Complexité

Laurent Signac – CC-BY-SA – 06-07-20 1320 b0e5b2499ce7dfcee89d

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.

On cherche à savoir comment évoluent les temps d'exécution en fonction de n pour les deux algorithmes de recherche et de tri : sont-il 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, quelque chose comme : `if v in lst` si `lst` est une liste ne s'exécute pas en temps constant, mais en un temps qui dépend de la taille de la liste. Le temps d'exécution ne dépend évidemment 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 $cout(b) + max(cout(p), cout(q))$
- le coût de `for i in it, faire p` est égal à $len(it).cout(p)$ si $cout(p)$ ne dépend pas de `i` et est égal à $\sum_{i \in it} cout(p(i))$ 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 Recherche linéaire

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

```
fonction recherche_lineaire(lst: list, val)
    i = 0
    répéter tant que i < longueur(lst) et lst[i] < val
        incrémenter i de 1
    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'incrémentation, le return final… tout s'exécute en temps constant (en effet, accéder à un élément d'une liste se fait en temps constant, indépendant 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'incrémentation 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 $T(n) = a + \frac{bn}{2} = \Theta(n)$ (dans a 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…

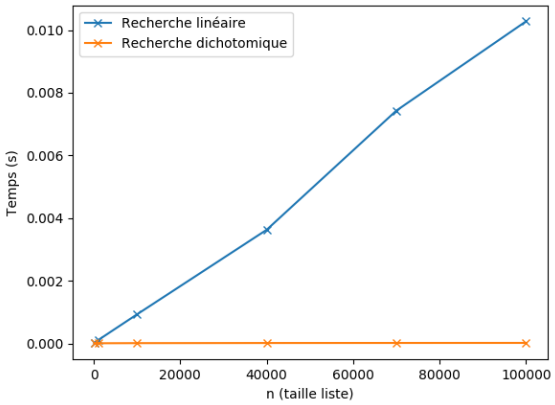
3 Recherche dichotomique

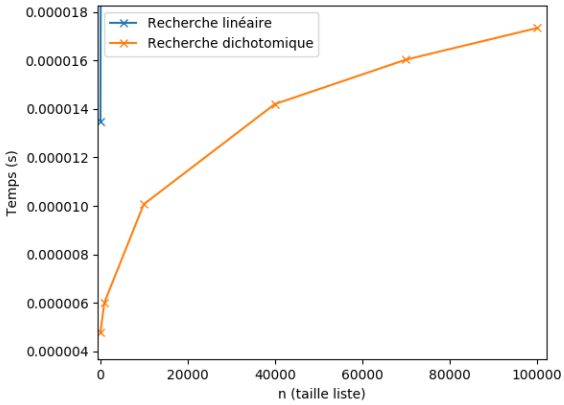
Il existe un moyen plus efficace de rechercher un élément dans une liste triée, la recherche dichotomique :

```
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 $\log_2(n)$ tours de boucle ($\log_2(n)$ est le nombre de fois qu'il faut diviser n par 2 pour obtenir 1 ou moins…). Le nombre d'opérations est donc : $a + \log_2(n) \times b$ avec a et b indépendants de n. 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.





4 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.

```
fonction rechmin(lst: list, i: entier) -> entier
    imin ← i
    répéter pour j de i à longueur(lst)-1
        si lst[j] < lst[imin] alors
            imin ← j
    retourner imin

procédure tri_selection(lst: list)
  répéter pour i de 0 à longueur(lst) - 2
      j ← rechmin(lst, i)
      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 :

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

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 !

5 Tri rapide

```
def partitionner(a, p: int, r: int) -> int:
    x = a[r]
    i = p - 1
    for j in range(p, r):
        if a[j] <= x:
            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 `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 :

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

Chaque ligne est multipliée par une puissance de 2 :

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

Puis en ajoutant membre à membre :

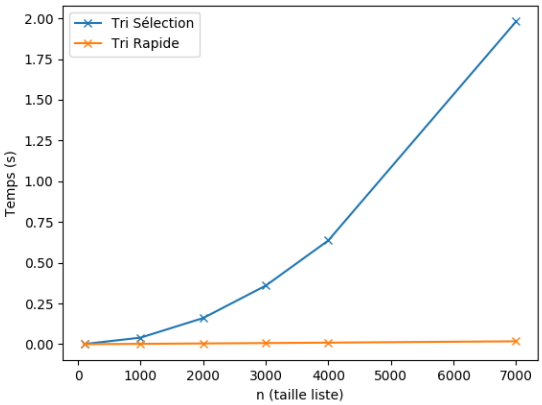
$$T(2^k) = (a' + c) \times 2^k + b \times k \times 2^k$$

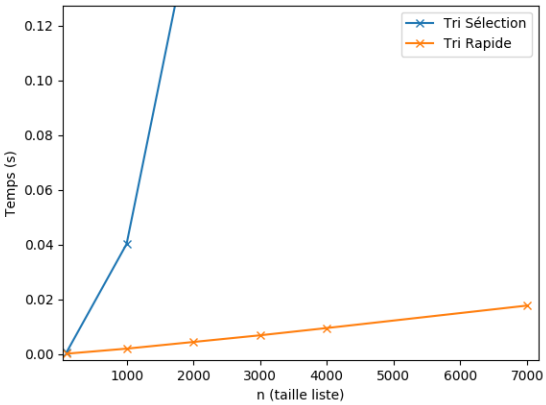
C'est à dire :

$$T(n) = b.n.\log_2(n) + (a' + c).n = \Theta(n.\log(n))$$

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 :





6 Comparer les temps d'exécution

Le tableau suivant est adapté de la page Wikipédia sur l'Analyse de la complexité des algorithmes[1] :

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	1µs
$O(n\log(n))$	complexité linéarithmique	100 ns	260 ns	2µs
$O(n^2)$	complexité quadratique	1 µs	4 µs	100 µs
$O(n^3)$	complexité cubique	10 µs	80 µs	10ms
$O(2^n)$	complexité exponentielle	10 µs	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 µs	100 µs	10 ms
$O(n\log(n))$	30 µs	400 µs	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	…	…
$O(n!)$	…	…	…

Temps	Problème exemple
$O(1)$	accès à une cellule de tableau
$O(\log(n))$	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

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

Temps	Problème exemple
$O(2^n)$	problème du sac à dos par force brute
$O(n!)$	problème du voyageur de commerce (approche naïve)