
Introduction à la complexité en temps

Laurent Signac – CC-BY-SA – 01-07-22 0940 8cfc81335cc092d272a5

Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it. – Alan Perlis

Évaluer la complexité en temps d'un algorithme permet de savoir comment le temps d'exécution de l'algorithme va évoluer en fonction de la taille de ses données d'entrées.

Par exemple, rechercher un élément dans une liste prendra plus de temps si la liste est plus longue. C'est ce *plus de temps* qu'il faut quantifier, indépendamment de la machine d'exécution.

Pour réaliser cela, on commence par déterminer ce qui caractérise la taille des données d'entrée. Si on recherche un élément dans une liste ou si on trie une liste, le nombre d'éléments de cette liste est un bon choix pour la taille des données d'entrée. Notons le n .

Dans la suite, on cherche à savoir comment évoluent les temps d'exécution en fonction de n pour différents algorithmes de recherche et de tri : sont-ils constants, linéaires, quadratiques, exponentiels ?

1 Compter les opérations

Afin d'évaluer le temps d'exécution, on est amené à compter le nombre d'opérations effectuées par l'algorithme. Ce nombre dépendra de la taille des données d'entrée (n). Les opérations qu'il faut compter dépendent de ce qu'on veut faire. On peut être amené à compter les comparaisons, ou bien les affectations, ou autre chose. Nous essaierons ici d'estimer un coût global.

Nous appellerons *opération élémentaire* toute opération qui s'effectue en temps constant, c'est à dire en un temps indépendant de la taille des données d'entrées. Reconnaître ces opérations est évident la plupart du temps. Il y a néanmoins parfois quelques pièges :

- Multiplier deux nombres, si ce sont de très grands nombres entiers (comme ça peut être le cas avec Python) ne se fait pas en temps constant, mais en temps proportionnel au nombre de chiffres des nombres à multiplier (arithmétique entière multiprécision)...
- Une ligne comme `lst.sort()` qui trie une liste ne le fait pas en temps constant (on se doute bien que la durée du tri dépend de la taille de la liste). De même, un test d'appartenance comme `if v in collection` aura un temps d'exécution qui dépend de la taille de `collection` si c'est une liste par exemple. En revanche, si `collection` est un ensemble, alors le temps d'exécution sera constant. Le temps d'exécution ne dépend donc pas du nombre de lignes écrites, à plus forte raison lorsqu'on utilise Python.

On estime par contre que :

- les opérations « élémentaires » : affectation, comparaison, opérations arithmétiques (sauf dans le cas de grand nombres) ou composition de tout ça (sans boucle) sont considérées comme ayant un temps d'exécution constant
- le coût de la séquence `p; q` est la somme des coûts de `p` et `q`.
- le coût de `si b alors p sinon q` est inférieur à $\text{cout}(b) + \max(\text{cout}(p), \text{cout}(q))$
- le coût de `for i in it, faire p` est égal à $\text{len}(it) \cdot \text{cout}(p)$ si $\text{cout}(p)$ ne dépend pas de `i` et est égal à $\sum_{i \in it} \text{cout}(p(i))$ sinon
- le coût de `tant que b, faire p` est égal à la somme des coûts de `b` puis `p` à chaque tour (le fait que la condition soit évaluée une fois de plus que le corps de la boucle n'a généralement pas d'importance)

2 Compter les opérations représentatives

Il est courant de s'intéresser à certaines opérations représentatives uniquement. Par exemple le nombre de comparaisons et le nombre d'affectations dans le cas d'un algorithme de tri. Toutefois, dans ce cours, sauf indication contraire, nous évaluerons le nombre d'opérations *élémentaires* total.

3 Recherche linéaire

La fonction suivante recherche un élément dans une liste triée par ordre croissant (version pseudo-code, puis Python)

```

1 fonction recherche_lineaire(lst: list, val)
2   i = 0
3   répéter tant que i < longueur(lst) et lst[i] < val
4     incrémenter i de 1
5   retourner ( i < longueur(lst) et lst[i] = val )

```

```

1 def recherche_lineaire(lst, val) :
2   i = 0
3   while i < len(lst) and lst[i] < val:
4     i = i + 1
5   return i < len(lst) and lst[i] == val

```

On appelle n la taille de la liste passée en paramètre (qui sera aussi la taille des données d'entrées) et on s'intéresse au nombre d'opérations élémentaires.

Dans l'algorithme qui précède, l'affectation, le test de la boucle, l'incréméntation, le `return` final... tout s'exécute en temps constant (en effet, accéder à un élément d'une liste se fait en temps constant, indépendamment de la taille de la liste).

Si on appelle a le temps qui correspond à l'exécution des lignes 2 et 5, et b le temps mis par le test de la ligne 3 et par l'incréméntation de la ligne 4, alors le temps total sera (avec k le nombre de tours de boucle, a et b des constantes indépendantes de n et k) : $T(n) = a + k \times b$

Le nombre de tours de boucle est variable. Dans le cas le plus favorable, la case recherchée est la première, et $k = 1$. Dans le cas le moins favorable, la valeur recherchée est strictement plus grande que la dernière valeur de la liste, et dans ce cas $k = n$. On peut estimer qu'en moyenne (pour être plus précis il faudrait connaître la distribution des nombres dans la liste, ainsi que celle des nombres recherchés...), $k = \frac{n}{2}$.

En moyenne, le temps d'exécution est donc $T(n) = a + \frac{bn}{2} \in \Theta(n)$ (dans la suite on notera $\Theta(n)$ plutôt que $\in \Theta(n)$).

Ce qui nous importe ici, c'est de savoir que le temps d'exécution est linéaire en la taille de la liste. Si la taille double, le temps double...

4 Recherche dichotomique

Il existe un moyen plus efficace de rechercher un élément dans une liste triée, la recherche dichotomique :

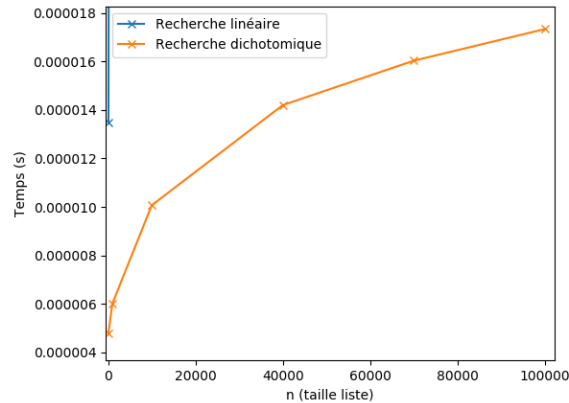
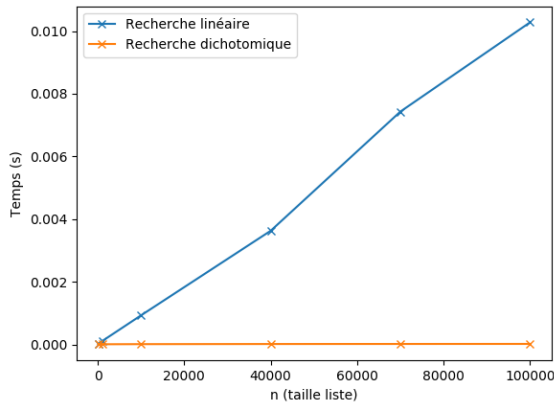
```

1 def recherche_dicho(lst, val):
2   g, d = 0, len(lst) - 1
3   while g != d:
4     m = (g + d) // 2
5     if val > lst[m]:
6       g = m + 1
7     else:
8       d = m
9   return lst[g] == val

```

Le corps de la boucle `while` s'exécute en temps constant (il y a des comparaisons, de l'arithmétique, et des affectations). La boucle s'exécute jusqu'à ce que les indices g et d soient égaux. La distance g, d est divisée par 2 à chaque tour et initialement $d - g + 1$ vaut n où n est le nombre d'éléments de la liste (taille des données d'entrées).

On a donc $\lceil \log_2(n) \rceil$ tours de boucle ($\lceil \log_2(n) \rceil$ est le nombre de fois qu'il faut diviser n par 2 pour obtenir 1 ou moins... dans la suite, on notera simplement $\log_2(n)$ qui n'est pas forcément un nombre entier). Le nombre d'opérations est donc : $a + \log_2(n) \times b$ avec a et b indépendants de n (a est le temps d'exécution des lignes 2 et 9, b est le temps d'exécution d'un tour de boucle). On dit que l'algorithme est en $\Theta(\log(n))$ ou logarithmique. Si on multiplie par 2 la taille de la liste, le temps de calcul n'est pas multiplié par 2, mais on lui ajoute une constante. Quelle que soit la valeur de cette constante, lorsque n devient grand, cet algorithme est incroyablement plus efficace que l'algorithme linéaire précédent.



5 Tri par sélection

L'idée du tri par sélection est la suivante : Rechercher le plus petit élément et le mettre au début, rechercher le second plus petit, le mettre en deuxième position...

Une fois le minimum placé en première position, tout se passe comme si on recommençait la même chose sur tout le tableau privé du premier élément. Puis à nouveau sur tout le tableau privé des deux premiers etc.

```

1 fonction rechmin(lst: list, i: entier) -> entier
2     imin ← i
3     répéter pour j de i à longueur(lst)-1
4         si lst[j] < lst[imin] alors
5             imin ← j
6     retourner imin
7
8 procédure tri_selection(lst: list)
9     répéter pour i de 0 à longueur(lst) - 2
10        j ← rechmin(lst, i)
11        échanger le contenu des cases i et j de lst

```

Si n est la taille de la liste, la fonction `rechmin` s'exécute en un temps qui dépend de n et de son paramètre i . Ce temps vaut $a + (n - i)b$ où a et b sont indépendants de n et de i .

La boucle dans `tri_selection` tourne $n - 1$ fois. Elle contient un échange de variable, qui s'effectue en temps constant c , et un appel à `rechmin`. Chaque appel à `rechmin` s'exécute en un temps différent (car i est passé en paramètre). On a donc :

$$T(n) = \sum_{i=0}^{n-2} c + a + (n - i)b \approx A.n + b \frac{n(n+1)}{2} = \Theta(n^2)$$

Cet algorithme est donc quadratique. Si on double la taille de la liste, le temps d'exécution est multiplié par 4. Si la liste devient 10 fois plus grande, le temps d'exécution sera multiplié par 100 !

6 Tri rapide

```

1 def partitionner(a, p: int, r: int) -> int:
2     x = a[r]
3     i = p - 1
4     for j in range(p, r):
5         if a[j] <= x:
6             i = i + 1
7             a[i], a[j] = a[j], a[i]
8     a[i+1], a[r] = a[r], a[i+1]
9     return i + 1
10
11 def tri_rapide(a, p: int, r: int):
12     if p < r:
13         q = partitionner(a, p, r)
14         tri_rapide(a, p, q - 1)
15         tri_rapide(a, q + 1, r)

```

On se convaincra sans mal que la fonction `partitionner` a un temps d'exécution linéaire en la taille de la partie de la liste à partitionner : $P(n) = a + b \times n$

La fonction `tri_rapide` est récursive, et le calcul de complexité est un peu plus difficile. Dans le cas où la liste est réduite à un seul élément ($p = r$), seul le test de la ligne 12 est évalué (coût c). Dans les autres cas il y a des appels récursifs.

Nous devons ici supposer que, en moyenne, la valeur de q renvoyée par `partitionner` est située au milieu de la tranche p, r . Et que les deux appels récursifs sont donc réalisés sur des tableaux de taille divisée par deux.

Dans ce cas, on a : $T(n) = c + P(n) + 2.T(n/2)$. Dans le cas où n est une puissance de 2, $n = 2^k$, on a :

$$T(2^k) = (a + c) + b.2^k + 2.T(2^{k-1}) = a' + b.2^k + 2T(2^{k-1})$$

Il existe plusieurs méthodes permettant de trouver l'expression de $T(2^k)$ en fonction de k . L'une consiste à combiner les égalités suivantes :

$$\begin{aligned}
 T(2^0) &= c \\
 T(2^1) &= a' + b.2^1 + 2T(2^0) \\
 T(2^2) &= a' + b.2^2 + 2T(2^1) \\
 &\dots \\
 T(2^{k-2}) &= a' + b.2^{k-2} + 2T(2^{k-3}) \\
 T(2^{k-1}) &= a' + b.2^{k-1} + 2T(2^{k-2}) \\
 T(2^k) &= a' + b.2^k + 2T(2^{k-1})
 \end{aligned}$$

Chaque ligne est multipliée par une puissance de 2 :

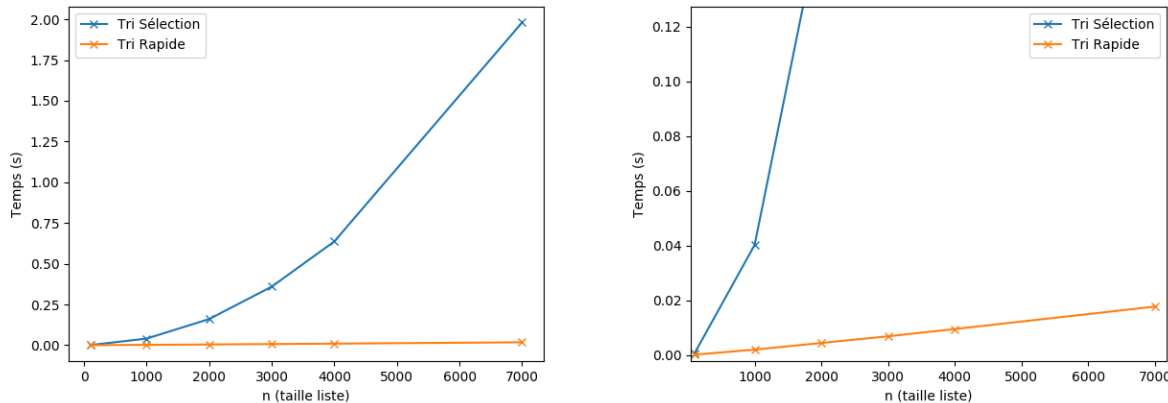
$$\begin{aligned}
 2^k \times T(2^0) &= c \times 2^k \\
 2^{k-1} \times T(2^1) &= (a' + b.2^1 + 2T(2^0)) \times 2^{k-1} \\
 2^{k-2} \times T(2^2) &= (a' + b.2^2 + 2T(2^1)) \times 2^{k-2} \\
 &\dots \\
 2^2 \times T(2^{k-2}) &= (a' + b.2^{k-2} + 2T(2^{k-3})) \times 2^2 \\
 2^1 \times T(2^{k-1}) &= (a' + b.2^{k-1} + 2T(2^{k-2})) \times 2^1 \\
 T(2^k) &= a' + b.2^k + 2T(2^{k-1})
 \end{aligned}$$

Puis en ajoutant membre à membre : $T(2^k) = (a' + c) \times 2^k - a' + b \times k \times 2^k$

C'est à dire : $T(n) \approx b.n.\log_2(n) + (a' + c).n = \Theta(n.\log(n))$

Si dans le cas moyen, le tri rapide a une complexité *linéarithmique*, dans le cas le pire, sa complexité est quadratique (dans les implémentations données plus haut, la constante est même moins bonne que dans le tri par sélection...).

En moyenne, le tri rapide est cependant beaucoup plus efficace que le tri par sélection :



7 Comparer les temps d'exécution

Le tableau suivant est adapté de la page Wikipédia sur l'Analyse de la complexité des algorithmes¹. Il donne une idée, selon le type de complexité, et la taille des données d'entrée, de la manière dont le temps d'exécution peut varier.

| Temps | Type | 10 | 20 | 100 |
|----------------|---------------------------|------------|------------|-------------------------|
| $O(1)$ | complexité constante | 10 ns | 10 ns | 10 ns |
| $O(\log(n))$ | complexité logarithmique | 10 ns | 10 ns | 20 ns |
| $O(n)$ | complexité linéaire | 100 ns | 200 ns | 1 μ s |
| $O(n \log(n))$ | complexité linéarithmique | 100 ns | 260 ns | 2 μ s |
| $O(n^2)$ | complexité quadratique | 1 μ s | 4 μ s | 100 μ s |
| $O(n^3)$ | complexité cubique | 10 μ s | 80 μ s | 10ms |
| $O(2^n)$ | complexité exponentielle | 10 μ s | 10 ms | 4×10^{14} ans |
| $O(n!)$ | complexité factorielle | 36 ms | 770 ans | 3×10^{142} ans |

| Temps | 1 000 | 10 000 | 1 000 000 |
|----------------|---------------------------|-------------|------------|
| $O(1)$ | 10 ns | 10 ns | 10 ns |
| $O(\log(n))$ | 30 ns | 40 ns | 60 ns |
| $O(n)$ | 10 μ s | 100 μ s | 10 ms |
| $O(n \log(n))$ | 30 μ s | 400 μ s | 60 ms |
| $O(n^2)$ | 10 ms | 1 s | 2.8 heures |
| $O(n^3)$ | 10 s | 2.7 heures | 316 ans |
| $O(2^n)$ | 3.4×10^{285} ans | ... | ... |
| $O(n!)$ | ... | ... | ... |

Toujours extrait de la même page, ce tableau donne des exemples d'algorithmes ayant des complexités en temps particulières.

| Temps | Exemple d'algorithme |
|--------|--------------------------------|
| $O(1)$ | accès à une cellule de tableau |

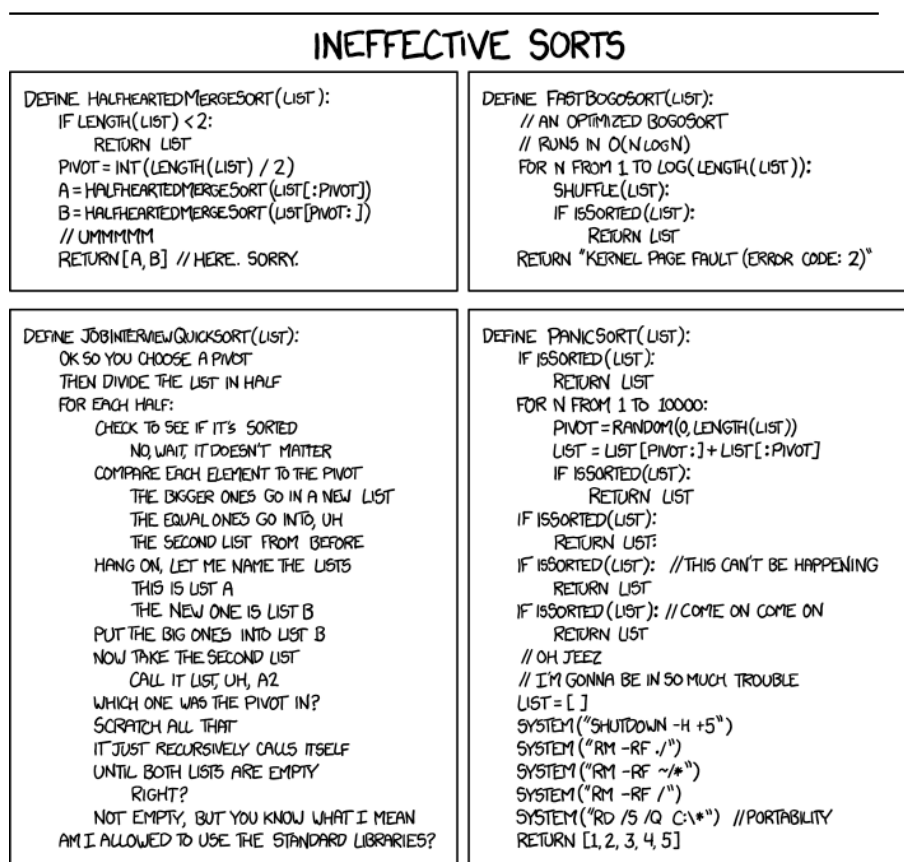
¹https://fr.wikipedia.org/wiki/Analyse_de_la_complexit%C3%A9_des_algorithmes

| Temps | Exemple d'algorithme |
|----------------|--|
| $O(\log(n))$ | recherche dichotomique |
| $O(n)$ | parcours de liste |
| $O(n \log(n))$ | tris par comparaisons optimaux (comme le tri fusion ou le tri par tas) |
| $O(n^2)$ | parcours de tableaux 2D |
| $O(n^3)$ | multiplication matricielle naïve |
| $O(2^n)$ | problème du sac à dos par force brute |
| $O(n!)$ | problème du voyageur de commerce (approche naïve) |

8 Comparer les données d'entrées envisageables

Supposons que pour des données d'entrée de taille $n = 10$, notre programme s'exécute en **1 seconde**. On veut savoir jusqu'à quelle valeur de n on peut monter, tout en conservant une durée d'exécution inférieure à un jour ou un mois. C'est ce qu'indique le tableau suivant.

| Complexité | n pour une exécution < 1 jour | n pour une exécution < 1 mois |
|--------------|---------------------------------|---------------------------------|
| $O(1)$ | $+\infty$ | $+\infty$ |
| $O(\log(n))$ | $\approx 10^{87000}$ | tout plein |
| $O(n)$ | 864 000 | 26 millions |
| $O(n^2)$ | 290 | 1600 |
| $O(n^3)$ | 44 | 137 |
| $O(2^n)$ | 26 | 31 |
| $O(n!)$ | 14 | 15 |



<https://xkcd.com/1185/>