Université Clermont Auvergne
ISIMA FISA 1
Algorithmique
Année 2025-2026

Florent Foucaud

TD1 - Complexité en temps d'un algorithme

VERSION AVEC SOLUTION

Les exercices commencent à la page 2

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/écriture d'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 $c \in \mathbb{R}^+$ $(c > 0)$ et un rang $n_0 \in \mathbb{N}$ tels que, pour tout entier $i \geq n_0$, 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 $c \in \mathbb{R}^+$ $(c > 0)$ et un rang $n_0 \in \mathbb{N}$ tels que, pour tout entier $i \geq n_0$, 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.

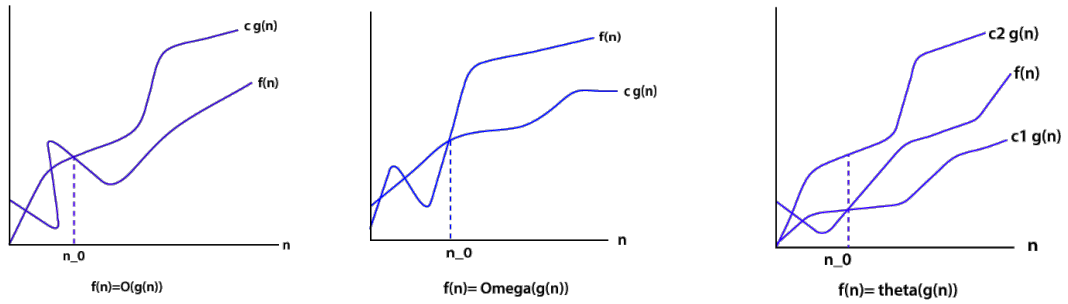


FIGURE 1 – Schéma des trois notations asympototiques

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)
- $T(n) \in \Theta(n \log(n))$
- $T(n) \in \Theta(n^2)$ (complexité quadratique)
- $T(n) \in \Theta(n^c)$ pour une constante $c \geq 1$ (complexité polynomiale)
- $T(n) \in \Theta(c^n)$ pour une constante $c > 1$ (complexité exponentielle)
- $T(n) \in \Theta(n!)$ (qui est dans $O(n^n)$)
- ...

2 Premières estimations de complexité

Exercice 1 (Complexité naïve)**.**

La fonction `f` s'exécute en 1 jour de calcul.

1. Est-ce que les programme 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. Est-ce que les programme A, B, C, et D calculent la même chose ?

Programme A :
```
Si y == f(x) + f(x) :
   Alors z = f(x)
Sinon z = f(x) + f(x)
```

Programme B :
```
a = 2 * f(x)
Si y == a :
   Alors z = f(x)
Sinon z = a
```

Programme C :
```
a = f(x)
Si y == 2 * a  :
   Alors z = a
Sinon z = f(x) + f(x)
```

Programme D :
```
a = 2*f(x)
Si y == a :
   Alors z = a/2
Sinon 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.

Programme 1 :

```
Pour i allant de 1 à n:
    truc()
Pour i allant de 1 à n:
    bidule()
```

Programme 2 :

```
Pour i allant de 1 à n:
    truc()
    Pour j allant de 1 à n:
        bidule()
```

Programme 3 :

```
Pour i allant de 1 à n:
    truc()
    Pour j allant de 1 à i:
        bidule()
```

Programme 4 :

```
Pour i allant de 1 à n:
    truc()
    Pour j allant de 1 à n:
        Pour k allant de 1 à n:
            bidule()
```

A) Donner le nombre d'exécutions de `truc()` et `bidule()`. On suppose que les deux fonctions `truc()` et `bidule()` s'exécutent chacune en temps constant : 1 — donner (sans justification) la complexité *asymptotique* de ces scripts avec la notation grand O, si les deux fonctions s'exécutent chacun en temps $O(1)$.

On pourra utiliser la formule suivante lorsque nécessaire :

$$\sum_{i=0}^{s} i = \frac{s(s+1)}{2}$$

B) On suppose maintenant que `truc()` prend un temps n et `bidule()` prend un temps n^2. Que deviennent les complexités ?

Solution.

A)

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

B) On obtient :

1. $n^2 + n^3 \in O(n^3)$
2. $n(n + n^2) = n^2 + n^4 \in O(n^4)$
3. $n^2 + \frac{n^4+n^3}{2} \in O(n^4)$
4. $n^2 + n^5 \in O(n^5)$

Exercice 3 (Maximum itératif)**.**

Quelle est la complexité en temps $T(n)$ *à l'instruction élémentaire 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 instruction élémentaire, une comparaison, une affectation et une addition sont aussi des instructions élémentaires.

```
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]
  • i = 2
  • Tant que i ≤ n :
      ⋆ Si TAB[i] > max alors :
        max = TAB[i]
      ⋆ i = i + 1
  • 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 instructions élémentaires (affectation et accès tableau)
  • i = 2                                                    // 1 instruction
  • Tant que i ≤ n : // 1 instruction par tour de boucle, plus une instruction à la sortie de boucle (le dernier test est négatif et on sort)
      ⋆ Si TAB[i] > max alors :                              // 2 instructions
        max = TAB[i]                                         // 2 instructions
      ⋆ i = i + 1                                            // 2 instructions
  • Retourner max                                            // 1 instruction
```

Le retourne compte 2, car il y a le retourne et le test de la sortie de la boucle.

La boucle est effectuée $n-1$ fois dans le pire des cas (car i va de 2 à n et on sort dès que $i = n+1$), donc on a

$$\begin{aligned} T(n) &\leq 3 + (n-1)(1+2+2+2) + 1 + 1 \\ &= 7(n-1) + 5 \\ &= 7n - 2 \end{aligned}$$

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'instruction élémentaire près* de l'algorithme, appelé sur un tableau à n éléments ?

```
triInsertion(TAB): tri par ordre croissant du tableau TAB d'entiers
Entrée : un tableau TAB[1...n] non trié de n entiers
Sortie : le tableau TAB trié
  • i = 2
  • Tant que i ≤ 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
      ⋆ i = i + 1
  • 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
  • i = 2                                                  //1 instruction
  • Tant que i ≤ n faire :  //1 instruction par tour de boucle, plus un
    pour le test qui fait sortir de la boucle
      ⋆ j = i − 1                                          //2 instructions
      ⋆ valeurInsertion = TAB[i]                           //2 instructions
      ⋆ Tant que j > 0 et TAB[j] > valeurInsertion faire : //4
        instructions par tour de boucle, plus 4 pour le test qui fait
        sortir de la boucle
          ◦ TAB[j + 1] = TAB[j]                            //4 instructions
          ◦ j = j − 1                                      //2 instructions
      ⋆ TAB[j + 1] = valeurInsertion                       //3 instructions
      ⋆ i = i + 1                                          //2 instructions
  • Retourner TAB                                          //2 instructions
```

La boucle principale est exécutée $n-1$ fois et contient 10 instructions élémentaires (sans compter la boucle imbriquée). Au $i$ème tour, la boucle imbriquée est exécutée au pire $i-1$ fois et contient 10 instructions élémentaires (mais le test de la boucle imbriquée est exécuté une fois supplémentaire sans

y rentrer ; pour cela on ajoute 4 au nombre d'instructions faites par la boucle principale). En sortant de la boucle principale on doit compter le test d esortie ainsi que le return. On a donc :

$$\begin{aligned} T(n) &\leq 1 + \sum_{i=2}^{n} \left(1 + 2 + 2 + \left(\sum_{j=1}^{i-1} 10 \right) + 4 + 3 + 2 \right) + 1 + 1 \\ &= 3 + \sum_{i=2}^{n} \left(14 + \sum_{j=1}^{i-1} 10 \right) \\ &= 3 + \sum_{i=2}^{n} \left(14 + 10(i-1) \right) \end{aligned}$$

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

$$\begin{aligned} T(n) &\leq 3 + \sum_{i=2}^{n} \left(14 + 10(i-1) \right) \\ &= 3 + \sum_{i=2}^{n} \left(14 + 10(n-1) \right) \\ &= 3 + (n-1)(14 + 10(n-1)) \\ &= 3 + 14n - 14 + 10(n-1)^2 \\ &= 3 + 14n - 14 + 10(n^2 - 2n + 1) \\ &= 10n^2 - 6n - 1 \end{aligned}$$

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

$$\begin{aligned} T(n) &\leq 3 + \sum_{i=2}^{n} \left(14 + 10(i-1) \right) \\ &= 3 + \sum_{i=2}^{n} (4 + 10i) \\ &= 3 + 4(n-1) + 10 \sum_{i=2}^{n} i \\ &= 3 + 4(n-1) + 10 \sum_{i=1}^{n-1} (i+1) \\ &= 3 + 4(n-1) + 10 \left(\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 \right) \\ &= 3 + 4(n-1) + 10 \frac{(n-1)n}{2} + 10(n-1) \\ &= 3 + 14(n-1) + 10 \frac{n^2 - n}{2} \\ &= 3 + 14n - 14 + 5n^2 - 5n \\ &= 5n^2 + 9n - 11 \end{aligned}$$

3 Notations asymptotiques, classes de complexité

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 $2n^2+n+10 \in \Theta(n^2)$. Plus généralement, comment déterminer la classe de complexité $\Theta(...)$ d'une somme de fonctions, si on connait la complexité de ses termes ?
5. Montrer que $4n - 3\sqrt{n} \in \Theta(n)$.
6. Montrer que $1 + \frac{1}{n} \in \Theta(1)$.
7. Montrer que si $f(n) \in O(g(n))$, alors $g(n) \in \Omega(f(n))$.
8. Montrer que $100n \notin \Omega(n^2)$. *Indice : supposer par l'absurde que* $100n \in \Omega(n^2)$ *et en déduire une contradiction.*

Solution.

Attention, dans la solution, le rang n_0 de la définition est appelé r.

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 $n_0 = 100$ 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 $r = 1$ et $c = 100$. Comme $100n \leq 100n^2 = cn^2$ dès que $n \geq 1$, on a bien aussi $100n \in O(n^2)$.

   (Encore une autre solution : 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, $100n \in \Theta(n)$ puisque $100n \in \Omega(n)$ et $100n \in O(n)$.
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 $n^2 \geq n \log_2(n) = c \cdot 3n \log_2(n)$.
   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 $2n^2 + n + 10 \in \Theta(n^2)$.
   Pour $r = 0$ et $c = 1$, on a bien pour tout $i \geq r$, $2i^2 + i + 10 \geq c \cdot i^2$, donc $2n^2 + n + 10 \in \Omega(n^2)$.

   Pour $r = 1$ et $c = 13$, on a bien pour tout $i \geq r$: $2i^2 \leq 2i^2$, $i \leq i^2$, et $10 \leq 10i^2$. Donc, $2i^2 + i + 10 \leq 13i^2 = c \cdot i^2$, donc $2n^2 + n + 10 \in O(n^2)$.

   Donc, $2n^2 + n + 10 \in \Theta(n^2)$ puisque $2n^2 + n + 10 \in \Omega(n^2)$ et $2n^2 + n + 10 \in O(n^2)$.
   Plus généralement, comment déterminer la classe de complexité $\Theta(...)$ d'une somme de fonctions, est donnée par le maximum des classes de complexité de ses termes. Ainsi, si $f_1(n) \in \Theta(g_1(n))$ et $f_2(n) \in \Theta(g_2(n))$, alors $f_1(n) + f_2(n) \in \Theta(\max(g_1(n), g_2(n)))$. Cela peut se montrer rigoureusement (voir exercice supplémentaire).

5. Montrer que $4n - 3\sqrt{n} \in \Theta(n)$.
   On a $4n - 3\sqrt{n} \leq 4n$ donc $4n - 3\sqrt{n} \in O(n)$ en prenant $c = 4$ et $r = 0$.
   On a $4n - 3\sqrt{n} \geq n$ car $3\sqrt{n} \leq 3n$, donc $4n - 3\sqrt{n} \in \Omega(n)$ en prenant $c = 1$ et $r = 0$.

   Donc $4n - 3\sqrt{n} \in \Theta(n)$.
6. Montrer que $1 + \frac{1}{n} \in \Theta(1)$.
   On a $1 + \frac{1}{n} \leq 2$ donc $1 + \frac{1}{n} \in O(1)$ avec $c = 2$ et $r = 0$.

   On a $1 + \frac{1}{n} \geq 1$ donc $1 + \frac{1}{n} \in \Omega(1)$ avec $c = 1$ et $r = 0$.

   Donc $1 + \frac{1}{n} \in \Theta(1)$.
7. Montrer que si $f(n) \in O(g(n))$, alors $g(n) \in \Omega(f(n))$.

   Par définition, il existe c, r tel que pour tout $n \geq r$, $f(n) \leq c \cdot g(n)$. Donc, $g(n) \geq \frac{f(n)}{c}$ pour tout $n \geq r$, et donc $g(n) \in \Omega(f(n))$ en prenant $r' = r$ et $c' = \frac{1}{c}$.
8. 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 $100n \notin \Omega(n^2)$.

4 Complexité avancée

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

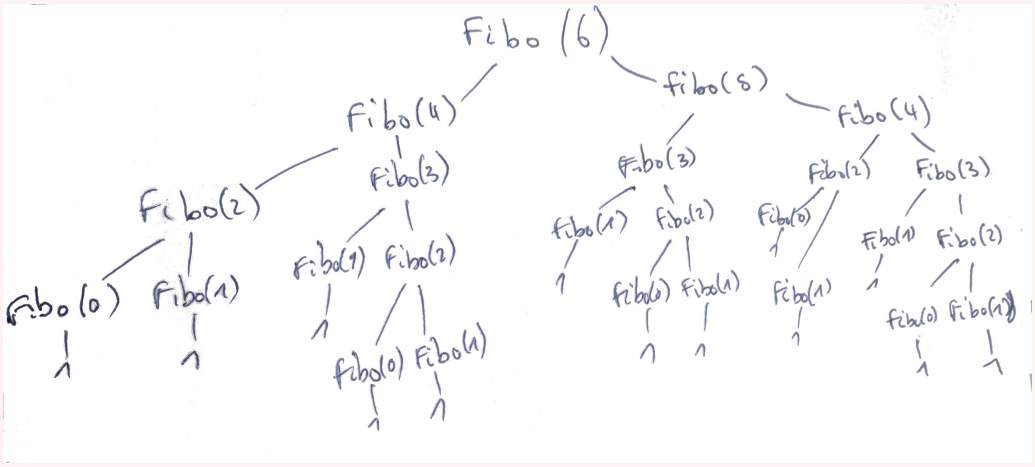
On considère le programme récursif suivant, permettant d'approximer la taille de la $n$ème génération d'une population de lapins.

```
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 instruction 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 :



2. $T(n) = T(n-1) + T(n-2) + 7$ (1 pour les 2 tests, 1 pour le "ou" booléen, 1 pour les 3 additions/soustractions, 1 pour le return)

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

3. Cela donne (en divisant par x^{n-2} entre les lignes 1 et 2 :

$$\begin{aligned} x^n &= x^{n-1} + x^{n-2} + 7 \\ x^2 &= x + 1 + \frac{7}{x^{n-2}} \\ x^2 - x - 1 &= \frac{7}{x^{n-2}} \end{aligned}$$

4. On peut négliger le terme $\frac{7}{x^{n-2}}$ 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
```

5 Pour aller plus loin

Exercice 7 (Notations asymptotiques, suite)**.**

1. Montrer que pour tout entier fixé $k > 0$, $n^k \in O(2^n)$.
2. Montrer que $2^n \notin \Theta(3^n)$.

Solution.

1. 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 $n^k \in O(2^n)$.
2. 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 8 (Complexités de sommes et produits de fonctions)**.**

A. Classer les fonctions suivantes en fonction des types de complexité proposés :
   $\log(n) + 2n^2 - 5$ $\quad 3n^4 + 2n$ $\quad 33\log(n) + 56$ $\quad 8^n + n^2$ $\quad 348n - \sqrt{n} + \log(n)$ $\quad 1 + \frac{1}{n^2}$
   1. temps constant $\Theta(1)$
   2. temps logarithmique $\Theta(\log n)$
   3. temps linéaire $\Theta(n)$
   4. temps quadratique $\Theta(n^2)$
   5. temps polynomial $\Theta(n^c)$ pour une constante $c \geq 1$
   6. temps exponentiel $\Theta(c^n)$ pour une constante $c > 1$

B. De manière générale, si on sait que $f_1 \in \Theta(g_1)$ et $f_2 \in \Theta(g_2)$, comment détermine-t-on la complexité asymptotique de la somme $f_1 + f_2$? Montrez-le rigoureusement.

C. Même question pour le produit $f_1 \times f_2$.

Solution.

A.
1. temps constant $\Theta(1)$ — $1 + \frac{1}{n^2}$
2. temps logarithmique $\Theta(\log n)$ — $33\log(n) + 56$
3. temps linéaire $\Theta(n)$ — $348n - \sqrt{n} + \log(n)$
4. temps quadratique $\Theta(n^2)$ — $\log(n) + 2n^2 - 5$
5. temps polynomial $\Theta(n^c)$ pour une constante $c \geq 1$ — $3n^4 + 2n$

6. temps exponentiel $\Theta(c^n)$ pour une constante $c > 1$ $8^n + n^2$

B. On va montrer que $f_1 + f_2 \in \Theta(\max(g_1, g_2))$. En effet, par définition il existe :

- un rang r_1 et une constante c_1 tel que pour tout $i \geq r_1$, $f_1(i) \leq c_1 \cdot g_1(i)$ $(f_1 \in O(g_1))$
- un rang r'_1 et une constante c'_1 tel que pour tout $i \geq r'_1$, $f_1(i) \geq c'_1 \cdot g_1(i)$ $(f_1 \in \Omega(g_1))$
- un rang r_2 et une constante c_2 tel que pour tout $i \geq r_2$, $f_2(i) \leq c_2 \cdot g_2(i)$ $(f_2 \in O(g_2))$
- un rang r'_2 et une constante c'_2 tel que pour tout $i \geq r'_2$, $f_2(i) \geq c'_2 \cdot g_2(i)$ $(f_2 \in \Omega(g_2))$

Soit $r = \max(r_1, r_2)$, $c_{max} = \max(c_1, c_2)$ et $c = 2c_{max}$. Pour tout $i \geq r$, on a :

$$\begin{aligned} f_1(i) + f_2(i) &\leq c_1 \cdot g_1(i) + c_2 \cdot g_2(i) \\ &\leq c_{max} \cdot g_1(i) + c_{max} \cdot g_2(i) \\ &\leq c_{max}(g_1(i) + g_2(i)) \\ &\leq 2c_{max} \max(g_1(i), g_2(i)) \end{aligned}$$

Et donc on a bien $f_1 + f_2 \in O(\max(g_1, g_2))$.

Ensuite, soit $r' = \max(r'_1, r'_2)$, et $c = c_{min} = \min(c'_1, c'_2)$. Pour tout $i \geq r'$, on a :

$$\begin{aligned} f_1(i) + f_2(i) &\geq c'_1 \cdot g_1(i) + c'_2 \cdot g_2(i) \\ &\geq c_{min} \cdot g_1(i) + c_{min} \cdot g_2(i) \\ &\geq c_{min}(g_1(i) + g_2(i)) \\ &\geq c_{min} \max(g_1(i), g_2(i)) \end{aligned}$$

Et donc on a bien $f_1 + f_2 \in \Omega(\max(g_1, g_2))$.

C. On va montrer que $f_1 \times f_2 \in \Theta(g_1 \times g_2)$. En effet, de nouveau, par définition il existe :

- un rang r_1 et une constante c_1 tel que pour tout $i \geq r_1$, $f_1(i) \leq c_1 \cdot g_1(i)$ $(f_1 \in O(g_1))$
- un rang r'_1 et une constante c'_1 tel que pour tout $i \geq r'_1$, $f_1(i) \geq c'_1 \cdot g_1(i)$ $(f_1 \in \Omega(g_1))$
- un rang r_2 et une constante c_2 tel que pour tout $i \geq r_2$, $f_2(i) \leq c_2 \cdot g_2(i)$ $(f_2 \in O(g_2))$
- un rang r'_2 et une constante c'_2 tel que pour tout $i \geq r'_2$, $f_2(i) \geq c'_2 \cdot g_2(i)$ $(f_2 \in \Omega(g_2))$

Soit $r = \max(r_1, r_2)$, $c = c_1 \cdot c_2$. Pour tout $i \geq r$, on a :

$$\begin{aligned} f_1(i) \times f_2(i) &\leq c_1 \cdot g_1(i) \cdot c_2 \cdot g_2(i) \\ &= (c_1 \cdot c_2) \cdot (g_1(i) \cdot g_2(i)) \\ &= c \cdot (g_1(i) \cdot g_2(i)) \end{aligned}$$

Et donc on a bien $f_1 \times f_2 \in O(g_1 \times g_2)$.

Ensuite, soit $r' = \max(r'_1, r'_2)$, et $c' = c'_1 \cdot c'_2$. Pour tout $i \geq r'$, on a :

$$\begin{aligned} f_1(i) \times f_2(i) &\geq c'_1 \cdot g_1(i) \cdot c'_2 \cdot g_2(i) \\ &\geq (c'_1 \cdot c'_2) \cdot (g_1(i) \cdot g_2(i)) \\ &\geq c' \cdot (g_1(i) \cdot g_2(i)) \end{aligned}$$

Et donc on a bien $f_1 \times f_2 \in \Omega(g_1 \times g_2)$.