Informatique PCSI
Analyse d'un programme

1 Complexité d'un algorithme

Lorsqu'un algorithme est correct, il doit encore, avant d'être programmé et exécuté, satisfaire à deux impératifs en terme de consommation de ressources :

- utiliser un espace en mémoire acceptable, on parle de complexité en espace ;
- produire la réponse attendue en un temps acceptable, on parle de complexité temporelle.

Nous parlons de la complexité d'un algorithme ou de son coût.

La complexité (ou le coût) en espace correspond aux tailles des variables utilisées. Elle intervient par exemple dans la manipulation d'images où on recherche pour les algorithmes une complexité constante, c'est-à-dire une écriture d'éléments en mémoire qui utilise un espace constant et réduit autant que possible. Diverses parties d'une machine sont utilisées pour la mémoire et plus les données à conserver ont une taille importante, plus elles sont stockées sur des parties éloignées du processeur avec un temps d'écriture et de lecture qui s'allonge.

Il faut être vigilant avec des algorithmes de tri pour lesquels de nouvelles listes sont construites à chaque appel récursif. Chaque fois que de nouvelles listes sont créées ou que des permutations d'éléments utilisent de l'espace mémoire, on perd en temps d'exécution une partie du bénéfice obtenu par l'algorithme sur le nombre d'opérations à effectuer.

Étudier la complexité temporelle consiste à évaluer le temps d'exécution d'un algorithme en fonction de la taille des données en entrée.

Pour conduire ces études, on évalue un ordre de grandeur du nombre d'opérations élémentaires contenues dans l'algorithme (affectations, opérations arithmétiques, opérations logiques, comparaisons) en supposant qu'elles ont le même temps d'exécution constant. C'est largement suffisant en première analyse. Pour un même problème, ce nombre d'opérations peut être différent suivant les cas. Par exemple trier une liste déjà triée est un cas à priori favorable. Il est important d'étudier le cas le moins favorable, le pire qui puisse se produire, celui qui coûte le plus cher.

Si une machine effectue 10^8 opérations en une seconde, la question n'est pas de savoir si un algorithme effectue 10^8+1 ou 10^8-1 opérations. La question est plutôt d'évaluer un ordre de grandeur du taux d'accroissement du nombre d'opérations en fonction de la taille n des données par exemple pour $n > 10^8$. Si n est multiplié par 2, est-ce que le nombre d'opérations est environ 2 fois plus grand, 4 fois plus grand, 8 fois plus grand ou pratiquement identique ?

Avec 10^8 opérations par seconde, si un programme comporte environ 10^{11} opérations et qu'il doit être exécuté mille fois, il faudra attendre la réponse pendant plus de dix jours. Il est donc important de pouvoir choisir l'algorithme le plus « performant ».

1.1 Temps d'exécution

Le temps d'exécution d'un programme dépend de la machine, du langage de programmation, de l'algorithme. La part de l'algorithme est obtenue par une évaluation de sa complexité temporelle.

Nous supposons que les temps d'exécution d'une affectation, d'une opération arithmétique simple, d'une comparaison sont pratiquement identiques et constituent une unité de base.

Nous posons les règles suivantes :

- le temps d'exécution d'une suite d'instructions est la somme des temps d'exécution de chaque instruction ;
- le temps d'exécution d'une instruction conditionnelle

```
si test :
```

```
     instructions1
sinon :
     instructions2
```

est inférieur ou égal au maximum des temps d'exécution de `instructions1` et `instructions2`, plus le temps d'exécution du test ;

- le temps d'exécution d'une boucle

```
pour i variant de 1 à p :
     instructions
```

est p fois le temps d'exécution de `instructions`, si ce temps est constant, plus p (le nombre d'affectations pour la variable i). Si le temps d'exécution de `instructions` dépend de la valeur de p, l'étude se mène au cas par cas.

Pour une boucle `tant que`, l'étude se mène aussi au cas par cas.

L'évaluation du temps d'exécution d'un algorithme se réduit ainsi à une évaluation en fonction d'un nombre n, (entier représentant la taille des données en entrée), du nombre total d'opérations élémentaires noté u_n. Le niveau de complexité correspond au type de croissance de la suite (u_n).

Suivant les valeurs de l'entrée, u_n peut pendre des valeurs très différentes. Si, par exemple, nous parcourons une liste à l'aide d'une boucle, à la recherche d'un élément, celui-ci peut se trouver en premier et nous sortons de la boucle, c'est le cas le plus favorable. Il peut se trouver à la fin de la liste, c'est le pire des cas.

1.2 Niveaux de complexité

Notation

Considérons deux suites (u_n) et (v_n).

Si pour tout n assez grand, u_n est majorée par v_n à une constante près, on écrit $u_n \in \mathcal{O}(v_n)$. On dit que u_n est de l'ordre de grandeur de v_n.

Mathématiquement, on écrit : $u_n \in \mathcal{O}(v_n)$ si $\exists N \in \mathbb{N}, \exists k \in \mathbb{R}, \forall n > N, u_n \leq kv_n$.

En pratique, on utilise souvent la propriété suivante : si $\lim\limits_{n\to+\infty} \dfrac{u_n}{v_n} = \ell$ alors $u_n \in \mathcal{O}(v_n)$.

Autrement dit, on vérifie si $\dfrac{u_n}{v_n}$ converge vers une constante.

- **Complexité constante** : $u_n \in \mathcal{O}(1)$. Le temps d'exécution est borné (indépendant de n). C'est le cas, par exemple, pour obtenir le premier élément d'une liste.

- **Complexité logarithmique** : $u_n \in \mathcal{O}(\log_2 n)$. On double le temps d'exécution, en élevant au carré la taille des données. C'est le cas avec la recherche dichotomique dans une liste triée.

Les algorithmes de complexité logarithmique sont très performants. Les ordres de grandeur $\log_2 n$, $\log_{10} n$ et $\ln n$ sont comparables.

- **Complexité linéaire** : $u_n \in \mathcal{O}(n)$. Cet ordre de grandeur peut s'obtenir avec une boucle non conditionnelle. Par exemple, le calcul de la somme ou de la moyenne de n termes, la recherche séquentielle dans une liste non triée de longueur n, ont une complexité en $\mathcal{O}(n)$.

Soit n la taille d'une donnée. On obtient une complexité linéaire en n si le nombre d'opérations à effectuer peut s'écrire sous la forme $\alpha n + \beta$, avec α et β réels, $\alpha > 0$.

- **Complexité log-linéaire ou linéarithmique** : $u_n \in \mathcal{O}(n \log_2 n)$. C'est la complexité de certains algorithmes de tri et pour les tris par comparaisons, il est impossible de faire mieux.

- **Complexité quadratique** : $u_n \in \mathcal{O}(n^2)$. C'est la complexité d'algorithmes construits avec deux boucles imbriquées comme certains algorithmes de tri.

Soit n la taille d'une donnée. On obtient une complexité quadratique en fonction de n si le nombre d'opérations à effectuer peut s'écrire $\alpha n^2 + \beta n + \gamma$, avec α, β et γ réels, $\alpha > 0$.

- **Complexité polynomiale** : $u_n \in \mathcal{O}(n^k)$, $k \geq 2$ fixé. Un algorithme d'une telle complexité n'est utilisable que dans quelques cas particuliers. La complexité est cubique si $k = 3$.
- **Complexité exponentielle** : $u_n \in \mathcal{O}(2^n)$ ou $u_n \in \mathcal{O}(k^n), k > 1$. C'est la complexité de certains algorithmes de résolution de problèmes pour lesquels personne n'a encore trouvé d'algorithmes plus performants et qui sont en pratique inutilisables.

1.3 Cas typiques

Avec deux boucles `for` imbriquées, nous avons trois cas typiques. Dans un cas, la complexité est linéaire. Dans les deux autres cas elle est quadratique.

Dans les codes qui suivent, les pointillés sous-entendent un nombre fixe d'opérations.

Premier cas : `n` est la taille de la donnée, `k` est un nombre fixé.

```
for i in range(n):
    ...
    for j in range(k):
        ...
```

Il y a `n` passages dans la boucle externe. À chaque passage dans cette boucle, il y a un nombre fixe q d'opérations puis `k` passages dans la boucle interne. Dans la boucle interne, nous avons un nombre fixe d'opérations r. Donc pour chaque valeur de `i`, nous avons $q + k \times r = \alpha$ opérations. Le nombre total d'opérations est donc αn et la complexité est linéaire.

Deuxième cas : `n` est la taille de la donnée.

```
for i in range(n):
    ...
    for j in range(n):
        ...
```

Il y a `n` passages dans la boucle externe. À chaque passage dans cette boucle, il y a un nombre fixe q d'opérations puis `n` passages dans la boucle interne. Dans la boucle interne, nous avons un nombre fixe d'opérations r. Donc pour chaque valeur de `i`, nous avons $q + n \times r$ opérations et le nombre total d'opérations est alors $n(q + n \times r) = rn^2 + qn$. La complexité est quadratique.

Troisième cas : `n` est la taille de la donnée.

```
for i in range(n):
    ...
    for j in range(i):
        ...
```

Il y a `n` passages dans la boucle externe. À chaque passage dans cette boucle, pour chaque valeur de `i`, il y a un nombre fixe d'opérations q puis `i` passages dans la boucle interne. Dans la boucle interne, nous avons un nombre fixe d'opérations r. Donc pour chaque valeur de `i`, nous avons $q + i \times r$ opérations. Les valeurs de `i` sont successivement 0, 1, 2, ..., n -1. Le nombre total d'opérations est donc $q + (q + 1 \times$

$r) + (q + 2 \times r) + \ldots + (q + (n-1) \times r)$, soit $nq + r(1 + 2 + \ldots + (n-1))$. Le calcul de la somme des entiers est connu. Le résultat est $nq + r\dfrac{n(n-1)}{2}$.

Finalement le nombre d'opérations est $\dfrac{r}{2}n^2 + \left(q - \dfrac{r}{2}\right)n$, qui est de la forme $\alpha n^2 + \beta n$ et la complexité est quadratique.

Dans les chapitres précédents, divers niveaux de complexité sont présents. Au chapitre 1, la recherche séquentielle d'un élément dans une liste a une complexité linéaire. Dans le cas d'une liste triée, cette recherche à l'aide d'un algorithme dichotomique au chapitre 4 a une complexité logarithmique. Au chapitre 2, le tri à bulles a une complexité quadratique. C'est aussi le cas avec les algorithmes de tri sélection et insertion au chapitre 8. Dans ce même chapitre, une amélioration est apportée par le tri fusion qui a une complexité log-linéaire.

De manière générale, la dichotomie permet de passer d'un ordre n à un ordre $\log_2 n$.

1.4 Propriétés

- Des algorithmes permettant de résoudre un même problème, peuvent être rangés suivant leur ordre de complexité du plus efficace au moins efficace. Le classement des ordres de complexité doit être connu : $\mathcal{O}(1)$, $\mathcal{O}(\log_2 n)$, $\mathcal{O}(n)$, $\mathcal{O}(n \log_2 n)$, $\mathcal{O}(n^2)$, $\mathcal{O}(n^2 \log_2 n)$, $\mathcal{O}(n^3)$, etc.
- Si deux blocs d'instructions successifs ont une complexité en $\mathcal{O}(u_n)$ alors la complexité totale est en $\mathcal{O}(u_n)$.
- Si on répète u_n fois un bloc d'instructions de complexité en $\mathcal{O}(v_n)$ alors la complexité totale est en $\mathcal{O}(u_n v_n)$. Si u_n a une valeur constante, la complexité totale est en $\mathcal{O}(v_n)$.
- Si deux blocs d'instructions successifs ont une complexité respectivement en $\mathcal{O}(u_n)$ pour le premier et en $\mathcal{O}(v_n)$ pour le second alors la complexité totale est en $\mathcal{O}(max(u_n, v_n))$.