Algorithmique et complexité

Polytech Paris-Saclay, PEIP 2, Informatique 3

Thibaut Benjamin

7 Novembre 2025

Amphi 2

Séance du jour

- Les outils pour mesurer la complexité
- Quantifier la complexité des algorithmes de tri

Introduction à la mesure de complexité

Ce que l'on cherche à faire

- Etant donné un algorithme, on veut savoir s'il va s'exécuter rapidement ou pas.
- On veut en particulier regarder comment la vitesse évolue sur des données de très grande taille.
- On cheche un ordre de grandeur, pas un calcul exact.

La complexité : un moyen détourné

- La complexité est un proxy pour la vitesse d'exécution
  Intuition : Plus un algorithme va effectuer d'opérations élementaires, et plus il va prendre du temps.

- C'est un modèle simplifié !
  En pratique toutes les « opérations élémentaires » ne prennent pas le même temps, l'OS peut interrompre un programme pour faire autre chose en attendant ...

La complexité asymptotique en action

- On reprend les algorithmes de tri
  `https://mszula.github.io/visual-sorting`
- Fixer le delay à 1.6 ms, et on lance les algorithmes suivants avec 100, 200 puis 300 éléments.
  - Tri par sélection
  - Tri fusion

Comment quantifier la différence de comportement entre ces deux algorithmes ?

- Définition de la le complexité comme une fonction mathématique : $T(n) =$ nombre d'opération que l'algorithme fait en donnant des entrées de taille n.

- **Idée importante** : comparer le nombre d'opérations avec des fonction mathématiques connues.

Les fonctions mathématiques et leur croissance

Objectif

Cette section a pour but de rappeler le comportement des fonctions mathématiques usuelles.

- Les fonctions linéaires
- La fonction carré
- La fonction exponentielle en base 2
- La fonction logarithme en base 2

Les fonctions linéaires

Definition
Une fonction linéaire est une fonction donnée par la formule $f(x) = ax$.

- Elles préservent les rapports $\frac{f(x)}{f(y)} = \frac{x}{y}$
- Si un programme a une complexité linéaire, en doublant la taille de ses entrées, on double le temps d'exécution.

La fonction carré

Definition
La fonction carré est donnée par la formule $f(x) = x^2$.

- Si un programme a comme complexité la fonction carré, à chaque fois que ses entrées doublent de taille, le temps d'exécution est multiplié par 4.

La fonction exponentielle en base 2

Definition
La exponentielle en base 2 est donnée par la formule $f(x) = 2^x$.

- Si un programme a comme complexité la fonction carré, à chaque fois que la taille des entrées augmente de 1, le temps d'exécution double.

La fonction logarithme en base 2

Definition
La fonction log (logarithme en base 2) est la solution de l'équation $2^{\log(x)} = x$.

- Si un programme a comme complexité logarithmique, à chaque fois que la taille des entrées double, le temps d'exécution augmente de 1.

Le comportement asymptotique des fonctions

Definition
Etant donné deux fonctions f, g, on écrit $f(x) = \mathcal{O}(g(x))$ (prononcer « grand O »), si il existe une constante $M > 0$, et un rang x_0 tels que pour tout $x \geq x_0$, $f(x) \leq M \times g(x)$.

C'est la bonne notion « f grandit au plus aussi vite que g, en ordre de grandeur ».

Attention Il est tout à fait possible que $f(x) = \mathcal{O}(g(x))$ et que pourtant $f(x) > g(x)$. Mais la fonction f ne peut pas grossir significativement plus vite que g.

La notation Ω

Definition
Etant donné deux fonctions f, g, on écrit $f(x) = \Omega(g(x))$ (prononcer « grand Omega »), si il existe une constante $m > 0$, et un rang x_0 tels que pour tout $x \geq x_0$, $f(x) \geq M \times g(x)$.

C'est la bonne notion « f grandit au moins aussi vite que g, en ordre de grandeur ».

C'est moins utile en pratique : C'est plus intéressant de savoir qu'une fonction ne va pas prendre plus qu'un temps donné, plutôt que de savoir qu'elle va prendre au moins un temps donné.

La notation Θ

Definition
Etant donné deux fonctions f, g, on écrit $f(x) = \Theta(g(x))$ (prononcer « grand Theta »), si on a à la fois $f(x) = \mathcal{O}(g(x))$ et $f(x) = \Omega(g(x))$.

C'est la bonne notion « f et g grandissent de manière à peu près similaire ».

Attention : il est tout à fait possible que $f(x) = \Theta(g(x))$ et que les valeurs de $f(x)$ et de $g(x)$ soient très différentes. La seule contrainte c'est qu'elles grandissent à peu près au même rythme.

Les ordres de grandeurs de complexité

- On classifie les programmes selon leur classe de complexité

- On dit qu'un programme a une complexité
  - linéaire si elle est en $\Theta(n)$.
  - quadratique si elle est en $\Theta(n^2)$
  - exponentielle si elle est en $\Theta(2^n)$
  - logarithmique si elle est en $\Theta(\log(n))$
  - en temps constant si elle est en $\Theta(1)$

Le tri par sélection et sa complexité

Les algorithmes de tri en C++

- On va étudier les algorithmes de tri sur les tableau
- L'entrée sera toujours un tableau
- La sortie sera un tableau contenant les mêmes éléments, mais trié par ordre croissant.

Tri par sélection : Présentation

- On commence par trouver le minimum du tableau, et on échange sa position avec la première case du tableau.
- Ensuite, on trouve le minimum dans le reste du tableau et on échange sa position avec la deuxième case du tableau.
- On continue comme cela, jusqu'à avoir trié tout le tableau.

Tri par sélection : Complexité de la recherche de minimum

Pour calculer la complexité, on compte le nombre de comparaisons.

- Trouver le minimum parmis k cases d'un tableau, il faut parcourir toutes ces cases les comparer avec un minimum courant $\rightarrow (k-1)$ comparaisons.
- On va calculer un ordre de grandeur, on peut donc ignorer le -1 qui est négligeable devant k.
- Pour trouver le minimum parmis k cases d'un tableau, on effectue $\Theta(k)$ opérations.

Tri par sélection : Analyse de complexité

- Partant d'un tableau de taille n, on fait les opérations suivantes
  - Minimum du tableau $\rightarrow \Theta(n)$
  - Minimum parmi les $n-1$ cases restantes $\rightarrow \Theta(n-1)$
  - Minimum parmi les $n-1$ cases restantes $\rightarrow \Theta(n-2)$

    $\vdots$
  - Minimum parmi la seul case restante $\rightarrow \Theta(1)$
- **Total** : $\Theta(1+\ldots+n) = \Theta(\frac{n(n+1)}{2}) = \Theta(n^2)$.

Tri par sélection : Vérification expérimentale

- Retourne sur `https://mszula.github.io/visual-sorting`
- Mesurer le temps que met le tri par sélection avec 200.
- Prédire le temps espéré avec 400 entrées, et le mesurer.

Le tri par insertion et sa complexité

Tri par insertion : Présentation

- On commence par trier les deux premières cases du tableau.
- On insère la troisième case à la bonne place pour que les trois premières cases du tableau soient triées.
- On insère la quatrième case à la bonne place pour que les quatres premières cases du tableau soient triées.
- On continue comme cela jusqu'à ce que tout le tableau soit trié.

Tri par insertion : Complexité de l'insertion

On va compter le nombre de comparaisons et d'affectations !

- Pour insérer une case à une position donnée dans un tableau, il faut réaffecter toutes les positions qui se trouvent après.
- Dans le pire des cas (insertion en première position), on est donc forcé de parcourir tout le tableau pour décaler toutes les cases du tableau.
- L'insertion dans un tableau fait donc $\mathcal{O}(n)$ affectations, où n est la taille du tableau.

Tri par insertion : Complexité de la recherche

- Pour trouver la position à laquelle on doit insérer, on parcourt le tableau jusqu'à trouver la bonne position.

- Dans le pire des cas (in faut insérer en dernière position), on parcourt donc tout le tableau

- La rechreche de la position dans un tableau fait donc $\mathcal{O}(n)$ comparaisons, où n est la taille du tableau.

Tri par insertion : Complexité globale

- Mettons tout cela ensemble
  - A la première étape, la recherche et l'insertion coûtent $\mathcal{O}(1)$ comparaisons et $\mathcal{O}(1)$ affectations.
  - A la seconde étape, la recherche et l'insertion coûtent $\mathcal{O}(2)$ comparaisons et $\mathcal{O}(2)$ affectations.

    $\vdots$

  - Jusqu'à la dernière étape (étape n), où elle fait $\mathcal{O}(n)$ comparaisons et $\mathcal{O}(n)$ affectations.

- Au global, cela fait $\mathcal{O}(1 + \ldots + n) = \mathcal{O}(n^2)$ comparaisons et $\mathcal{O}(1 + \ldots + n) = \mathcal{O}(n^2)$ affectations.

Tri par insertion : Optimisation

- Il est possible de réduire le nombre de comparaisons en changeant l'algorithme de recherche de position.
- On peut par exemple utiliser une rechreche dichotomique, qui a une complexité logarithmique.
- Le nombre total de comparaisons $\mathcal{O}(n \log(n))$. La complexité globale reste quadratique dans le pire des cas, à cause des $\mathcal{O}(n^2)$ affectations.