Université Clermont Auvergne
ISIMA FISA 1
Algorithmique
Année 2024-2025

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 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 $c \in \mathbb{R}^+$ $(c > 0)$ 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 $c \in \mathbb{R}^+$ $(c > 0)$ 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.

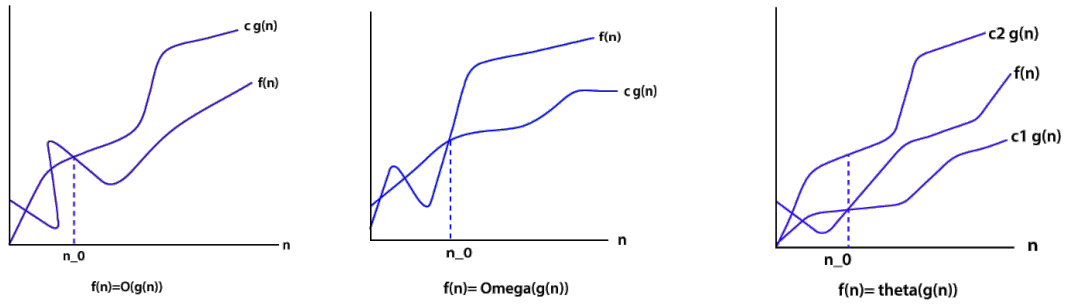

FIGURE 1 – Schéma des trois notations asympototiques

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)
- $T(n) \in \Theta(n \log(n))$
- $T(n) \in \Theta(n^2)$ (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 Exercices : 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 font-ils 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()`, ainsi que la complexité asymptotique de ces scripts (notation grand O), 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 $\sum_{i=1}^{n} n^2 = n^2 \sum_{i=1}^{n} 1 = n^2 * n = n^3$ 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, une comparaison, une affectation et une addition sont aussi des opérations 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]
  • 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)
    // moins 1 incrémentation au premier tour
    // plus une série de incrémentation/réaffectation/test supplémentaire
    // à la sortie de boucle (quand i passe à 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

$$\begin{aligned} T(n) &\leq 2 + (3+4)(n-1) - 1 + 3 + 1 \\ &= 7n - 2 \end{aligned}$$

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'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
    // moins 1 incrémentation au premier tour
    // plus une série de incrémentation/réaffectation/test
    supplémentaire
    // à la sortie de boucle (quand i passe à n + 1)
      ⋆ 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 "$10+1$").

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

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

$$\begin{aligned} T(n) &\leq \sum_{i=2}^{n} \left(13 + \sum_{j=1}^{n-1} 10 \right) + 1 \\ &= (n-1)\left(13 + 10(n-1)\right) + 1 \\ &= 13n - 13 + 10(n-1)^2 + 1 \\ &= 13n - 13 + 10(n^2 - 2n + 1) + 1 \\ &= 10n^2 - 7n \end{aligned}$$

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.)

$$\begin{aligned} T(n) &\leq \sum_{i=2}^{n} \left(10 + 2 + 1 + \sum_{j=1}^{i-1} 10 \right) + 1 \\ &= \sum_{i=2}^{n} (13 + 10(i-1)) + 1 \\ &= \left(\sum_{i=2}^{n} 13 + \sum_{i=2}^{n} 10(i-1) \right) + 1 \\ &= \left(13(n-1) + 10 \sum_{i=2}^{n} (i-1) \right) + 1 \\ &= \left(13(n-1) + 10 \sum_{i=1}^{n-1} i \right) + 1 \\ &= 13(n-1) + 10\frac{(n-1)n}{2} + 1 \\ &= 13n - 13 + 10\frac{n^2 - n}{2} + 1 \\ &= 5n^2 + 8n - 12 \end{aligned}$$

Exercice 5 (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 :

$$\begin{aligned}
T(n) &\leq \sum_{i=1}^{n} \left(3 + \sum_{j=1}^{i-1} (4+4+2+4+3) + 4 - 1 \right) + 3 - 1 + 1 \\
&\leq \sum_{i=1}^{n} \left(6 + \sum_{j=1}^{i-1} 17 \right) + 3 \\
&= n\,(6 + 17(n-1)) + 3 \\
&= 6n + 17n(n-1) + 3 \\
&= 17n^2 - 11n + 3
\end{aligned}$$

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.)

$$\begin{aligned}
T(n) &\leq \sum_{i=1}^{n} \left(6 + \sum_{j=1}^{i-1} 17 \right) + 3 \\
&= \leq \sum_{i=1}^{n} \left(6 + \sum_{j=1}^{i-1} 17 \right) + 3 \\
&= 3 + 6n + 17 \sum_{i=1}^{n} (i-1) \\
&= 3 + 6n + 17 \left(\left(\sum_{i=1}^{n} i \right) - n \right) \\
&= 3 + 6n + 17 \left(\frac{n(n+1)}{2} - \frac{2n}{2} \right) \\
&= 3 + 6n + 17 \left(\frac{n^2 - n}{2} \right) \\
&= \frac{17n^2}{2} - \frac{5n}{2} + 3
\end{aligned}$$

3 Exercices : Notations asymptotiques, classes de complexité

Exercice 6 (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 $n^2 + n + 10 \in \Theta(n^2)$

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 $n^2 + n + 10 \in \Theta(n^2)$.
   Pour $r = 0$ et $c = 1$, on a bien pour tout $i \geq r$, $i^2 + i + 10 \geq c \cdot i^2$, donc $n^2 + n + 10 \in \Omega(n^2)$.
   Pour $r = 1$ et $c = 12$, on a bien pour tout $i \geq r$, $i^2 + i + 10 \leq c \cdot i^2 = 12i^2$, donc $1 + 1/i + 10/i \leq 12$, donc $n^2 + n + 10 \in O(n^2)$.
   Donc, $n^2 + n + 10 \in \Theta(n^2)$.

Exercice 7 (Complexités de sommes de fonctions)**.**

A. Classer les fonctions suivantes en fonction des types de complexité proposés :
$2n^2 - 5$ $\quad 3n^4 + 2n$ $\quad 33 \log n + 56$ $\quad 8^n + n^3$ $\quad 348n - \sqrt{n}$ $\quad 1 + \frac{1}{n}$

   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 $f_1 + f_2$? Démontrez-le rigoureusement.

Solution.

A.
1. temps constant $\Theta(1)$ — $1 + \frac{1}{n}$
2. temps logarithmique $\Theta(\log n)$ — $33 \log n + 56$
3. temps linéaire $\Theta(n)$ — $348n - \sqrt{n}$
4. temps quadratique $\Theta(n^2)$ — $2n^2 - 5$
5. temps polynomial $\Theta(n^c)$ pour une constante $c \geq 1$ — $3n^4 + 2n$
6. temps exponentiel $\Theta(c^n)$ pour une constante $c > 1$ — $8^n + n^3$

B. $f_1 + f_2 \in \Theta(\max(g_1, g_2))$. 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 $f_1 + f_2 \in \Omega(\max(g_1, g_2))$.

Exercice 8 (Notations asymptotiques, suite)**.**

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

Solution.

1. 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 $100n \notin \Omega(n^2)$.

2. 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 $n^k \in O(2^n)$.
3. 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 $2^n \notin \Omega(3^n)$.

4 Exercices : exemples de complexités avancées

Exercice 9 (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 10 (Colorations de graphes)**.**

On rappelle le problème de *coloration de graphes* : étant donné un graphe non orienté $G=(V,E)$, attribuer un entier $c(v)$ à chaque sommet v (appelé *couleur*), de telle façon que pour chaque arête xy, $c(x) \neq c(y)$. On cherche à trouver le plus petit nombre possible de couleurs permettant de colorer le graphe selon cette contrainte.

1. Proposer une fonction `verif_col(G,c)` qui, étant donné un graphe $G=(V,E)$ à n sommets et une coloration potentielle $c : V \to \mathbb{N}$, vérifie que c est une coloration valide de G.
2. Quelle est sa complexité en fonction du nombre n de sommets ?
3. Quel est le nombre de colorations (pas forcément valides) possibles pour un graphe à n sommets, avec au plus k couleurs ?
4. On propose l'algorithme suivant pour le problème de coloration de graphes :

```
coloration(G): coloration du graphe G
Entrée : un graphe G=(V,E) à n sommets
Sortie : une coloration optimale de G
  • nb_opt = n
  • col_opt = []
  • Pour chaque coloration c possible de G:
      - nb_c = nombre de couleurs de c
      - Si verif_col(G,c) et nb_c < nb_opt alors:
        nb_opt = nb_c
        col_opt = c
  • Retourner col_opt
```

Quelle est la complexité en temps $T(n)$ de `coloration(G)` en fonction du nombre n de sommets de G ?

Solution.

1.
```
  • Pour chaque arête xy de E faire:
      - Si c(x) == c(y):
        Retourner Faux
  • Retourner Vrai
```

2. La complexité est en $O(|E|)$, c'est à dire $O(n^2)$ car il y a au plus $\binom{n}{2} = \frac{n(n-1)}{2}$ arêtes.
3. On peut représenter une coloration par une liste de longueur n où chaque élément prend sa valeur entre 1 et k (ou 0 et $k-1$). Il y a donc $k \times k \times \ldots k = k^n$ listes possibles, donc k^n colorations potentielles.

4. Le nombre de couleurs maximum est n, d'après la question 3, il y a donc n^n colorations possibles à tester. Pour chacune, d'après la question 2, on a un traitement en $O(n^2)$. La complexité totale est donc de $O(n^n \times n^2) = O(n^{n+2})$. Cela fait également $2^{log_2(n)^{(n+2)}} = 2^{log_2(n)\times(n+2)}$ que l'on peut écrire $2^{O(n\log(n))}$ (avec un léger abus de notation), à comparer à 2^n.