Université Clermont Auvergne
ISIMA FISA 1
Algorithmique
Année 2023-2024

Florent Foucaud

TD1 - Complexité en temps d'un algorithme
VERSION AVEC SOLUTION

Les exercices commencent à la page 3

1 Rappel de cours : complexité en temps d'un algorithme

Pour un algorithme donné, on souhaite déterminer le nombre d'instructions de base $T(n)$ qui seront effectuées *dans le pire des cas*, pour une entrée de taille n (où $T : \mathbb{N} \to \mathbb{N}$). La taille est le nombre de bits nécessaires pour représenter l'entrée. On considère généralement qu'un nombre (par exemple) a une taille constante, un tableau a donc une taille proportionnelle à sa longueur.

Les instructions de base sont les affectations, les opérations mathématiques, les comparaisons, l'accès à un élément de tableau... et on considère que chacune prend une unité de temps.

En général, on s'intéresse surtout à l'ordre de grandeur de $T(n)$, pour cela on utilise les *notations asymptotiques* suivantes (étant données deux fonctions $f, g : \mathbb{N} \to \mathbb{N}$) :

- $f(n) \in O(g(n))$ si il existe une constante $c \in \mathbb{R}$ et un rang $r \in \mathbb{N}$ tels que, pour tout entier $i \geq r$, on a $f(i) \leq c \cdot g(i)$. Cela veut dire que f ne croît pas de façon plus rapide que g (à un facteur constant près) lorsque le paramètre n est suffisamment grand.
- $f(n) \in \Omega(g(n))$ si il existe une constante $c \in \mathbb{R}$ et un rang $r \in \mathbb{N}$ tels que, pour tout entier $i \geq r$, on a $f(i) \geq c \cdot g(i)$. Cela veut dire que f ne croît pas de façon moins rapide que g (à un facteur constant près) lorsque le paramètre n est suffisamment grand.
- $f(n) \in \Theta(g(n))$ si $f(n) \in O(g(n))$ et $f(n) \in \Omega(g(n))$. Cela signifie que f et g se comportent de la même façon (à facteurs constants près) lorsque le paramètre n est suffisamment grand.

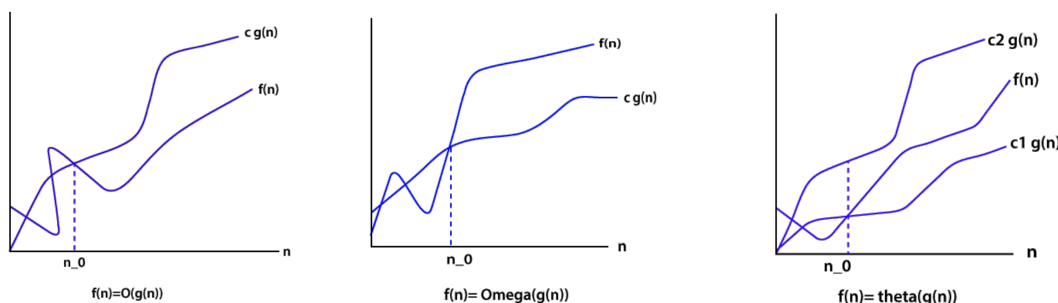


FIGURE 1 – Schéma des trois notations asympototiques

Remarque : si $f(n) \in O(g(n))$, alors $g(n) \in \Omega(f(n))$, et inversement.

Un *problème algorithmique* est défini par le type d'entrée attendu (par exemple : un entier, un tableau d'entiers, etc) et une tâche à effectuer (généralement, une sortie à renvoyer). Plusieurs algorithmes qui résolvent un même problème algorithmique donné peuvent exister, on cherche bien entendu à trouver l'algorithme le plus efficace possible parmi ceux-ci.

Une *classe de complexité* est un ensemble de problèmes algorithmiques pour lesquels le meilleur algorithme possible se comporte de manière similaire (par exemple, en termes de temps de calcul).

Les notations asymptotiques vues ci-dessus permettent de définir des classes de complexité classiques pour les problèmes algorithmiques, en utilisant des fonctions usuelles, par exemple (du meilleur au pire) :

- $T(n) \in \Theta(1)$ (complexité constante)
- $T(n) \in \Theta(\log(\log(n)))$
- $T(n) \in \Theta(\log(n))$ (complexité logarithmique)
- $T(n) \in \Theta(\sqrt{n})$
- $T(n) \in \Theta(n)$ (complexité linéaire)
- $T(n) \in \Theta(n \log(n))$
- $T(n) \in \Theta(n^c)$ pour une constante $c > 1$ (complexité polynômiale)
- $T(n) \in \Theta(c^n)$ pour une constante $c > 0$ (complexité exponentielle)
- $T(n) \in \Theta(n!)$ (qui est dans $O(n^n)$)
- ...

2 Exercices

Exercice 1 (Complexité naïve)**.** La fonction `f` s'exécute en 1 jour de calcul.

1. Est-ce que les programmes 1 et 2 calculent-ils la même chose ?
   Programme 1 :
   ```
   a = f(x) + f(x)
   ```
   Programme 2 :
   ```
   a = 2 * f(x)
   ```
2. Quel est le programme le plus rapide ? Justifier votre réponse.
3. Comparer ce que font les morceaux de programme A, B, C, et D (on suppose que la variable `y` a été instanciée en amont).
   Programme A :
   ```
   if y == f(x) + f(x) :
      z = f(x)
   else :
      z = f(x) + f(x)
   ```
   Programme B :
   ```
   a = 2 * f(x)
   if y == a :
      z = f(x)
   else :
      z = a
   ```
   Programme C :
   ```
   a = f(x)
   if y == 2 * a :
      z = a
   else :
      z = f(x) + f(x)
   ```
   Programme D :
   ```
   a = 2*f(x)
   if y == a :
      z = a/2
   else :
      z = a
   ```
4. Classer ces programmes du plus rapide au plus lent et justifier votre réponse en jours de calcul.

Solution.

1. f(x) + f(x) = 2 * f(x) donc les programmes font la même chose.
2. Le programme 1 met 2 jours alors que le programme 2 met 1 jour !
3. Les programmes font tous la même chose.
4. Voici la complexité des programmes dans le pire des cas :

- Programme A : 4 jours
- Programme B : 2 jours
- Programme C : 3 jours
- Programme D : 1 jours

Exercice 2 (Boucles imbriquées de trucs et bidules)**.**

`truc()` et `bidule()` sont deux fonctions quelconques, sans argument. Déterminer le nombre d'appels à ces fonctions dans les scripts ci-dessous, en fonction de n. On rappelle que `range(n)` renvoie la liste `[0,...,n-1]`.

Programme 1 :

```
for i in range(n):
    truc()
for i in range(n):
    bidule()
```

Programme 2 :

```
for i in range(n):
    truc()
    for j in range(n):
        bidule()
```

Programme 3 :

```
for i in range(n):
    truc()
    for j in range(i):
        bidule()
```

Programme 4 :

```
for i in range(n):
    truc()
    for j in range(n):
        for k in range(n):
            bidule()
```

A) Donner le nombre d'exécutions de `truc()` et `bidule()`, ainsi que la complexité asymptotique de ces scripts, en supposant que les deux fonctions `truc()` et `bidule()` s'exécutent en temps constant $O(1)$. On pourra utiliser la formule suivante lorsque nécessaire :
$\sum_{i=0}^{s} i = \frac{s(s+1)}{2}$

B) On suppose maintenant que `truc()` et `bidule()` prennent un temps n chacune. Que deviennent les complexités ?

Solution.

A)

1. truc et bidule s'exécutent n fois chacune. Complexité $O(n)$.
2. truc s'exécute n fois, bidule n^2 fois. Complexité $O(n^2)$.
3. truc s'exécute n fois, bidule $\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$ fois. Complexité $O(n^2)$.
4. n trucs et n^3 bidule donc $n + n^3 = O(n^3)$.

B) On multiplie tout par n :

1. $2n^2$
2. $n^2 + n^3$
3. $\frac{n^3+n^2}{2}$
4. $n^4 + n^2$

Exercice 3 (Maximum itératif)**.**

Quelle est la complexité en temps $T(n)$ *à l'opération près* de l'algorithme ci-dessous, appelé sur un tableau à $n \geq 1$ éléments ? On rappelle que l'accès $TAB[i]$ à une case du tableau est une opération élémentaire.

```
maxiter(TAB): Recherche du maximum d'un tableau TAB

Entrée : un tableau TAB[1...n] non trié de n entiers

Sortie : le plus grand entier de TAB

  • max = TAB[1]
  • Pour i allant de 2 à n :
      ⋆ Si TAB[i] > max alors :
        max = TAB[i]
  • Retourner max
```

Solution.

```
maxiter(TAB): Recherche du maximum d'un tableau TAB

Entrée : un tableau TAB[1...n] non trié de n entiers

Sortie : le plus grand entier de TAB

  • max = TAB[1] //2 opérations
  • Pour i allant de 2 à n : //3 opérations implicites par tour
    // (incrémentation et réaffectation du compteur et test de sa valeur)
    // - incrémentation au premier tour + un test supplémentaire à la
    sortie de boucle (quand i = n + 1)
      ⋆ Si TAB[i] > max alors : //2 opérations
        max = TAB[i] // 2 opérations
  • Retourner max //1 opération
```

La boucle est effectuée $n-1$ fois dans le pire des cas, donc on a

$$\begin{aligned} T(n) &\leq 2+7(n-1)+1 \\ &= 7n-4 \end{aligned}$$

Exercice 4 (Tri par insertion)**.**

Dans l'algorithme de tri par insertion, on trie le tableau de gauche à droite, en insérant la valeur courante à la bonne place dans la partie déjà triée.

Quelle est la complexité en temps $T(n)$ *à l'opération près* de l'algorithme, appelé sur un tableau à n éléments ?

```
triInsertion(TAB): tri d'un tableau TAB d'entiers par ordre
croissant

Entrée : un tableau TAB[1...n] non trié de n entiers
Sortie : le tableau TAB trié

  • Pour i allant de 2 à n faire :
      ⋆ j = i − 1
      ⋆ valeurInsertion = TAB[i]
      ⋆ Tant que j > 0 et TAB[j] > valeurInsertion faire :
          − TAB[j + 1] = TAB[j]
          − j = j − 1
      ⋆ TAB[j + 1] = valeurInsertion
  • Retourner TAB
```

Solution.

```
triInsertion(TAB): tri d'un tableau TAB d'entiers par ordre
croissant

Entrée : un tableau TAB[1...n] non trié de n entiers
Sortie : le tableau TAB trié

  • Pour i allant de 2 à n faire : // 3 opérations par tour
      ⋆ j = i − 1 //2 opérations
      ⋆ valeurInsertion = TAB[i] //2 opérations
      ⋆ Tant que j > 0 et TAB[j] > valeurInsertion faire : //4
        opérations + 1 pour le dernier test
          − TAB[j + 1] = TAB[j] //4 opérations
          − j = j − 1 //2 opérations
      ⋆ TAB[j + 1] = valeurInsertion //3 opérations
  • Retourner TAB //1 opération
```

La boucle principale est exécutée $n-1$ fois. La boucle imbriquée est exécutée au pire $i-1$ fois

(mais le test de la boucle imbriquée est exécuté une fois supplémentaire sans y rentrer, et comme c'est un “et” il n'exécute que le test $j > 0$; pour cela on ajoute 1 au nombre d'opérations faites par la boucle principale, d'où le “$10 + 1$”).

$$T(n) \leq \sum_{i=2}^{n} \left(10 + 1 + \sum_{j=1}^{i-1} 10 \right) + 1$$

Si on utilise le fait que $i \leq n$ on obtient :

$$\begin{aligned}
T(n) &\leq \sum_{i=2}^{n} \left(11 + \sum_{j=1}^{n-1} 10 \right) + 1 \\
&= (n-1)\left(11 + 10(n-1)\right) + 1 \\
&= 11n - 11 + 10(n-1)^2 + 1 \\
&= 11n - 11 + 10(n^2 - 2n + 1) + 1 \\
&= 10n^2 - 9n
\end{aligned}$$

Cependant, on peut faire une meilleure estimation ci-dessous. (On utilise le fait que $\sum_{i=1}^{s} i = \frac{s(s+1)}{2}$ dans le passage de la 5e à la 6e ligne.)

$$\begin{aligned}
T(n) &\leq \sum_{i=2}^{n} \left(10 + 1 + \sum_{j=1}^{i-1} 10 \right) + 1 \\
&= \sum_{i=2}^{n} \left(11 + 10(i-1)\right) + 1 \\
&= \left(\sum_{i=2}^{n} 11 + \sum_{i=2}^{n} 10(i-1) \right) + 1 \\
&= \left(11(n-1) + 10 \sum_{i=2}^{n} (i-1) \right) + 1 \\
&= \left(11(n-1) + 10 \sum_{i=1}^{n-1} i \right) + 1 \\
&= 11(n-1) + 10\frac{(n-1)n}{2} + 1 \\
&= 11n - 11 + 10\frac{n^2 - n}{2} + 1 \\
&= 5n^2 + 6n - 10
\end{aligned}$$

Exercice 5 (Notations asymptotiques)**.**

1. Montrer que $100n \in O(n^2)$.
2. Montrer que $100n \in \Theta(n)$.
3. Montrer que $n^2 \in \Omega(3n \log_2(n))$.
4. Montrer que $n^2 + n \in \Theta(n^2)$

5. Montrer que $100n \notin \Omega(n^2)$.
6. Montrer que pour tout entier fixé $k > 0$, $n^k \in O(2^n)$.
7. Montrer que $2^n \notin \Theta(3^n)$.

Solution.

1. Montrer que $100n \in O(n^2)$.
   À partir d'un certain rang r, la fonction n^2 va dépasser la fonction $100n$. Pour déterminer ce rang r, on résoud l'équation $100n \leq n^2$. En divisant par n des deux côtés, on obtient que $100 \leq n$, donc $r = 100$.

   Donc, pour $r = 100$ et $c = 1$, on a bien pour tout $i \geq r$, $100i \leq c \cdot i^2$, donc $100n \in O(n^2)$.

   Autre solution : utiliser $r = 0$ et $c = 100$.

   Encore une autre : utiliser $r = 10$ et $c = 10$
2. Montrer que $100n \in \Theta(n)$.
   Pour $r = 0$ et $c = 1$, on a bien pour tout $i \geq r$, $100i \geq c \cdot i$, donc $100n \in \Omega(n)$.
   Pour $r = 0$ et $c = 100$, on a bien pour tout $i \geq r$, $100i \leq c \cdot i$, donc $100n \in O(n)$.
   Donc, $100n \in \Theta(n)$.
3. Montrer que $n^2 \in \Omega(3n \log_2(n))$.
   Pour tout $n \geq 0$, on a $\log_2(n) \leq n$ donc avec $c = 1/3$, et $r = 0$, on a bien pour tout $i \geq r$, que $n^2 \geq c \cdot 3n \log_2(n)$.

   On aurait aussi pu prendre $c = 1$ et trouver le bon rang r en résolvant l'équation $n \geq 3 \log_2(n)$ qui est vrai quand $n \geq 10$ (on peut voir facilement que c'est vrai quand $n = 16$).
4. Montrer que $n^2 + n \in \Theta(n^2)$.
   Pour $r = 0$ et $c = 1$, on a bien pour tout $i \geq r$, $i^2 + i \geq c \cdot i^2$, donc $n^2 + n \in \Omega(n^2)$.
   Pour $r = 0$ et $c = 2$, on a bien pour tout $i \geq r$, $i^2 + i \leq c \cdot i^2 = 2i^2$, donc $n^2 + n \in O(n^2)$.
   Donc, $n^2 + n \in \Theta(n^2)$.
5. Montrer que $100n \notin \Omega(n^2)$.

   Supposons, par l'absurde, que $100n \in \Omega(n^2)$. Donc, dans ce cas, il existe $r \in \mathbb{N}$ et $c \in \mathbb{R}$ tels que pour tout $i \geq r$, $100i \geq c \cdot i^2$. En divisant par i des deux côtés, cela signifie que $100 \geq c \cdot i$ et donc $100/c \geq i$. Intuitivement, cela signifie que i ne peut pas être très grand, ce qui contredit notre hypothèse puisque cela devait être vrai pour tout $i \geq r$. Formellemnt, il suffit de prendre la première valeur de n, disons n_0, qui vérifie $n_0 \geq r$ et $n_0 > 100/c$. Or, dans ce cas, $100/c \geq n_0$ n'est pas vrai. C'est donc une contradiction avec le fait que cela devrait être vrai pour tout $i \geq r$ (donc y compris pour $i = n_0$), et $100n \notin \Omega(n^2)$.
6. Montrer que pour tout entier $k > 0$, $n^k \in O(2^n)$.

   À partir d'un certain rang r, la fonction 2^n va dépasser la fonction n^k. Pour le déterminer, on résoud l'équation $n^k \leq 2^n$. En prenant le logarithme en base 2 des deux côtés, on obtient que $\log_2(n^k) \leq n$, donc $k \log_2(n) \leq n$.

   Ceci est vérifié lorsque n est suffisamment grand par rapport à k. Par exemple, si on prend $n = 2^{2^k}$, on a $\log_2(n) = 2^k$ et on a bien $k \log_2(n) = k2^k \leq 2^{2^k} = n$.

   Autre solution, si on prend $n = 2^k$ on a $\log_2(n) = k$ et on a bien $k \log_2(n) = k^2 \leq 2^k = n$ (sauf quand $k = 3$).

   On peut alors fixer $r = 2^k$ et $c = 2$ (on met $c = 2$ pour accommoder le cas $k = 3$) et on a bien, pour tout $i \geq r$, $i^k \leq c \cdot 2^i$, donc $n^k \in O(2^n)$.
7. Montrer que $2^n \notin \Theta(3^n)$.

   Il est vrai que $2^n \in O(3^n)$ puisque $2^n \leq 3^n$ dès lors que $n \geq 0$. On veut donc montrer que $2^n \notin \Omega(3^n)$.

   Supposons, par l'absurde, que $2^n \in \Omega(3^n)$. Donc, il existe $r \in \mathbb{N}$ et $c \in \mathbb{R}$ tels que pour tout $i \geq r$, $2^i \geq c \cdot 3^i$.

Or, $2^i \geq c \cdot 3^i$ se réécrit comme $\frac{2^i}{3^i} \geq c$ c'est à dire $\left(\frac{2}{3}\right)^i \geq c$. Cependant, lorsque i grandit, $\left(\frac{2}{3}\right)^i$ tend vers 0, donc il y a aura un rang à partir duquel cela ne sera pas vrai, ce qui est une contradiction aussi.
Autre raisonnement : 3^i peut se réécrire comme $2^{\log_2(3)i}$. En prenant le logarithme en base 2 des deux côtés, cela signifie que $i \geq \log_2\left(c2^{\log_2(3)i}\right) = \log_2(c) + \log_2(3)i$, et donc $(1 - \log_2(3))i \geq \log_2(c)$. Or, $(1 - \log_2(3)) < 0$ et donc quand i grandit, le côté gauche diminue, alors que $\log_2(c)$ reste constant. C'est donc une contradiction, et donc $2^n \notin \Omega(3^n)$.

Exercice 6 (Récursivité et Fibonacci)**.**

On considère le programme récursif suivant, permettant d'approximer la taille de la $n$ème génération d'une population de lapins.

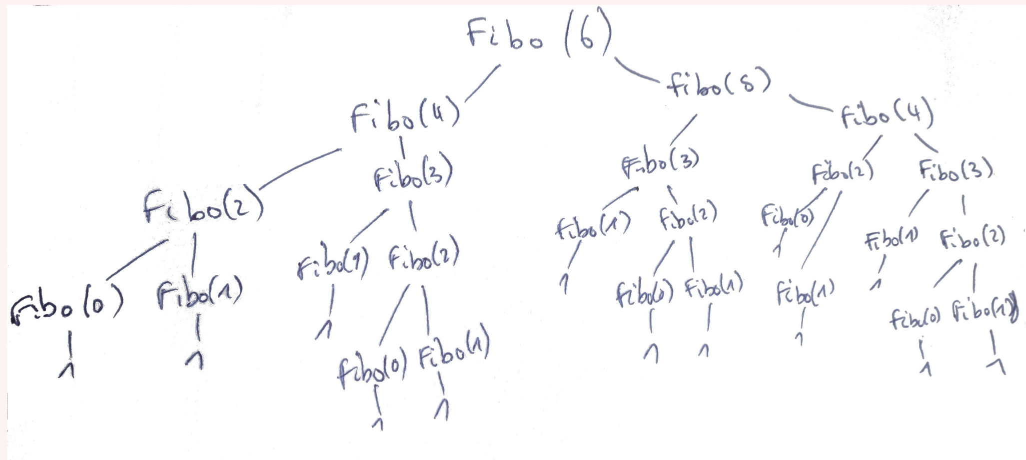
```
Fibo(n):

Entrée : un entier n

Sortie : la nème valeur de la suite de Fibonacci

  • Si n == 0 ou n == 1:
    Retourner 1

  • Sinon:
    Retourner Fibo(n − 1) + Fibo(n − 2)
```

1. Dessiner l'arbre des appels récursifs pour $Fibo(6)$. Intuitivement, quelle est la complexité du programme ?
2. Écrire la complexité en temps $T(n)$ en fonction de $T(n-1)$ et $T(n-2)$, pour $n > 1$, en supposant qu'une opération élémentaire coûte 1.
3. Supposons que $T(n) = x^n$ pour un certain $x > 0$. Remplacer $T(n)$ dans l'équation précédente, et en déduire une équation (qui ressemble à une équation du second degré) ayant x comme variable.
4. Considérer cette équation comme une équation du second degré (en approximant les termes qui sont très petits quand n est grand, à 0), la résoudre,[1] et en déduire une valeur approximative pour $T(n)$.
5. Pourrait-on calculer $Fibo(n)$ de façon plus efficace ? Si oui, comment ?

Solution.

1. On se rend compte que cela "a l'air" exponentiel :

1. Rappel : pour résoudre l'équation $ax^2 + bx + c = 0$, on calcule le discriminant $\Delta = b^2 - 4ac$, si $\Delta < 0$ il n'y a pas de solution, si $\Delta = 0$ il y a une solution unique à $\frac{-b}{2a}$, sinon il y a deux solutions : $\frac{-b-\sqrt{\Delta}}{2a}$ et $\frac{-b+\sqrt{\Delta}}{2a}$.



2. $T(n) = T(n-1) + T(n-2) + 2$ (1 pour le test sur n, et 1 pour l'addition)
3. Cela donne (en divisant par x^{n-2} entre les lignes 1 et 2 :

$$x^n = x^{n-1} + x^{n-2} + 2$$

$$x^2 = x + 1 + \frac{2}{x^{n-2}}$$

$$x^2 - x - 1 = \frac{2}{x^{n-2}}$$

4. On peut négliger le terme $\frac{2}{x^{n-2}}$ qui est proche de 0 quand n est grand. On peut alors résoudre l'équation du second degré $x^2 - x - 1 = 0$ c'est à dire $ax^2 + bx + c = 0$ où $a = 1$, $b = -1$ et $c = -1$. Le discriminant $\Delta = b^2 - 4ac = (-1)^2 - 4 \times 1 \times (-1) = 5 > 0$ et on obtient donc deux solutions :
$x_1 = \frac{-b-\sqrt{\Delta}}{2a} = \frac{1+\sqrt{5}}{2}$ et $x_2 = \frac{-b+\sqrt{\Delta}}{2a} = \frac{1-\sqrt{5}}{2}$.
Dans notre cas, on cherche $x > 0$. Or, $x_1 < 0$, donc on prend $x = x_2$ et on peut approximer $T(n)$ par $\left(\frac{1+\sqrt{5}}{2}\right)^n$. C'est bien une fonction exponentielle.
Remarque : $x_2 \approx 1,618033$ est historiquement (probablement connu depuis l'Antiquité) appelé le *nombre d'or*.
5. Oui, de façon itérative, en gardant toujours les deux dernières valeurs en mémoire, on peut le caculer en temps linéaire. Par exemple :

```
Fibo(n):

Entrée : un entier n

Sortie : la nème valeur de la suite de Fibonacci

  • Si n = 0 ou n = 1:
    Retourner 1

  • Sinon:
      - Fprecprec = 1
      - Fprec = 1
      - i = 2
      - Tant que i < n faire :
          ⋆ Fnew = Fprecprec + Fprec
          ⋆ Fprecprec = Fprec
          ⋆ Fprec = Fnew
          ⋆ i = i + 1
      - Retourner Fprecprec + Fprec
```

Exercice 7 (Sous-tableaux)**.**

Quelle est la complexité asymptotique en temps $T(n)$ de l'algorithme ci-dessous, appelé sur un tableau à n éléments ?

```
soustableautriemax(TAB): Recherche du plus grand sous-tableau
d'éléments croissants d'un tableau TAB

Entrée : un tableau TAB[1...n] non trié de n entiers

Sortie : la taille d'un plus grand sous-tableau (pas forcément
     consécutif) de TAB dont tous les éléments sont rangés dans
     l'ordre croissant

  • max = 0
  • Pour chaque sous-tableau S de TAB :
      ⋆ test = Vrai
      ⋆ Pour i allant de 1 à |S| − 1:
        si S[i] > S[i + 1]:
        test = Faux
      ⋆ si test == Vrai et |S| > max:
        max = |S|
  • Retourner max
```

Solution.

Il y a 2^n sous-tableaux de TAB, donc la boucle principale est exécutée 2^n fois. Pour chaque sous-tableau S, on fait $1 + 3|S|$ plus 2 ou 3 opérations. Pour faire simple, on approxime avec $3n + 4$, ce qui donne $(3n + 4)2^n \in \Theta(n2^n)$.

Exercice 8 (Tri par sélection)**.**

Dans l'algorithme de tri par sélection (aussi appelé tri par minimum), on trie le tableau de gauche à droite, en allant chercher la valeur minimale de la partie non triée, et en la déplaçant à la suite de la partie déjà triée.

Quelle est la complexité en temps $T(n)$ *à l'opération près* de l'algorithme, appelé sur un tableau à n éléments ?

```
triSelection(TAB): tri d'un tableau TAB d'entiers par ordre
croissant

Entrée : un tableau TAB[1...n] non trié de n entiers

Sortie : le tableau TAB trié

  • Pour i allant de 1 à n − 1 faire :
      ⋆ indiceMin = i
      ⋆ valeurMin = TAB[i]
      ⋆ Pour j allant de i + 1 à n faire :
          – Si TAB[j] < valeurMin alors :
              indiceMin = j
              valeurMin = TAB[j]
      ⋆ TAB[indiceMin] = TAB[i]
      ⋆ TAB[i] = valeurMin
  • Retourner TAB
```

Solution.

```
triSelection(TAB): tri d'un tableau TAB d'entiers par ordre
croissant

Entrée : un tableau TAB[1...n] non trié de n entiers

Sortie : le tableau TAB trié

  • Pour i allant de 1 à n − 1 faire : //4 opérations par tour (test,
    n − 1, incrémentation et réaffectation de i)
    // -1 car pas d'incrémentation le premier tour mais +1 pour le
    dernier test après le dernier tour
      ⋆ indiceMin = i //1 opération
      ⋆ valeurMin = TAB[i] // 2 opérations
      ⋆ Pour j allant de i + 1 à n faire : //4 opérations par tour,
        comme précédemment
          – Si TAB[j] < valeurMin alors : //2 opérations
              indiceMin = j //1 opération
              valeurMin = TAB[j] //2 opérations
      ⋆ TAB[indiceMin] = TAB[i] //3 opérations
      ⋆ TAB[i] = valeurMin //2 opérations
  • Retourner TAB //1 opération
```

La boucle principale est exécutée $n-1$ fois. La boucle imbriquée est exécutée au pire $n-i$ fois (mais le test de la boucle imbriquée est exécuté une fois supplémentaire sans y rentrer).

$$T(n) \leq \sum_{i=1}^{n-1} \left(4+1+2+3+2+\sum_{j=i+1}^{n} (4+2+1+2) \right) +1$$

Si on utilise le fait que $n-i \leq n-1$ on obtient :

$$
\begin{aligned}
T(n) &\leq \sum_{i=1}^{n-1}\left(12+\sum_{j=i+1}^{n} 9\right)+1 \\
&= \sum_{i=1}^{n-1}\left(12+\sum_{j=i+1}^{n} 9\right)+1 \\
&= \sum_{i=1}^{n-1}\left(12+\sum_{j=1}^{n-i} 9\right)+1 \\
&= (n-1)\left(12+9(n-1)\right)+1 \\
&= 12n-12+9(n-1)^2+1 \\
&= 12n-11+9(n^2-2n+1) \\
&= 9n^2-6n-2
\end{aligned}
$$

Cependant, on peut faire une meilleure estimation ci-dessous. (On utilise le fait que $\sum_{i=1}^{s} i = \frac{s(s+1)}{2}$ dans le passage de la 4e à la 5e ligne.)

$$
\begin{aligned}
T(n) &\leq \sum_{i=1}^{n-1}\left(4+1+2+3+2+\sum_{j=i+1}^{n}(4+2+1+2)\right)+1 \\
&= \sum_{i=1}^{n-1}\left(12+\sum_{j=1}^{n-i} 9\right)+1 \\
&= 1+12(n-1)+9\sum_{i=1}^{n-1}(n-i) \\
&= 1+12(n-1)+9\sum_{i=1}^{n-1} i \\
&= 1+12(n-1)+9\frac{n(n-1)}{2} \\
&= 1+12n-12+9\frac{n^2-n}{2} \\
&= \frac{9}{2}n^2+\frac{15}{2}n-11
\end{aligned}
$$

Exercice 9 (Tri à bulles)**.**

Dans le tri à bulles, on fait remonter vers la fin du tableau (comme une bulle dans un liquide) la valeur la plus grande par échanges successifs des valeurs consécutives du tableau.

Exprimer le plus précisément possible, la complexité en temps $T(n)$ de l'algorithme du tri à bulles, où n est la taille du tableau.

```
triBulles(TAB): tri d'un tableau TAB d'entiers par ordre croissant

Entrée : un tableau TAB[1...n] non trié de n entiers
Sortie : le tableau TAB trié

  • Pour i allant de n à 1 faire :
      ⋆ Pour j allant de 1 à i − 1 faire :
          − Si TAB[j + 1] < TAB[j] alors :
              x = TAB[j]
              TAB[j] = TAB[j + 1]
              TAB[j + 1] = x
  • Retourner TAB
```

Solution.

```
triBulles(TAB): tri d'un tableau TAB d'entiers par ordre croissant

Entrée : un tableau TAB[1...n] non trié de n entiers

Sortie : le tableau TAB trié

  • Pour i allant de n à 1 faire : // 3 opérations par tour
    (incrémentation et affectation de i, test) - pas d'incrémentation
    au premier tour, mais un test en plus après le dernier tour

      ⋆ Pour j allant de 1 à i−1 faire : // 4 opérations par tour

          – Si TAB[j+1] < TAB[j] alors : // 4 opérations
            x = TAB[j] // 2 opérations
            TAB[j] = TAB[j+1] // 4 opérations
            TAB[j+1] = x // 3 opérations

  • Retourner TAB // 1 opération
```

La boucle principale est exécutée n fois. La boucle imbriquée est exécutée $i-1$ fois (mais le test de la boucle imbriquée est exécuté une fois supplémentaire sans y rentrer). Comme $i \leq n$ on a :

$$\begin{aligned}
T(n) &\leq \sum_{i=1}^{n}\left(3+\sum_{j=1}^{i-1}(4+4+2+4+3)\right)+1 \\
&\leq \sum_{i=1}^{n}\left(3+\sum_{j=1}^{i-1}17\right)+1 \\
&= n\left(3+17(n-1)\right)+1 \\
&= 3n+17n(n-1)+1 \\
&= 17n^2-14n+1
\end{aligned}$$

Cependant, on peut faire une meilleure estimation ci-dessous. (On utilise le fait que $\sum_{i=1}^{s} i = \frac{s(s+1)}{2}$ dans le passage de la 4e à la 5e ligne.)

$$\begin{aligned}
T(n) &\leq \sum_{i=1}^{n}\left(3+\sum_{j=1}^{i-1}17\right)+1 \\
&= \leq \sum_{i=1}^{n}\left(3+\sum_{j=1}^{i-1}17\right)+1 \\
&= 1+3n+17\sum_{i=1}^{n}(i-1) \\
&= 1+3n+17\left(\left(\sum_{i=1}^{n} i\right)-n\right) \\
&= 1+3n+17\left(\frac{n(n+1)}{2}-\frac{2n}{2}\right) \\
&= 1+3n+17\left(\frac{n^2-n}{2}\right) \\
&= \frac{17n^2}{2}-\frac{11n}{2}+1
\end{aligned}$$