Cours 2: Complexité

1 Introduction

Pour résoudre un problème donné, il peut exister plusieurs méthodes, le choix d'un algorithme se base sur plusieurs critères qui peuvent être:

- le temps d'exécution ,
- la place mémoire utilisée,
- la qualité du code source,
- la facilité de compréhension,
- etc...

Les deux premiers critères sont appelés la complexité en temps et en espace. Nous allons nous intéresser dans le cadre de ce cours à la complexité en temps.

2 Complexité en temps

La complexité en temps revient à calculer le nombre d'opérations fondamentales qui peuvent être:

- des comparaisons,
- des accès mémoire,
- des opérations arithmétiques
- des opérations logiques,
- etc …

Pour réaliser une analyse exacte, il est nécessaire de compter le nombre d'opérations de chaque type puis de leur affecter un poids proportionnel au temps d'exécution. Pour simplifier l'analyse, et puisque ces temps peuvent varier d'une machine à l'autre, on considère généralement un (ou deux) type(s) d'opération(s) comme fondamentale(s), en négligeant les autres. Les opérations choisies dépendent du problème traité. Par exemple :

- recherche d'un élément dans une liste : nombre de comparaisons,
- tri d'un tableau : nombre de comparaisons et de déplacements,
- produit de matrices : nombre de multiplications et d'additions.

La complexité dépend également du nombre de données traitées, par exemple une liste de 3 éléments sera naturellement plus rapide à trier qu'une liste de 10000 éléments. La complexité d'un algorithme est exprimée en fonction de la taille des données à traiter:

$$T(n)$$

Le temps d'exécution peut également dépendre des données, on considère en général la complexité dans le pire cas $T_{max}(n)$ et la complexité moyenne $T_{moy}(n)$ (celle-ci nécessite des suppositions sur la distributions des données).

$$T_{moy}(n) = \sum_{d \in D_n} p(d).T(f(d))$$

avec :

p(d) la probabilité d'avoir la donnée *d* en entrée de l'algorithme.

3 Calcul de la complexité

Il n'existe pas de système strict pour le calcul de la complexité, mais voici quelques règles communément admises:

- Le temps d'une séquence d'instructions est la somme des temps d'exécution.

$$T(P;Q)=T(P)+T(Q)$$

- Pour le cas des branchements conditionnels, on considère le pire cas.

$$T(if\ (t)\ P\ else\ Q)=max(T(P),T(Q))$$

- Pour les boucles, on somme les temps d'exécution du corps de la boucle pour chaque itération.

$$T(for\ (i=0;i<n;i=i+1)\ P(i))=\sum_{i=0}^{n-1} T(P(i))$$

- Les fonctions récursives donnent généralement lieu à la résolution de relations de récurrence du type :

$$T(n)=f(T(n-1))$$

4 Ordre de grandeur

Le calcul exact de la complexité est souvent long et fastidieux. Comme le but est généralement de comparer les algorithmes entre eux, on utilise la notion d'ordre de grandeur:

$O(n)$ se prononce « grand O de n » ou « O de n »

L'ordre de grandeur représente le comportement asymptotique du temps d'exécution d'un algorithme, c'est à dire comment évolue le temps d'exécution lorsque l'on augmente la taille des données. Voici les principales complexités :

$O(1)$	Temps constant
$O(\log_2(n))$	Logarithmique
$O(n)$	Linéaire
$O(n.log_2(n))$	Linéarithmique
$O(n^k)$	Polynomiale
$O(2^n)$	Exponentielle

Il est à noter que les formes polynomiale (avec $k>3$) et exponentielle ne sont en pratique utilisables que sur des données de petite taille.

$f=O(g)$ si et seulement si $\exists c\in\mathbb{R}^{+*}$, $\exists n_0\in\mathbb{N}$ tel que

$$\forall\ n>n_0\ f(n)\leqslant c.g(n)$$

On dit que f est dominée asymptotiquement par g. Cette définition donne un majorant, mais on cherche généralement le plus petit majorant lorsque l'on calcul l'ordre de grandeur.

Voici l'évolution des principales complexités en fonction du nombre de données :