Université Clermont Auvergne
ISIMA FISA 1
Algorithmique
Année 2023-2024

Florent Foucaud

TD1 - Complexité en temps d'un algorithme

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.

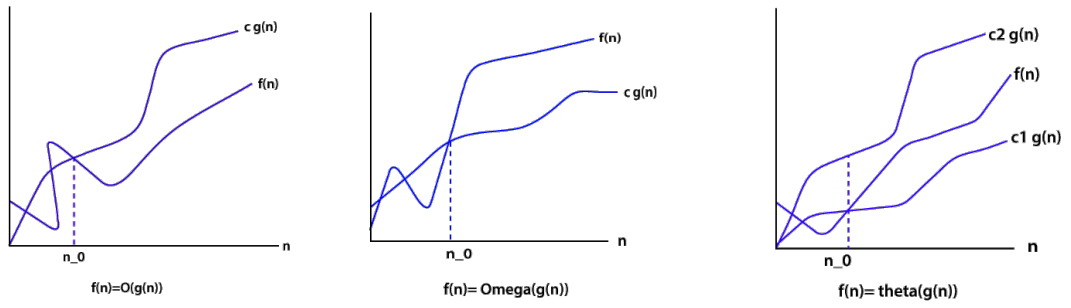

FIGURE 1 – Schéma des trois notations asympototiques

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.

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 ?

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`

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
```

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 $2^n \notin \Theta(3^n)$.

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 ?

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`

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 ?

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 $\frac{-b+\sqrt{\Delta}}{2a}$.

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

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