Complexité

INFORMATIQUE COMMUNE - TP n° 2.2 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ calculer la complexité d'un algorithme simple
- ☞ donner les complexités dans le pire des cas des tris comparatifs génériques
- ☞ expliquer la différence entre le tri rapide et le tri fusion

A Complexité d'algorithmes simples

A1. Calculer la complexité de l'algorithme 1. Y-a-il un pire et un meilleur des cas ?

Algorithme 1 Produit scalaire

```
1 : Fonction PRODUIT_SCALAIRE(x, y)        ▷ x et y sont des vecteurs à n éléments
2 :     s ← 0
3 :     pour i = 0 à n − 1 répéter
4 :         s ← s + x_i y_i                ▷ coût ?
5 :     renvoyer s
```

Solution : Il n'y a pas de pire ou meilleur cas.
On fait l'hypothèse que l'affectation et l'addition ont un coût constant c.

$$C(n) = c + \sum_{i=0}^{n-1} c = c + c \sum_{i=0}^{n-1} 1 = c + cn = O(n)$$

A2. Calculer la complexité de l'algorithme 2 dans le meilleur et dans le pire des cas.

Solution : Dans le meilleur des cas, la première lettre est différente de la dernière : l'algorithme renvoie donc faux dès le premier tour de boucle. La complexité est constante, car il s'agit d'un nombre fini d'opérations qui ne dépendent pas de la dimension de la chaîne de caractères d'entrée.
Dans le pire des cas, le mot est un palindrome. La boucle tant que effectue `n//2` itérations puisque les lettres sont comparées deux à deux. La complexité est donc linéaire en $O(n)$.

Algorithme 2 Palindrome

```
1 : Fonction PALINDROME(w)
2 :     n ← la taille de la chaîne de caractères w
3 :     i ← 0
4 :     j ← n − 1
5 :     tant que i < j répéter
6 :         si w[i] = w[j] alors
7 :             i ← i + 1
8 :             j ← j − 1
9 :         sinon
10 :            renvoyer Faux
11 :    renvoyer Vrai
```

A3. Calculer la complexité de l'algorithme 3. Y-a-t-il un pire et un meilleur des cas ? On fait l'hypothèse que la fonction *power*(a,b) est de complexité logarithmique $O(1)$. Cette hypothèse vous semble-t-elle raisonnable ?

Algorithme 3 Évaluation simple d'un polynôme

```
1 : Fonction EVAL_POLYNÔME(p, v)
2 :     d ← degré de p
3 :     r ← p[0]
4 :     pour i = 1 à d répéter
5 :         r ← r + p[i] ×POWER(v, i)
6 :     renvoyer r
```

Solution : Il n'y a pas de pire ou meilleur cas.

L'hypothèse sur la complexité de la fonction *power* peut paraître surprenante car avec l'algorithme d'exponentiation rapide qui est de type diviser pour régner on atteint la complexité $O(\log n)$, ce qui est déjà bien. La complexité $O(1)$ ne peut être atteinte qu'avec le support de l'électronique et de l'implémentation de Floating Point Unit, des circuits dédiés aux calculs sur les flottants qui implémente les logarithmes binaires. En les utilisant explicitement dans le code, on peut calculer $p^i = \exp(i \log p)$ avec une approximation correct et la complexité est quasiment constante.

La complexité s'exprime en fonction du degré du polynôme qui caractérise la variation de la donnée d'entrée. Pour un même degré, la complexité sera toujours la même.

$$C(d) = c + \sum_{i=1}^{d} c = c + c \sum_{i=1}^{d} 1 = c + cd = O(d)$$

La complexité de cette fonction est donc linéaire.

A4. Utiliser la méthode de Horner pour écrire un autre algorithme pour évaluer un polynôme. Quelle complexité pouvez-vous obtenir ? Est-ce plus rapide ?

Solution : L'algorithme de Horner permet de se passer d'appel à la fonction *power*, elle est donc a priori plus rapide même si la complexité reste linéaire au final.

$$C(n) = c + \sum_{i=0}^{d-1} 2c = c + 2cd = O(d)$$

Sur une machine basique sans circuits spécifiques pour calculer efficacement les puissances sur les flottants, il est fortement conseillé d'utiliser cette méthode. On peut vérifier en Python que la méthode d'Horner est toujours la plus rapide.

```
1 : Fonction HORNER(p, d, v)
2 :     r ← p[d]
3 :     pour i = d − 1 à 0 répéter
4 :         r ← r × v
5 :         r ← r + p[i]
6 :     renvoyer r
```

B Tri fusion

Algorithme 4 Tri fusion

```
1 : Fonction TRI_FUSION(t, g, d)
2 :     si g < d alors
3 :         m ← (g+d)//2                          ▷ on découpe au milieu
4 :         TRI_FUSION(t, g, m)
5 :         TRI_FUSION(t, m+1, d)
6 :         FUSION(t, g, m, d)
                                                  ▷ Sinon on n'arrête !
```

B1. Programmer le tri fusion en Python.

Solution :

```python
def merge(t,g,m,d):
    ng = m - g + 1
    nd = d - m
    G = [0] * (ng)
    D = [0] * (nd)

    for i in range(0, ng):
        G[i] = t[g + i]

    for j in range(0, nd):
        D[j] = t[m + 1 + j]

    i = 0
    j = 0
    k = g
```

Algorithme 5 Fusion de sous-tableaux triés

```
1 : Fonction FUSION(t, g, m, d)
2 :     ng ← m - g +1
3 :     nd ← d - m
4 :     G, D ← deux tableaux de taille ng et nd
5 :     pour k de 0 à ng répéter
6 :         G[k] ← t[g + k]
7 :     pour k de 0 à nd répéter
8 :         D[k] ← t[m + 1 + k ]
9 :     i ← 0
10 :    j ← 0
11 :    k ← g
12 :    tant que i < ng et j <nd répéter
13 :        si G[i] ⩽ D[j] alors
14 :            t[k] ← G[i]
15 :            i ← i + 1
16 :        sinon
17 :            t[k] ← D[j]
18 :            j ← j + 1
19 :        k ← k + 1
20 :    tant que i < ng répéter
21 :        t[k] ← G[i]
22 :        i ← i + 1
23 :        k ← k + 1
24 :    tant que j < nd répéter
25 :        t[k] ← D[j]
26 :        j ← j + 1
27 :        k ← k + 1
```

```
16
17      while i < ng and j < nd:
18          if G[i] <= D[j]:
19              t[k] = G[i]
20              i += 1
21          else:
22              t[k] = D[j]
23              j += 1
24          k += 1
25
26      while i < ng:
27          t[k] = G[i]
28          i += 1
29          k += 1
30
31      while j < nd:
32          t[k] = D[j]
33          j += 1
34          k += 1
35
36  def merge_sort(t, g, d):
37      if g < d:
38          m = (g + d) // 2
39          merge_sort(t, g, m)
40          merge_sort(t, m +1, d)
41          merge(t, g, m, d)
```

B2. Quelle est la complexité de cet algorithme ? Y-a-t-il un pire et un meilleur cas ?

Solution : il n'y a pas de pire ou meilleur cas : l'algorithme effectue systématiquement la découpe et la fusion des sous-tableaux.

Pour le calcul de la complexité, on a la relation de récurrence $T(n) = 2T(n/2) + f(n)$ où $f(n)$ représente le nombre d'opérations élémentaires nécessaires pour fusionner deux sous-tableaux de taille $n/2$. On fait l'hypothèse que n est une puissance de deux pour simplifier le calcul.

La condition d'arrêt de la récursivité est atteinte pour $\frac{n}{2^k} = 1$ c'est à dire pour $k = \log_2 n$. Par ailleurs, on remarque que $T(1) = c$ est un opération à coût constant, suite à la condition d'arrêt on ne fait que retourner le tableau non modifié.

Dans le pire des cas, l'étape de fusion des deux tableaux effectue la combinaison de deux sous-tableaux de dimension $n/2$. Les boucles de la fonction FUSION ne sont pas imbriquées et effectuent $n/2$ tours au maximum. On a alors $f(n) = cn$, la complexité de la fusion est linéaire.

$$
\begin{aligned}
T(n) &= 2T(n/2) + cn && (1)\\
&= 4T(n/4) + 2cn && (2)\\
&= 8T(n/8) + 3cn && (3)\\
&= \dots && (4)\\
&= 2^k T(1) + kcn && (5)\\
&= O(\log_2 n + n\log_2 n) && (6)\\
&= O(n\log n) && (7)
\end{aligned}
$$

C'est pourquoi, la complexité du tri fusion vaut $T(n) = O(n\log n)$

B3. Vérifier, en mesurant le temps d'exécution, la justesse de votre calcul précédent.

C Tri rapide

Algorithme 6 Tri rapide

```
1 : Fonction TRI_RAPIDE(t, p, d)
2 :     si p < d alors
3 :         i_pivot ← PARTITION(t, p, d)
4 :         TRI_RAPIDE(t, p, i_pivot -1)
5 :         TRI_RAPIDE(t, i_pivot+1, d)
                                          ▷ Sinon on n'arrête !
```

Algorithme 7 Partition en deux sous-tableaux

```
1 : Fonction PARTITION(t, p, d)
2 :     i_pivot ← un nombre au hasard entre p et d inclus
3 :     pivot ← t[i_pivot]
4 :     ÉCHANGER(t,d,i_pivot)                ▷ On met le pivot à la fin du tableau
5 :     i = p - 1                            ▷ i va pointer sur le dernier élément du premier tableau
6 :     pour j = p à d -1 répéter
7 :         si t[j] ⩽ pivot alors
8 :             i ← i + 1
9 :             ÉCHANGER(t,i,j)              ▷ On échange les places de t[i] et t[j]
10 :    t[d] ← t[i+1]                        ▷ t[i+1] appartient au tableau de droite
11 :    t[i+1] ← pivot                       ▷ Le pivot est entre les deux tableaux
12 :    renvoyer i + 1                       ▷ La place du pivot !
```

C1. Programmer le tri rapide en Python.

Solution :

```python
def partition(t, p, d):
    i_pivot = randrange(p, d + 1)  # random pivot
    pivot = t[i_pivot]
    t[i_pivot], t[d] = t[d], pivot  # put it at the end
    #pivot = t[d] # or take the last
    i = p - 1
    for j in range(p, d):
        if t[j] <= pivot:
            i = i + 1
            t[i], t[j] = t[j], t[i]  # swap !
    t[i+1], t[d] = t[d], t[i+1]
    # put the pivot at the right place. The other is greater than the pivot
    return i+1


def quick_sort(t, p, d):
    if p < d:
        i_pivot = partition(t, p, d)
        quick_sort(t, p, i_pivot - 1)
        quick_sort(t, i_pivot + 1, d)
```

C2. Quelle est la complexité de cet algorithme ? Y-a-t-il un pire et un meilleur cas ?

Solution : Il y a effectivement un pire et un meilleur des cas, selon le choix du pivot.

Dans cette implémentation, on a choisi un pivot aléatoirement, ce qui permet de garantir une bonne complexité dans la pratique.

Cependant, cela ne nous épargne pas d'un état du tableau particulier et d'un choix malheureux :

cas 1 le tableau peut être trié et on peut choisir le premier élément comme pivot.

cas 2 le tableau peut être trié en sens inverse et on peut choisir le dernier élément comme pivot.

cas 3 le tableau peut ne comporter que des éléments identiques.

Dans ces trois cas, la complexité du tri rapide est mauvaise.

cas 1 la partition produit toujours un tableau à un élément (P) et autre à $n-1$ éléments (D). Sa complexité est linéaire en $O(n-1)$ à cause de la boucle for. La complexité du tri est directement lié au nombre d'appels récursifs : $T(n) = T(P) + T(D) + n - 1$. Si $T(P)$ termine immédiatement, il faudra $n-1$ appels récursifs pour résoudre le problème $T(D)$. Au final, on a donc

$$T(n) = T(P) + T(D) + n - 1 = 1 + (T(P') + T(D') + n - 2) + n - 1 = n + \sum_{i=0}^{n-1} k = O(n^2)$$

.

cas 2 On fait le même raisonnement et on aboutit au même résultat.

cas 3 Quelque soit le pivot choisi, la partition va résulter en un tableau à un seul élément et un tableau à $n-1$ éléments. On fait le même raisonnement et on aboutit à la même conclusion.

Donc, dans le pire des cas, l'algorithme de tri rapide est en $O(n^2)$.

Dans le meilleur des cas, l'arbre des appels récursifs est équilibré, le pivot choisi est toujours la médiane : on divise par deux le tableau à chaque étape. Le nombre d'appels récursif est $\log n$ et la complexité $O(n \log n)$.

On peut étudier la complexité moyenne de cet algorithme : c'est le cas lorsqu'on choisit aléatoirement le pivot par exemple. Dans le cas, la récurrence s'écrit :

$$T(n) = \left(\frac{1}{n} \sum_{i=0}^{n-1} T(i) - T(n-i-1) \right) + n - 1 \tag{8}$$

$$= \left(\frac{2}{n} \sum_{i=0}^{n-1} T(i) \right) + n - 1 \tag{9}$$

$$\tag{10}$$

En effet, si le tableau possède n éléments et qu'on choisit le pivot au hasard, on a une probabilité $1/n$ de choisir le pivot d'indice i. On choisit la convention que le premier élément du tableau

est 0. Si on fait ce choix, alors le premier tableau comportera i éléments et le dernier $n-i-1$. D'où la récurrence.

En multipliant à gauche par 2 et n, on obtient :

$$nT(n) = n(n-1) + \sum_{i=0}^{n-1} T(i) \tag{11}$$

En travaillant un peu cette équation récursive, on obtient :

$$nT(n) - (n+1)T(n-1) = 2(n-1) \tag{12}$$

Finalement, en divisant des deux côtés par $n(n+1)$, on a :

$$\frac{T(n)}{n+1} = \frac{1}{3} + 2\sum_{i=3}^{n} \frac{1}{i+1} - 2\sum \lim_{i=3}^{n} \frac{1}{i(i+1)} \tag{13}$$

La première série est la série harmonique, la seconde converge.

Donc on a $T(n) \sim n \log n$

C3. Vérifier, en mesurant le temps d'exécution, la justesse de vos calculs précédents.