Cours_complexite_d_un_algorithme

April 26, 2021

1 Notion de complexité ou de coût d'un algorithme

Supposons qu'une machine doit effectuer une tâche, c'est-à-dire exécuter un algorithme :

- Voulez-vous que la machine prenne beaucoup ou peu de temps pour effectuer la tâche ?
- Voulez-vous que la machine nécessite beaucoup de mémoire pour effectuer la tâche ?

Bien sûr, on préfère qu'un algorithme prenne peu de temps et nécessite peu de mémoire pour être exécuté par la machine !

Pour un algorithme, on parle de **complexité** ou de **coût** et on distingue :

- la **complexité spatiale** : permet de quantifier l'utilisation des ressources mémoire par l'algorithme
- la **complexité temporelle** : permet de quantifier le temps d'exécution de l'algorithme

… on n'abordera ici que la complexité temporelle !

2 Estimation du coût d'un algorithme

Estimer la complexité algorithmique temporelle permet de comparer l'efficacité d'algorithmes résolvant le même problème. Dans une situation donnée, cela permet donc d'établir lequel des algorithmes disponibles s'exécutera le plus rapidement.

Pour estimer la complexité en temps, on va **compter le nombre d'opérations élémentaires** (affectation, calcul arithmétique ou logique, comparaison…) effectuées par l'algorithme. Par soucis de simplicité, on fera l'hypothèse que **toutes les opérations élémentaires sont à égalité de coût**, soit 1 "unité de temps".

Exemple : pour l'instruction `a = b * 3`, on a 2 opérations élémentaires (1 multiplication + 1 affectation). Cettes instruction consommera donc "2 unités de temps"

La complexité en temps d'un algorithme dépend :

- de la **taille des données** passées en paramètres (par exemple la longueur des listes): plus ces données seront volumineuses, plus il faudra d'opérations élémentaires pour les traiter. On notera `n` la taille des données à traiter (dans le cas de liste, `n` correspond à la longueur de la liste, c'est-à-dire le nombre d'éléments qu'elle contient).
- de la façon dont sont réparties les différentes éléments qui constituent la liste. Par exemple, si on effectue une recherche séquentielle d'un élément dans une liste non triée, on parcourt un par un les éléments jusqu'à trouver, ou pas, celui recherché. Ce parcours peut s'arrêter dès

le début si le premier élément est « le bon ». Mais on peut également être amené à parcourir la liste en entier si l'élément cherché est en dernière position, ou même n'y figure pas.

Cette remarque nous conduit à préciser un peu notre définition de la complexité en temps. En toute rigueur, on peut en effet distinguer deux formes de complexité en temps :

- la complexité dans le **meilleur des cas** : c'est la situation la plus favorable. Par exemple : recherche d'un élément situé à la première position d'une liste
- la complexité dans le **pire des cas** : c'est la situation la plus défavorable. Par exemple : recherche d'un élément dans une liste alors qu'il n'y figure pas

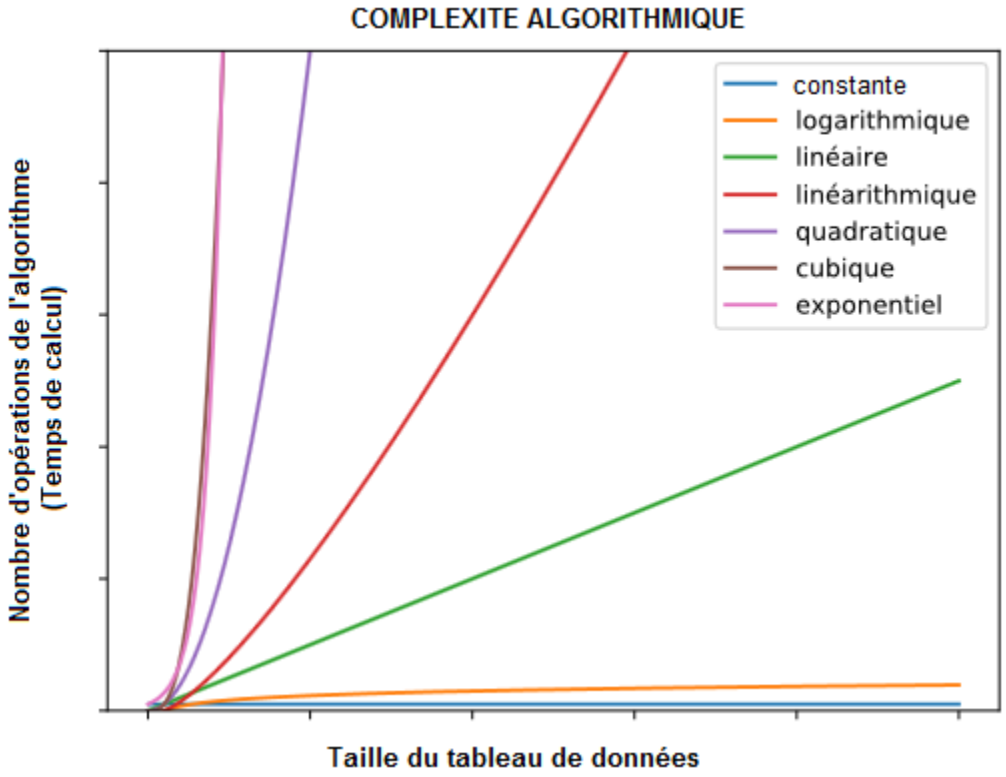
On calculera le plus souvent la complexité dans le pire des cas, car elle est la plus pertinente. Il vaut mieux en effet toujours envisager le pire.

3 Classe de complexité

Pour comparer l'efficacité des algorithmes, il suffit de comparer leur classe de complexité. Ainsi un algorithme en $0(n)$ (de complexité linéaire) est meilleur qu'un algorithme en $0(n^2)$ (de complexité quadratique) car pour des très grands tableaux de données ($n \to \infty$) il sera plus rapide

complexité algorithmique	type de complexité
$0(1)$	constante
$0(log(n))$	logarithmique
$0(n \times log(n))$	linéarithmique
$0(n)$	linéaire
$0(n^2)$	quadratique
$0(n^3)$	cubique
$0(2^n)$	exponentielle
$0(n!)$	factorielle

La recherche en informatique consiste à mettre au point des algorithmes “dans le haut” du tableau précédent


COMPLEXITE ALGORITHMIQUE
Nombre d'opérations de l'algorithme
(Temps de calcul)
Taille du tableau de données
constante
logarithmique
linéaire
linéarithmique
quadratique
cubique
exponentiel