Complexité : résolution d'inégalités

Judicaël Courant

25 octobre 2018

Table des matières

1 Introduction

Pour majorer ou minorer la complexité d'un algorithme, on part souvent d'inégalités de récurrence. Le but de ce texte est de montrer comment les utiliser pour trouver des majorations ou minorations asymptotiques, en introduisant les techniques utiles sur des exemples.

Il existe sur ce sujet des résultats généraux et notamment un résultat appelé *master theorem*. Mais d'une part celui-ci est hors-programme et d'autre part, avec un tout petit peu d'entraînement, il est plus facile de maîtriser les techniques présentées ci-après plutôt que retenir par cœur ce résultat.

2 La recherche dichotomique

On sait que la complexité en temps d'un appel à une fonction récursive de recherche dichotomique est dominé par le nombre maximum $C(n)$ d'appels récursifs effectués pour effectuer une recherche dichotomique sur un tableau de taille n. Pour majorer $C(n)$, nous allons utiliser une inégalité, qui dépend de l'algorithme précis utilisé.

2.1 Première version

2.1.1 Inégalité

```
def dicho(t, g, d, v):
    """Recherche v dans t[g:d] et retourne k dans range(g, d)
    tel que t[k]=v. Si un tel k n'existe pas, retourne -1."""
    n = d - g # nb d'éléments dans l'intervalle
    if n == 1 and t[g] == v:
        return g # on a trouvé !
    elif n <= 1:
        return -1
    # l'intervalle contient au moins deux éléments
    p = n // 2
    m = g + p
    if v < t[m]:
        return dicho(t, g, m, v)
    else:
        return dicho(t, m, d, v)
```

Le nombre d'appels récursifs $C(n)$ effectués quand on appelle `dicho(t, g, d, v)`

avec $n = d - g$ dans le cas le pire vérifie l'inégalité

$$C(n) \leq 1 + \max\left(C\left(\lfloor n/2 \rfloor\right), C\left(\lceil n/2 \rceil\right)\right) \quad \text{pour } n \geq 2$$

2.1.2 Cas des puissances de 2

Une première idée est de poser $u_k = C(2^k)$, pour $k \in \mathbb{N}$.
On montre alors :

$$u_k \leq 1 + u_{k-1} \quad \text{pour } k \geq 1$$

Alors, pour tout $k \geq 1$:

$$u_k \leq k + u_0$$

C étant une fonction à valeurs positives, on en déduit alors que pour tout $k \geq 1$, on a

$$C(2^k) = O(k)$$

d'où, pour toute puissance de deux n :

$$C(n) = O(\log n)$$

2.1.3 Cas C croissante

Si on fait l'hypothèse que C est croissante, il est aisé d'en déduire le cas général. Il suffit en effet de remarquer que pour tout n, on a $n \leq 2^{\lceil \log_2 n \rceil}$, pour en déduire, par croissance de C :

$$C(n) \leq C\left(2^{\lceil \log_2 n \rceil}\right) = O(\lceil \log_2 n \rceil) = O(\log n)$$

Et C étant à valeurs positives, on en déduit, sous réserve que C soit croissante :

$$C(n) = O(\log n)$$

2.1.4 Cas général

Dans le cas général, on ne peut supposer C croissante. En revanche, il est possible de noter, pour $n \in \mathbb{N}$:

$$C'(n) = \max(C(0), \ldots, C(n))$$

Alors C' est manifestement croissante puisque, pour tout n

$$\max(C(0), \ldots, C(n)) \leq \max(C(0), \ldots, C(n), C(n+1))$$

De plus, on peut alors montrer que C' vérifie, pour $n \geq 1$:

$$C'(n) \leq 1 + \max\left(C'(\lfloor n/2 \rfloor), C'(\lceil n/2 \rceil)\right) = 1 + C'(\lceil n/2 \rceil)$$

En effet, on a

$$\begin{aligned}
C'(n) &= \max(C(0), C(1), \ldots, C(n)) \\
C'(n) &= \max(C(1), \ldots, C(n)) && (\text{car } C(0) \leq C(1)) \\
&\leq \max\Big(1 + \max\left(C(\lfloor 1/2 \rfloor), C(\lceil 1/2 \rceil)\right), \\
&\qquad \ldots, \\
&\qquad 1 + \max\left(C(\lfloor n/2 \rfloor), C(\lceil n/2 \rceil)\right)\Big) \\
&\leq 1 + \max\Big(C'(\lfloor n/2 \rfloor), C'(\lceil n/2 \rceil)\Big)
\end{aligned}$$

On peut donc en déduire, comme précédemment

$$C'(n) = O(\log n)$$

Et comme pour tout n, $0 \leq C(n) \leq C'(n)$, on a

$$C(n) = O(\log n)$$

2.1.5 Rédaction

Nous proposons ici une rédaction de la réponse à une question de la forme «Quelle est, dans le cas le pire, la complexité d'un appel à la fonction `dicho(t, g, d, v)`*, en fonction de $d - g$?»*

On remarque que, lors d'un appel à `dicho(t, g, d, v)`, on effectue, en dehors de l'éventuel appel récursif, un nombre borné d'opérations élémentaires. Le temps de calcul de `dicho(t, g, d, v)` est donc dominé par le nombre d'appels récursifs effectués.

Notons, pour $n \in \mathbb{N}$, $C(n)$ le nombre maximum d'appels récursifs effectués lors de l'exécution d'un appel à `dicho(t, g, d, v)` pour $d - g$ *inférieur ou égal*[1] *à* n.

Pour tout entier n, tout couple (g, d) vérifiant $d - g \leq n$ vérifie $d - g \leq n + 1$, donc $C(n) \leq C(n + 1)$. C est donc une fonction croissante.

Soit $n \geq 2$. Montrons qu'on a

$$C(n) \leq 1 + C\left(\lceil n/2 \rceil\right)$$

Pour cela, considérons un appel à `dicho(t, g, d, v)` où $d - g \leq n$ et montrons qu'il conduit au plus à $1 + C\left(\lceil n/2 \rceil\right)$ appels récursifs. Si $d - g \leq 1$, on n'effectue aucun appel récursif et l'inégalité voulue est vérifiée. Sinon on effectue un appel récursif `dicho(t,g,m,d,v)` ou `dicho(t,g,m,d,v)` avec $m = g + \left\lfloor \frac{d-g}{2} \right\rfloor$. On effectue donc un appel récursif sur un sous-tableau de taille $m - g = \left\lfloor \frac{d-g}{2} \right\rfloor \leq \lceil n/2 \rceil$ dans le premier cas, et de taille $d - m = d - g - \left\lfloor \frac{d-g}{2} \right\rfloor = \left\lceil \frac{d-g}{2} \right\rceil \leq \lceil n/2 \rceil$ dans le second.

1. La fonction C est donc ce que nous avions appelé C' précédemment.

Le nombre d'appels récursifs effectués est donc au plus $1 + C(\lceil n/2 \rceil)$.

Cela étant vrai pour tout appel à `dicho(t, g, d, v)` pour lequel $d - g \leq n$, on a bien

$$C(n) \leq 1 + C(\lceil n/2 \rceil)$$

Par récurrence, on a

$$\forall k \in \mathbb{N} \quad C(2^k) \leq k + C(2^0)$$

Soit alors $n \in \mathbb{N}$. On a

$$0 \leq C(n) \leq C\left(2^{\log_2 n}\right) \leq \log_2 n + C(1)$$

D'où

$$C(n) = O(\log_2 n)$$

2.2 Deuxième version

On cherche maintenant la complexité de l'algorithme donné en cours.

2.2.1 Inégalité

Le nombre d'appels récursifs $C(n)$ effectués quand on appelle `dicho(t, g, d, v)` avec $n = d - g$ dans le cas le pire vérifie l'inégalité

$$C(n) \leq 1 + \max\left(C\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right), C\left(\left\lceil \frac{n-1}{2} \right\rceil\right)\right) \quad \text{pour } n \geq 1$$

2.2.2 Repérer les cas «faciles»

Il est intéressant de remarquer ici que l'inégalité peut être simplifiée lorsque $n-1$ est pair. On peut ainsi montrer aisément

$$\begin{aligned} C(1) &\leq 1 + C(0) \\ C(3) &\leq 1 + C(1) \\ C(7) &\leq 1 + C(3) \\ C(15) &\leq 1 + C(7) \\ C(31) &\leq 1 + C(15) \end{aligned}$$

Autrement dit, en posant $x_0 = 0$ et, pour tout $k \in \mathbb{N}^*$, $x_k = 2x_{k-1} + 1$, on a, pour tout $k \in \mathbb{N}^*$:

$$C(x_k) \leq 1 + C(x_{k-1})$$

Posons alors $u_k = C(x_k)$, pour $k \in \mathbb{N}$.

Alors, pour tout $k \geq 1$:

$$u_k \leq k + u_0$$

C étant une fonction à valeurs positives, on en déduit alors :

$$C(x_k) = O(k)$$

En outre, on a

$$\forall k \in \mathbb{N} \quad x_k = 2^k - 1$$

d'où, pour tout n de la forme $2^k - 1$, avec $k \in \mathbb{N}$:

$$C(n) = O\left(\log\left(n+1\right)\right)$$

2.2.3 Cas C croissante

Si on fait l'hypothèse que C est croissante, il est aisé d'en déduire le cas général. Il suffit en effet de remarquer que pour tout n, on a $n \leq x_{\lceil \log_2(n+1)\rceil} = 2^{\lceil \log_2(n+1)\rceil} - 1$, pour en déduire, par croissance de C :

$$C(n) \leq C\left(x_{\lceil \log_2 n+1\rceil}\right) = O\left(\lceil \log_2\left(n+1\right)\rceil\right) = O(\log n)$$

Et C étant à valeurs positives, on en déduit, sous réserve que C soit croissante :

$$C(n) = O(\log n)$$

2.2.4 Cas général

Dans le cas général, on ne peut supposer C croissante. En revanche, il est possible de noter, pour $n \in \mathbb{N}$:

$$C'(n) = \max(C(0), \ldots, C(n))$$

C' est alors croissante et on peut montrer

$$C'(n) \leq 1 + \max\left(C'\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right), C'\left(\left\lceil \frac{n-1}{2} \right\rceil\right)\right) \quad \text{pour } n \geq 1$$

D'où on déduit successivement

$$C'(n) = O(\log n)$$
$$C(n) = O(\log n)$$

2.2.5 Rédaction

Nous proposons ici une rédaction de la réponse à une question de la forme «Quelle est, dans le cas le pire, la complexité d'un appel à la fonction `dicho(t, g, d, v)`*, en fonction de $d - g$?»*

On remarque que, lors d'un appel à `dicho(t, g, d, v)`, on effectue, en dehors de l'éventuel appel récursif, un nombre borné d'opérations élémentaires. Le temps de calcul de `dicho(t, g, d, v)` est donc dominé par le nombre d'appels récursifs effectués.

Notons, pour $n \in \mathbb{N}$, $C(n)$ le nombre maximum d'appels récursifs effectués lors de l'exécution d'un appel à `dicho(t, g, d, v)` pour $d - g$ *inférieur ou égal à* n.

Pour tout entier n, tout couple (g, d) vérifiant $d - g \leq n$ vérifie $d - g \leq n + 1$, donc $C(n) \leq C(n + 1)$. C est donc une fonction croissante.

Soit $n \geq 2$. Montrons qu'on a

$$C(n) \leq 1 + C\left(\left\lceil \frac{n-1}{2} \right\rceil\right)$$

Pour cela, considérons un appel à `dicho(t, g, d, v)` où $d - g \leq n$ et montrons qu'il conduit au plus à $1 + C\left(\left\lceil \frac{n-1}{2} \right\rceil\right)$ appels récursifs. Si $d - g \leq 1$, on n'effectue aucun appel récursif et l'inégalité voulue est vérifiée. Sinon on effectue un appel récursif `dicho(t,g,m,d,v)` ou `dicho(t,g,m+1,d,v)` avec $m = g + \left\lfloor \frac{d-g-1}{2} \right\rfloor$. On effectue donc un appel récursif sur un sous-tableau de taille $m - g = \left\lfloor \frac{d-g}{2} \right\rfloor \leq \left\lceil \frac{n-1}{2} \right\rceil$ dans le premier cas, et de taille $d - m = d - g - 1 - \left\lfloor \frac{d-g-1}{2} \right\rfloor = \left\lceil \frac{d-g-1}{2} \right\rceil \leq \left\lceil \frac{n-1}{2} \right\rceil$ dans le second.

Le nombre d'appels récursifs effectués est donc au plus $1 + C\left(\left\lceil \frac{n-1}{2} \right\rceil\right)$.

Cela étant vrai pour tout appel à `dicho(t, g, d, v)` pour lequel $d - g \leq n$, on a bien

$$C(n) \leq 1 + C\left(\left\lceil \frac{n-1}{2} \right\rceil\right)$$

Par récurrence, on a

$$\forall k \in \mathbb{N} \quad C\left(2^k - 1\right) \leq k + C(2^0 - 1)$$

Soit alors $n \in \mathbb{N}$. On a

$$0 \leq C(n) \leq C\left(2^{\log_2(n+1)} - 1\right) \leq \log_2(n + 1) + C(0)$$

D'où

$$C(n) = O(\log_2 n)$$

3 Tri fusion

3.1 Inégalités

Notons $T_p(n)$ le temps d'exécution de l'algorithme de tri fusion pour un tableau de taille inférieure ou égale à n dans le cas le pire, $T_m(n)$ ce temps d'exécution dans le cas le meilleur pour un tableau de taille supérieure ou égale à n, $E_p(n)$ l'espace mémoire nécessaire à son exécution (en plus du tableau argument) pour un tableau de taille inférieure ou égale à n dans le cas le pire et $E_m(n)$ l'espace nécessaire pour un tableau de taille supérieure ou égale à n.

Remarquez qu'une conséquence immédiate de ces définitions est que T_p, T_m, E_p et E_m sont croissantes.

On peut montrer les inégalités suivantes :

$$T_p(n) \leq 2T_p(\lceil n/2 \rceil) + O(n) \tag{1}$$
$$T_m(n) \geq 2T_m(\lfloor n/2 \rfloor) + \Omega(n) \tag{2}$$
$$E_p(n) \leq E_p(\lceil n/2 \rceil) + O(n) \tag{3}$$
$$E_m(n) \geq E_m(\lfloor n/2 \rfloor) + \Omega(n) \tag{4}$$

3.2 Complexité en temps dans le cas le pire

D'après l'inégalité (1), il existe deux constantes α et β vérifiant

$$T_p(n) \leq 2T_p(\lceil n/2 \rceil) + \alpha n + \beta$$

On a alors, pour tout $k \in \mathbb{N}$:

$$T_p(2^{k+1}) \leq 2T_p(2^k) + \alpha \cdot 2^{k+1} + \beta$$

Posons alors $u_k = T_p(2^k)/2^k$. *Remarque : ce choix n'est évidemment pas fait au hasard. On divise par (2^k), qui est une solution particulière de l'équation homogène associée à l'inéquation $v_{k+1} \leq 2v_k + \alpha 2^{k+1} + \beta$.* On a alors :

$$u_{k+1} \leq u_k + \alpha + \beta 2^{-(k+1)}$$

Ce qui nous permet de majorer $u_k - u_0$ par somme téléscopique.

On obtient ainsi successivement :

$$\begin{aligned} u_k - u_0 &= \sum_{i=0}^{k-1} (u_{i+1} - u_i) \\ u_k - u_0 &\leq \sum_{i=0}^{k-1} \left(\alpha + \beta 2^{-(k+1)}\right) \\ u_k &\leq \alpha k + \beta + u_0 \end{aligned}$$

D'où

$$\begin{aligned} T_p(n) &\leq T_p\left(2^{\lceil \log_2 n \rceil}\right) \\ &\leq 2^{\lceil \log_2 n \rceil} \cdot u_{\lceil \log_2 n \rceil} \\ &\leq 2n \cdot O\left(\lceil \log_2 n \rceil\right) \\ &\leq O(n \log n) \end{aligned}$$

Et comme $T_p(n) \geq 0$, on a

$$T_p(n) = O(n \log n)$$

3.3 Complexité en temps dans le cas le meilleur

De la même façon, l'inégalité (2) conduit à poser $v_k = T_m(2^k)/2^k$. On montre alors, comme précédemment

$$v_k \geq \Omega(k)$$

D'où on déduit

$$T_m(n) = \Omega(n \log n)$$

$$\begin{aligned} T_m(n) &\geq T_m\left(2^{\lfloor \log_2 n \rfloor}\right) \\ &\geq 2^{\lfloor \log_2 n \rfloor} \cdot u_{\lfloor \log_2 n \rfloor} \\ &\geq \frac{1}{2} n \cdot O\left(\lfloor \log_2 n \rfloor\right) \\ &\geq O(n \log n) \end{aligned}$$

3.4 Complexité en espace dans le cas le pire

Le traitement de l'inégalité (3) est à peine différent. On remarque tout d'abord qu'il existe λ et μ vérifiant

$$E_p(n) \leq E_p(\lceil n/2 \rceil) + \lambda n + \mu$$

On a alors, pour tout $k \in \mathbb{N}$:

$$E_p(2^{k+1}) \leq E_p(2^k) + \lambda \cdot 2^{k+1} + \mu$$

On pose alors $w_k = E_p(2^k)$ et on obtient successivement :

$$\begin{aligned} w_k - w_0 &= \sum_{i=0}^{k-1} (w_{i+1} - w_i) \\ w_k - w_0 &\leq \sum_{i=0}^{k-1} \left(\lambda 2^{i+1} + \mu\right) \\ w_k &\leq (2^k - 1) \cdot 2\lambda + k\mu + w_0 \end{aligned}$$

D'où

$$E_p(n) = O(n)$$

3.5 Complexité en espace dans le cas le meilleur

Enfin, on fait de même pour l'inégalité (4) et on trouve

$$E_m(n) = \Omega(n)$$

3.6 Conclusion

La complexité du tri fusion pour un tableau de taille n est donc $\Theta(n \log n)$ en temps et $\Theta(n)$ en espace, quelles que soient les données (le rapport de la complexité dans le cas le pire et le cas le meilleur est majoré par une constante, indépendamment de n — il est évidemment minoré par 1).

4 Algorithme de Karatsuba

La complexité en temps $C(n)$ de la multiplication de deux nombres d'au plus n chiffres par l'algorithme de multiplication rapide de Karatsuba vérifie, pour $n \geq 2$, l'inégalité

$$C(n) \leq 3C(\lceil n/2 \rceil + 1) + O(n)$$

Notons $x_0 = 3$, et pour $k \in \mathbb{N}$, $x_{k+1} = 2(x_k - 1)$. Alors pour tout $k \in \mathbb{N}$, $x_k = 2^k + 2$. De plus

$$C(x_{k+1}) \leq 3C(x_k) + O(2^k)$$

On pose alors $v_k = C(x_k) \cdot 3^{-k}$. On a :

$$v_{k+1} - v_k \leq O\left(\left(\frac{2}{3}\right)^k\right)$$

Donc la série des $\sum_k (v_{k+1} - v_k)$ est majorée par une série convergente, donc la suite (v_k) est majorée. Elle est de plus minorée par 0, donc $v_k = O(1)$.

On en déduit

$$C(x_k) = O(3^k)$$

Remarquons alors que pour avoir $x_k \geq n$, il suffit d'avoir $2^k + 2 \geq n$ et pour cela, il suffit d'avoir $k \geq \log_2(n-2)$ (pour $n \geq 3$).

Donc, pour tout $n \geq 3$, on a $n \leq x_{\lceil \log_2(n-2) \rceil}$ et

$$\begin{aligned} C(n) &\leq C\left(x_{\lceil \log_2(n-2) \rceil}\right) \\ &\leq O\left(3^{\lceil \log_2(n-2) \rceil}\right) \\ &\leq O\left(\exp\left(\ln 3 \cdot \ln n / \ln 2\right)\right) \\ &\leq O\left(n^{\ln 3 / \ln 2}\right) \\ &\leq O\left(n^{\log_2 3}\right) \end{aligned}$$

Par croissance de $\log_2$*, on a évidemment* $1 < log_2 3 < 2$*. Plus précisément, on a* $1,58 < log_2 3 < 1,59$*.*

5 Construction d'un tas

Ce problème fait partie du programme de l'option informatique. Mais l'équation obtenue est un exemple intéressant dans le cadre du tronc commun.

Contexte pour les étudiants d'option informatique : La construction d'un tas de taille n en $O(n)$ peut se faire de façon récursive. Il s'agit essentiellement de faire les mêmes opérations de percolation que dans la construction par forêt mais en exprimant l'algorithme de façon récursive. Intuitivement l'idée est que pour $n > 1$, on prend k' le plus grand entier tel que $2^{k'} - 1 \leq \frac{n-1}{2}$ et on construit un tas avec $2^{k'} - 1'$ éléments. Puis on construit un tas avec les éléments restants sauf un, soit $n - 2^{k'} + 1 - 1$ éléments

(on a $n - 2^{k'} \leq 2^{k'} - 1$), et enfin on réunit les deux tas en un seul, en effectuant une percolation descendante de coût $O(\log n)$.

En notant $C(n)$ le temps mis par cet algorithme pour construire un tas de taille au plus n, on a l'inégalité suivante :

$$C(n) \leq 2C\left(2^{-1+\log_2(n-1)}\right) + O(\log n)$$

Pour tous : il s'agit donc de résoudre l'inégalité ci-dessus.

En posant $x_k = 2^k - 1$, on a alors

$$C(x_{k+1}) \leq 2C\left(x_k\right) + O(k)$$

D'où, en posant $u_k = C(x_k) \cdot 2^{-k}$,

$$u_{k+1} \leq u_k + O\left(k \cdot 2^{-k}\right)$$

La série $\sum_k k \cdot 2^{-k}$ est à valeurs positives et converge, donc la série $\sum_k (u_{k+1} - u_k)$ est majorée, donc (u_k) est majorée, d'où $C(x_k) \leq O(2^k)$ et, par croissance, de C, $0 \leq C(n) \leq C\left(x_{\lceil \log_2(n+1) \rceil}\right) \leq O\left(2^{\lceil \log_2(n+1) \rceil}\right)$. D'où

$$C(n) = O(n)$$

6 Tri rapide

6.1 Cas le pire

La complexité en temps du tri rapide sur un tableau d'au plus n éléments est dominée par le nombre de comparaisons $C_p(n)$ effectuées dans le cas le pire. Ce nombre de comparaison vérifie, $C_p(0) = 0$ et pour $n \geq 0$:

$$C_p(n) \leq \max_{i \in [\![0,n[\![} \left(C_p(i) + C_p(n-i-1)\right) + n - 1$$

Il est difficile d'appliquer les techniques précédentes directement. Cependant, on peut assez vite conjecturer que $C_p(n)$ est majoré par le nombre de comparaisons u_n qu'il faut effectuer dans le cas d'un tableau déjà trié par ordre croissant.

La suite u vérifie $u_0 = 0$ et pour tout $n \geq 1$, $u_n = u_{n-1} + n - 1$.

On a alors :

$$\forall n \in \mathbb{N} \quad u_n = \frac{n(n-1)}{2}$$

Remarquons que pour tout $n \geq 1$ et $i \in [\![0, n[\![$, on a

$$\begin{aligned} u_0 + u_{n-1} &= \frac{1}{2}\left(n^2 - 3n + 2\right) \\ u_i + u_{n-i-1} &= \frac{1}{2}\left(i^2 - i + (n-i)^2 - 3(n-i) + 2\right) \\ &= \frac{1}{2}\left(n^2 - 3n + 2 - 2i(n-i-1)\right) \end{aligned}$$

Or

$$\begin{aligned} i^2 - i + (n-i)^2 - 3(n-i) + 2 &= n^2 - 3n + 2i^2 + 2i - 2ni \\ &= n^2 - 3n + 2 - 2i(n-i-1) \end{aligned}$$

D'où

$$u_i + u_{n-i-1} \leq u_0 + u_{n-1}$$

Nous pouvons alors montrer par récurrence forte que, pour tout $n \in \mathbb{N}$, $C_p(n)$ est majorée par u_n :

— D'une part, on a évidemment $C_p(0) \leq u_0$.

— D'autre part, pour tout $n \geq 1$ tel que pour tout $k < n$, $C_p(k) \leq u_k$, on a :

$$\begin{aligned} C_p(n) &\leq \max_{i \in [\![0,n[\![} (C_p(i) + C_p(n-i-1)) + n - 1 \\ &\leq \max_{i \in [\![0,n[\![} (u_i + u_{n-i-1}) + n - 1 \\ &\leq u_0 + u_{n-1} + n - 1 \\ &\leq u_n \end{aligned}$$

D'après le principe de récurrence, on a donc, pour tout $n \in \mathbb{N}$, $C_p(n) \leq u_n$. On en déduit

$$C_p(n) = O(n^2)$$

De plus, u_n étant le nombre de comparaisons dans le cas où le tableau est trié, on a $C_p(n) \geq u_n = \Omega(n^2$. Donc la majoration asymptotique donnée est serrée :

$$C_p(n) = \Omega(n^2)$$

6.2 Cas le meilleur

Notons $C_m(n)$ le nombre de comparaisons effectuées par le tri par pivot dans le cas le meilleur sur un tableau de taille au moins n. Alors

$$C_m(n) \geq \min_{i \in [\![0,n[\![} (C_m(i) + C_m(n-i-1)) + n - 1$$

6.2.1 Conjectures

Comme dans le cas le pire, il est difficile d'appliquer directement les techniques précédentes. Cependant on peut émettre deux conjectures :

1. En notant (x_k) la suite définie par $x_0 = 0$ et $x_{k+1} = 2x_k + 1$, il est possible, pour tout k, de trouver un tableau t_k de taille x_k tel que le tri par pivot se passe de façon «agréable», c'est-à-dire que l'exécution du tri sur ce tableau va conduire

récursivement à trier deux sous-tableaux de taille $(x_k - 1)/2 = x_{k-1}$, ce qui va conduire récursivement à trier quatre sous-tableaux de taille x_{k-2}, puis 2^3 tableaux de taille x_{k-3}, ..., puis 2^k tableaux de taille $x_0 = 0$. On note alors v_k le nombre de comparaisons effectuées.

2. Pour un tableau de taille x_k, cette exécution est la meilleure possible, c'est-à-dire $C_m(x_k) = v_k$.

Le terme générique de la suite (x_k) est $2^k - 1$. De plus la suite (v_k) vérifie $v_0 = 0$ et pour tout $k \geq 1$:

$$v_k = 2v_{k-1} + x_k - 1 = 2v_{k-1} + 2^k - 2$$

En posant $w_k = v_k \cdot 2^{-k}$, on a

$$w_k = w_{k-1} + 1 - 2^{1-k}$$

Par somme téléscopique, et en utilisant que $w_0 = 0$, on obtient :

$$w_k = k - \sum_{i=1}^{k} 2^{1-i} = k - 2 + 2^{1-k}$$
$$\text{d'où } v_k = 2^k k - 2^{k+1} + 2 = 2^k \cdot (k-2) + 2$$

En posant $n = x_k$, on a $n = 2^k - 1$, donc $k = \log_2(n+1)$, donc $v_k = (n+1)(\log_2(n+1) - 2) + 2$.

6.2.2 Démonstration

Ce qui précède nous conduit à poser, pour $x \in [0, +\infty[$, $f(x) = (x+1)(\log_2(x+1) - 2) + 2$.

Montrons alors par récurrence forte que pour tout n, on a $C_m(n) \geq f(n)$.

— On a évidemment $C_m(0) \geq 0 = f(0)$.

— Soit n un entier tel que pour tout $k < n$, on a $C(k) \geq f(k)$. On a alors

$$C_m(n) \geq \min_{i \in [\![0,n[\![} (f(i) + f(n-i-1)) + n - 1$$

Posons, pour $x \in [0, n-1]$, $\phi(x) = f(x) + f(n-1-x)$ et étudions le sens de variation de ϕ. On a évidemment f et ϕ dérivables, et pour $x \in [0, n-1]$, on trouve successivement :

$$f'(x) = \frac{\ln(x+1)}{\ln 2} - 2 + \frac{1}{\ln 2}$$
$$\phi'(x) = \frac{\ln(x+1)}{\ln 2} - \frac{\ln(n-x)}{\ln 2}$$

$\phi'(x)$ est donc du signe de $\ln(x+1)-\ln(n-x)$, qui est négatif pour $x \leq (n-1)/2$ et positif pour $x \geq (n-1)/2$. ϕ atteint donc un minimum en $(n-1)/2$. On a donc :

$$C_m(n) \geq \min_{i \in [\![0,n[\![} \phi(i) + n - 1$$
$$C_m(n) \geq \phi\left(\frac{n-1}{2}\right) + n - 1$$

Or on a :

$$\begin{aligned}
\phi\left(\frac{n-1}{2}\right) + n - 1 &= 2f\left(\frac{n-1}{2}\right) + (n-1) \\
&= 2\left(\frac{n-1}{2}+1\right)\left(\log_2\left(\frac{n-1}{2}+1\right)-2\right) + 4 + n - 1 \\
&= (n+1)\left(\log_2(n+1) - 1 - 2\right) + 4 + n - 1 \\
&= (n+1)\left(\log_2(n+1) - 2\right) - (n+1) + 4 + n - 1 \\
&= (n+1)\left(\log_2(n+1) - 2\right) + 2 \\
&= f(n)
\end{aligned}$$

On en déduit $C_m(n) \geq f(n)$.

D'après le principe de récurrence forte, on a donc

$$\forall n \in \mathbb{N} \quad C_m(n) \geq f(n)$$

En particulier

$$C_m(n) = \Omega(n \log n)$$

Remarquons que nous avons démontré ce résultat sans faire appel à nos deux conjectures initiales.

Nous admettons maintenant la première (on peut la démontrer mais c'est assez pénible à faire).

Alors la seconde est facile à démontrer : pour tout $k \in \mathbb{N}$, on a $C_m(x_k) \geq f(x_k) = v_k$ et comme v_k est le nombre de comparaisons effectuées pour un choix particulier de tableau t_k, on a $C_m(x_k) \leq v_k$, d'où l'égalité.

Pour tout $n \in \mathbb{N}$, on a $n \leq x_{\lceil \log_2(n+1) \rceil}$. Or C_m est croissante, donc

$$C_m(n) \leq C_m(x_k) = v_k = f\left(2^{\lceil \log_2(n+1) \rceil}\right) = O(n \log n)$$

La minoration que nous avons donnée est donc une minoration serrée :

$$C_m(n) = \Theta(n \log n)$$

