toujours possible de modifier un pointeur pour insérer une case supplémentaire. En résumé :

- une liste est une structure de données dynamique ;
- le n-ième élément d'une liste est accessible en temps $\mathcal{O}(n)$.

On notera que le type de liste que l'on vient de présenter est le plus courant (il s'agit de listes chaînées) mais il en existe d'autres : listes doublement chaînées permettant l'accès non seulement à la donnée suivante mais aussi à la donnée précédente, listes circulaires dans lesquelles la dernière case pointe vers la première, etc.
Contrairement à ce que pourrait laisser croire son nom, la classe `list` en Python n'est pas une liste au sens qu'on vient de lui donner, mais une structure de données plus complexe qui cherche à concilier les avantages des tableaux et des listes, à savoir être une structure de données dynamique dans laquelle les éléments sont accessibles à coût constant.

Implémentation d'une liste chaînée