CHAPITRE 1

Analyse des algorithmes

Plan du chapitre

Introduction

Dans ce chapitre, nous allons étudier l'analyse des algorithmes en nous posant les trois questions suivantes.

1) **Problème de la terminaison :** est-ce que l'algorithme se termine ?
2) **Problème de la correction :** est-ce que le résultat renvoyé par l'algorithme est correct ?
3) **Problème de la complexité :** est-ce que l'algorithme se termine en un temps raisonnable ?

Pour répondre à ces différentes questions, nous allons introduire différents outils généraux.

Partie I Preuve des algorithmes

Dans cette partie, nous allons étudier les problèmes de la terminaison et de la correction d'un algorithme.

I.A - Terminaison d'un algorithme

Définition (Terminaison d'un algorithme) : La terminaison d'un algorithme désigne la propriété qu'un algorithme se termine pour toute entrée valide.

Remarques 1 :

a) Si un algorithme vérifie la propriété de la terminaison, on dit que l'algorithme se termine (pour toutes ses entrées valides).

b) Le problème de la terminaison d'un algorithme se pose principalement dans deux situations :
- si l'algorithme contient une ou plusieurs boucles `while` (ce cas sera étudié ci-dessous) ;
- si l'algorithme est récursif (ce cas sera étudié dans la dernière sous-partie).

Exemple 1 : Considérons les fonctions `exa` et `exb` définies ci-dessous.

```python
def exa(n:int) -> None:
    for k in range(n):
        print(k)
```

```python
def exb(n:int) -> None:
    while n != 0:
        n -= 1
```

La fonction `exa` se termine pour tout entier `n` passé en entrée : on dit que la fonction `exa` termine. Par contre, la fonction `exb` définie ci-dessous ne se termine que si l'entier `n` passé en entrée est positif : la fonction `exb` ne se termine pas (pour toutes les entrées valides).

Pour prouver qu'un algorithme se termine, nous allons introduire la notion de variant de boucle.

Définition (Variant de boucle) : Un variant de boucle est une expression à valeurs dans $\mathbb{N}$ qui diminue strictement à chaque itération de la boucle.

Proposition (Terminaison d'une boucle avec variant) : Si une boucle admet un variant, alors elle termine nécessairement.

DÉMONSTRATION :

Si la boucle ne se terminait pas, alors les valeurs successives du variant fournirait une suite $(v_n)_{n\in\mathbb{N}}$ strictement décroissante d'entiers naturels, ce qui est impossible.

Exemple 2 : Considérons la fonction `factorielle(n)` définie ci-dessous prenant un argument en entier naturel `n` et qui renvoie la factorielle de `n`.

```
def factorielle(n:int) -> int:
    k = 1
    p = 1
    while k < n:
        k += 1
        p *= k
    return(p)
```

On remarque que la quantité $n-k$ est un variant de boucle. D'une part, elle est entière et positive par la condition imposée par la boucle **while**. D'autre part, si on note k' la valeur mise à jour de k après le passage dans la boucle, on a $k' = k+1$, donc $n-k' = n-k-1 < n-k$. On conclut que la fonction `factorielle` se termine.

Exemple 3 : Considérons la fonction `recherche_dicho(t, x)` définie ci-dessous prenant en argument un tuple non vide `t` d'entiers triés dans l'ordre croissant et un entier `x` et qui renvoie un booléen indiquant si `x` est dans `t`.

```
def recherche_dicho(t:tuple, x:int) -> bool:
    a = 0
    b = len(t)
    while b - a > 1:
        c = (a + b) // 2
        if t[c] <= x:
            a = c
        else:
            b = c
    return(t[a]==x)
```

On remarque que la quantité $b-a$ est un variant de boucle. D'une part, elle est entière et positive par la condition imposée par la boucle **while**. D'autre part, notons a' et b' les valeurs mises à jour de a et b après le passage dans la boucle. On introduit également le quotient $c \in \mathbb{N}$ et le reste $r \in \{0,1\}$ dans la division euclidienne de $a+b$ par 2, ce qui nous permet d'écrire

$$a+b = 2c+r \quad \Leftrightarrow \quad c = \frac{a+b-r}{2}.$$

Pour comparer $b'-a'$ avec $b-a$, on distingue deux cas.

- Si $t[c] \leqslant x$, alors on a $(a', b') = (c, b)$ et

$$b'-a' = b-c = b-\left(\frac{a+b-r}{2}\right) = \frac{b-a+r}{2} \leqslant \frac{b-a}{2}+\frac{1}{2} < b-a \quad \text{car} \quad b-a > 1.$$

- Sinon, on a $(a', b') = (a, c)$ et

$$b'-a' = c-a = \left(\frac{a+b-r}{2}\right)-a = \frac{b-a-r}{2} \leqslant \frac{b-a}{2} < b-a \quad \text{car} \quad b-a > 0.$$

On conclut que la fonction `recherche_dicho` se termine.

Exercice 1 : On considère la fonction `div_euc(a, b)` définie ci-dessous prenant en argument deux entiers naturels `a` et `b` avec `b` non nul et qui renvoie le quotient et le reste dans la division euclidienne de `a` par `b`.

```python
def div_euc(a:int, b:int) -> tuple:
    r = a
    q = 0
    while r >= b:
        r -= b
        q += 1
    return(q, r)
```

Montrer la terminaison de la fonction `div_euc`.

Exercice 2 : On considère la fonction `mystere(a, b)` définie ci-dessous prenant en argument un entier `a` et un entier naturel `b`.

```python
def mystere(a:int, b:int) -> int:
    r = 1
    e = b
    p = a
    while e > 0:
        if e % 2 == 1:
            r = r * p
            p = p * p
            e = e // 2
    return(r)
```

Montrer la terminaison de la fonction `mystere`.

Exercice 3 : On considère la fonction `tri_insertion(L)` définie ci-dessous prenant en argument une liste d'entiers `L` et qui trie cette liste.

```python
def tri_insertion(L:list) -> None:
    for k in range(1, len(lst)):
        x = L[k]
        i = k
        while i > 0 and L[i-1] > x:
            L[i] = L[i-1]
            i -= 1
        L[i] = x
```

Montrer la terminaison de la fonction `tri_insertion`.

I.B - Correction d'un algorithme

Définition (Correction partielle) : On dit que la correction d'un algorithme est partielle si le résultat renvoyé est correct pour toute entrée valide pour laquelle l'algorithme se termine.

Définition (Correction totale) : On dit que la correction d'un algorithme est totale si elle est partielle et si l'algorithme vérifie la propriété de la terminaison.

Remarque 2 : Autrement dit, la correction d'un algorithme est totale si pour toute entrée valide, il se termine et que le résultat renvoyé est correct.

Pour prouver la correction partielle d'un algorithme, nous allons introduire la notion d'invariant de boucle.

Définition (Invariant de boucle) : Un invariant de boucle est une propriété vérifiant les conditions suivantes :

(i) la propriété est vraie avant la première itération de la boucle ;

(ii) si cette propriété est vraie avant une itération de la boucle, elle le demeure après l'itération.

Remarque 3 : Un invariant de boucle peut être utilisé en amont de la rédaction d'un algorithme ; il peut aussi servir à prouver la correction partielle d'un algorithme existant et enfin il peut servir à analyser un algorithme pour en comprendre le rôle.

Exemple 4 : Reprenons la fonction `factorielle` définie dans l'exemple 2. Montrons que la propriété $\mathscr{P} : p = k!$ est un invariant de boucle.

(i) Avant la première itération de la boucle, on a $p = 1 = 1! = k!$, donc la propriété est vraie.

(ii) On suppose que la propriété est vraie avant une itération de la boucle pour des nombres p et k. Notons p' et k' les valeurs mises à jour de p et k après le passage dans la boucle. On a alors $p' = p \times k' = k! \times (k+1) = (k+1)! = k'!$.

Finalement, la boucle s'arrêtera lorsque $k = n$. Par l'invariant de boucle ci-dessus, on aura $p = n!$, donc nous venons de prouver la correction partielle de la fonction `factorielle`. Comme nous avions déjà prouvé la terminaison de cette fonction dans l'exemple 2, on conclut que nous avons prouvé sa correction totale.

Exemple 5 : Reprenons la fonction `recherche_dicho` définie dans l'exemple 3. Montrons que la propriété

$$\mathscr{P} : \big(x \in t \ \Rightarrow \ \exists k \in [\![a, b[\![, \quad x = t[k] \big)$$

est un invariant de boucle.

(i) Avant la première itération de la boucle, on a $a = 0$ et $b = \text{len}(t) - 1$, donc la propriété est vraie.

(ii) On suppose que la propriété est vraie avant une itération de la boucle pour des nombres a et b. Notons a' et b' les valeurs mises à jour de a et b après le passage dans la boucle. Supposons que $x \in t$.

- Si $t[c] \leqslant x$, alors on a $(a', b') = (c, b)$ et $t[a'] = t[c] \leqslant x < t[b] = t[b']$.
- Sinon, on a $(a', b') = (a, c)$ et $t[a'] = t[a] \leqslant x < t[c] = t[b']$.

La boucle s'arrêtera lorsque la condition $b - a > 1$ n'est plus vérifiée, donc on aura $b - a \leqslant 1$. Par l'invariant de boucle ci-dessus, on aura également

$$\mathscr{P} \ : \ \big(x \in t \ \Rightarrow \ \exists k \in [\![a, b[\![, \quad x = t[k] \big).$$

On conclut que $x \in t$ si et seulement si $t[a] = x$, ce qui permet de conclure que la fonction renvoie le résultat correct : nous venons de prouver la correction partielle de la fonction `recherche_dicho`. Comme nous avions déjà prouvé la terminaison de cette fonction dans l'exemple 3, on conclut que nous avons prouvé sa correction totale.

Exercice 4 : On considère la fonction `somme(lst)` définie ci-dessous prenant en argument une liste d'entiers `lst` et qui renvoie la somme de ses éléments.

```python
def somme(lst:list) -> int:
    s = 0
    for k in range(len(lst)):
        s += lst[k]
    return(s)
```

Montrer la correction totale de la fonction `somme`.

Exercice 5 : On considère la fonction `maxi(t)` définie ci-dessous prenant en argument un tuple d'entiers `t` de longueur non nulle et qui renvoie l'élément maximal du tuple `t`.

```python
def maxi(t:tuple) -> int:
    res = t[0]
    for k in range(len(t)):
        if x > t[k]:
            res = x
    return(res)
```

Montrer la correction totale de la fonction `maxi`.

Exercice 6 : On considère la fonction `mystere(a, b)` définie dans l'exercice 2.

1) Montrer que la propriété « $rp^e = a^b$ » est un invariant de la boucle **`while`**.
2) En déduire ce que fait la fonction `mystere`.

Exercice 7 : On considère la fonction `tri_insertion(L)` définie dans l'exercice 3.

1. Montrer que « `L[i] > x` » est un invariant de la boucle **`while`**.
2. Montrer que « la liste `L[:k]` est triée dans l'ordre croissant » est un invariant de la boucle **`for`**.
3. En déduire la correction totale de `tri_insertion`.

I.C - Correction totale d'un algorithme récursif

Pour prouver la correction totale d'un algorithme récursif, on peut souvent utiliser un raisonnement par récurrence. Nous nous limiterons à étudier un exemple simple.

Exemple 6 : Considérons la fonction récursive `produit(L)` définie ci-dessous prenant en argument une liste d'entiers `L` et qui renvoie le produit des éléments de la liste `L`.

```
1 def produit(L:list) -> int:
2     n = len(L)
3     if n == 0:
4         res = 1
5     else:
6         L_2 = L[:n-1]
7         res = produit(L_2) * L[n-1]
8     return(res)
```

Nous allons montrer la correction totale de cette fonction en introduisant pour tout $n \in \mathbb{N}$ la propriété

$$\mathscr{P}_n : \left\{ \text{Pour toute liste d'entiers } L = [a_0, \dots, a_{n-1}] \text{ de longueur } \texttt{n}, \text{ la fonction } \texttt{produit} \right. \\ \left. \text{avec l'entrée } \texttt{L} \text{ se termine et elle renvoie } \prod_{k=0}^{n-1} a_k. \right.$$

que nous allons démontrer par récurrence sur l'entier $n \in \mathbb{N}$.

- **Initialisation.** Pour $n = 0$, on a par définition que `produit(L)` se termine et renvoie 1, ce qui correspond bien par convention au produit vide, donc $\mathscr{P}_0$ est vraie.
- **Hérédité.** Soit $n \in \mathbb{N}$ tel que la propriété $\mathscr{P}_n$ est vraie. Considérons une liste d'entiers $L = [a_0, \dots, a_n]$ de longueur $n+1$. En notant $L_2 = [a_0, \dots, a_{n-1}]$ qui est une liste de longueur n, on a par hypothèse de récurrence que l'exécution de `produit(L_2)` se termine et renvoie $\prod_{k=0}^{n-1} a_k$. Par définition de la fonction `produit`, on conclut que `produit(L)` se termine et qu'elle renvoie

  $$\left(\prod_{k=0}^{n-1} a_k \right) \times a_n = \prod_{k=0}^{n} a_k,$$

  donc la propriété $\mathscr{P}_{n+1}$ est vraie.

On conclut que la propriété $\mathscr{P}_n$ est vraie pour tout $n \in \mathbb{N}$, donc nous avons prouvé la correction totale de la fonction `produit`.

Partie II Complexité temporelle d'un algorithme

II.A - Une première approche de la complexité temporelle

Pour de nombreux problèmes d'informatique, il existe souvent plusieurs algorithmes permettant de les résoudre. Il est alors naturel de vouloir comparer l'efficacité de ses différentes solutions.

Dans le cadre du programme d'informatique, nous nous intéressons principalement à la **complexité temporelle** d'un algorithme, i.e. à l'évaluation de la durée nécessaire à l'algorithme pour fournir son résultat. Pour fixer le cadre de cette partie, nous supposerons que la technologie employée est celle d'une machine à processeur unique pour laquelle les instructions seront exécutées l'une après l'autre (i.e. sans opérations simultanées).

Une première approche pourrait s'appuyer sur des expérimentations : on mesurerait le temps effectif pris par différents algorithmes pour résoudre le problème étudié. Cependant, cette approche présente de nombreuses limites :

- les durées mesurées dépendront entièrement de la machine utilisée ;
- un ordinateur gère de nombreuses tâches de fond de manière invisible pour l'utilisateur (fonctionnement du système d'exploitation, gestion de la mémoire, etc.) : ces processus peuvent altérer de manière très significative les mesures de durées ;
- on souhaite pouvoir évaluer l'évolution du temps d'exécution de l'algorithme pour des données d'entrée de plus en plus grandes sans avoir besoin de réellement exécuter le programme sur un ordinateur) ;
- etc.

Nous allons donc procéder à une analyse du code lui-même pour évaluer la complexité temporelle d'un algorithme. De plus, comme le traitement d'un certain volume de données requiert un temps d'exécution lié à ce volume de données, nous allons évaluer le nombre d'**instructions élémentaires** qu'effectue le programme en fonction de la **taille de l'entrée**. Les instructions élémentaires principales sont les suivantes :

- affectation d'un entier, d'un flottant ou d'un booléen dans une variable ;
- accès à une variable ;
- les opérations arithmétiques sur les entiers ou les flottants : `+`, `-`, `*`, `/`, `//`, `%` ;
- les opérations booléennes : `and`, `or`, `not` ;
- les comparaisons d'entiers, de flottants ou de booléens.

Remarques 4 :

a) On considère que les différentes instructions élémentaires ci-dessus sont exécutées par l'ordinateur en temps constant, i.e. il existe une constante $\tau > 0$ (dépendante de l'ordinateur utilisé) de sorte que l'exécution d'une des instructions ci-dessus soit traitée en moins de τ secondes. Ainsi, il suffit de compter le nombre d'instructions élémentaires $C(n)$ dans un algorithme pour obtenir une majoration de sa durée d'exécution.

b) Pour que l'hypothèse précédente soit raisonnable, nous supposons que les nombres entiers et flottants sont codés sur un nombre fixe de bits dans la mémoire de l'ordinateur. Dans notre contexte d'utilisation, cette hypothèse sera vérifiée en général : les entiers et les flottants sont représentés usuellement sur 64 bits. Dans d'autres cadres (hors programme), cette supposition ne peut être faire et nécessiterait d'utiliser un autre modèle. Par exemple, certains algorithmes de chiffrement et déchiffrement manipulent des entiers de plusieurs centaines de bits.

Finalement, nous devons préciser ce qu'on appelle la taille de l'entrée. De manière générale, il s'agit d'une quantité permettant d'évaluer la place occupée par les entrées en mémoire. Voici les tailles usuelles des données les plus fréquentes :

- la taille d'une entrée entière est sa valeur en général ;
- la taille d'une structure séquentielle (chaîne de caractères, tuple, liste, etc.) est sa longueur en général.

Lorsque l'algorithme a plusieurs entrées, on peut utiliser la taille de chaque entrée pour évaluer sa complexité.

Exemple 7 : Considérons la fonction `puissance(a, n)` définie ci-dessous prenant en entrée un entier `a` et un entier `n` et qui renvoie la puissance `n`-ième de `a`.

```
def puissance(a:int, n:int) -> int:
    res = 1         # 1 instruction élémentaire
    i = 0           # 1 instruction élémentaire
    while i < n:    # La boucle fait n tours. 3 instructions élémentaires
        res *= a            # 4 instructions élémentaires dans la boucle
        i += 1              # 3 instructions élémentaires dans la boucle
    return(res)
```

On obtient que le nombre d'instructions élémentaires est

$$C(n) = 2 + (3+4+3)n + 3 = 10n + 5.$$

Exemple 8 : Considérons la fonction `somme(L)` définie ci-dessous prenant en argument une liste d'entiers `L` et qui renvoie la somme des éléments de `L`.

```
def somme(L:lst) -> int:
    res = 0             # 1 instruction élémentaire
    for x in L:         # La boucle fait n = len(L) tours.
        res += x            # 4 instructions élémentaires dans la boucle
    return(res)
```

En notant n la longueur de la liste `L` en entrée, on obtient que le nombre d'opérations élémentaires effectuées par la fonction est $C(n) = 4n + 1$.

Finalement, il est fréquent que le nombre d'opérations élémentaires diffèrent pour différentes entrées de même taille. Dans ce cas, nous retiendrons le nombre d'opérations élémentaires dans le pire des cas, ce qui permet d'obtenir un majorant du nombres d'opérations élémentaires pour toutes les entrées possibles d'une même taille.

Exemple 9 : Considérons la fonction définie ci-dessous.

```
def exemple(n:int) -> int:
    res = 0                 # 1 instruction élémentaire
    if n >= 0:              # 2 instructions élémentaires
        for k in range(n):  # La boucle fait n tours.
            res += k            # 4 instructions élémentaires
    return(res)
```

Dans la fonction ci-dessus : si $n < 0$, alors il y a 3 opérations élémentaires qui sont exécutées par le programme, tandis que si $n \geqslant 0$, il y a $4n+3$ opérations élémentaires qui sont exécutées par le programme. Comme on retient le pire des cas, on conclut que $C(n) = 4n + 3$.

Remarque 5 : D'autre choix sont possibles pour étudier l'efficacité d'un programme. Par exemple, on aurait pu étudier la complexité en moyenne, i.e. d'effectuer la moyenne du nombre d'instructions élémentaires pour toutes les entrées possibles d'une même taille.

II.B - Complexité temporelle dans le pire cas en ordre de grandeur

Finalement, compter de manière aussi précise que dans la sous-partie précédente le nombre d'opérations élémentaires a peu d'intérêt pour de nombreuses raisons (notamment parce que le temps d'exécution des opérations élémentaires diffèrent selon les opérations élémentaires). Ainsi, nous allons nous limiter à étudier l'ordre de grandeur de la complexité temporelle dans le pire des cas d'un algorithme en fonction de la taille de ses entrées.

Pour formaliser cette idée, on introduit la notion de domination pour des suites définies ci-dessous.

Définition (Relation de domination pour les suites) : Soient $(u_n)_{n\geqslant n_0}$ et $(v_n)_{n\geqslant n_0}$ deux suites de nombres réels. On dit que la suite $(u_n)_{n\geqslant n_0}$ est dominée par la suite $(v_n)_{n\geqslant n_0}$, ce que l'on note $u_n = O(v_n)$, si

$$\exists A \in \mathbb{R}, \quad \exists N \in [\![n_0, +\infty[\![, \quad \forall n \in [\![N, +\infty[\![, \quad |u_n| \leqslant A|v_n|.$$

Remarques 6 :

a) On peut reformuler la définition : on a $u_n = O(v_n)$ si et seulement si

$$\exists (A,B) \in \mathbb{R}^2 \quad \forall n \in [\![n_0, +\infty[\![, \quad |u_n| \leqslant A|v_n| + B.$$

b) On a $u_n = O(1)$ si et seulement si la suite $(u_n)_{n\geqslant n_0}$ est bornée.

c) Pour tout $a \in]1, +\infty[$ et tout $d \in \mathbb{N}$, on a les relations de domination suivantes

$$1 = O\big(\log(n)\big), \quad \log(n) = O(n), \quad n = O\big(n^2\big), \quad n^d = O\big(n^{d+1}\big), \quad n^d = O\big(a^n\big).$$

d) Soient (u_n), (v_n), (w_n) et (t_n) quatre suites de nombres réels. On a les propriétés suivantes.

- Si $u_n = O(w_n)$ et $v_n = O(w_n)$, alors $u_n + v_n = O(w_n)$.
- Si $u_n = O(w_n)$ et $v_n = O(t_n)$, alors $u_n v_n = O(w_n t_n)$.

Exemple 10 : En utilisant les règles ci-dessus, on a

$$\log(n) + 1 + (n+3)\log(n) = O\big(n\log(n)\big), \qquad n^3 + n^2 + n + 1 = O\big(n^3\big), \qquad 2^n + n^2 + 1 = O\big(2^n\big).$$

Définition (Complexité temporelle) : On considère un algorithme dont on note $C(n)$ le nombre d'opérations élémentaires dans le pire des cas pour une taille n de ses entrées. On dit que l'algorithme est

(i) de complexité constante si $C(n) = O(1)$;
(ii) de complexité logarithmique si $C(n) = O\big(\log(n)\big)$;
(iii) de complexité linéaire si $C(n) = O(n)$;
(iv) de complexité quasi-linéaire si $C(n) = O\big(n\log(n)\big)$;
(v) de complexité quadratique si $C(n) = O\big(n^2\big)$;
(vi) de complexité polynomiale si $C(n) = O\big(n^d\big)$ pour un entier $d \geqslant 2$;
(vii) de complexité exponentielle si $C(n) = O\big(a^n\big)$ pour un réel $a > 1$.

Exemple 11 : Les différents exemples de la sous-partie précédente sont de complexité linéaire.

Remarque 7 : La fonction `len` permettant d'obtenir la longueur d'une liste, d'une chaîne de caractères, d'un tuple ou d'un dictionnaire est de complexité constante. En effet, l'implémentation de ces objets en Python sauvegarde en mémoire leur longueur, donc la fonction `len` n'a pas besoin d'effectuer un parcours de la structure.

Exemples 12 :

a) Considérons la fonction définie ci-dessous prenant en argument une liste d'entiers `L`.

```
def exemple(L:list) -> list:
    n = len(L)              # 2 = O(1) instructions élémentaires
    for k in range(1, n):   # La boucle fait n-1 tours.
        L[k] += L[k-1]          # 6 = O(1) instructions élémentaires
    for k in range(n-1):    # La boucle fait n-1 tours.
        L[k] += L[k+1]          # 6 = O(1) instructions élémentaires
    return(L)
```

Comme les deux boucles dans le code ci-dessus sont successives (elles ne sont pas imbriquées), on conclut que la complexité temporelle de cette fonction est linéaire.

b) Considérons la fonction `tri_bulles(L)` prenant en entrée une liste d'entiers `L` et qui trie la liste `L`.

```
def tri_bulles(L:list) -> None:
    n = len(lst)            # 2 = O(1) inst. élém.
    for k in range(1, n):   # La boucle fait n-1 tours.
        for i in range(n-k):  # La boucle fait n-k tours, donc moins de n tours.
            if L[i] > L[i+1]:                 # 5 = O(1) inst. élém.
                L[i], L[i+1] = L[i+1], L[i]   # 5 = O(1) inst. élém.
```

Comme les deux boucles dans le code ci-dessus sont imbriqués, on conclut que la complexité temporelle de cette fonction est quadratique.

La détermination de la complexité temporelle en ordre de grandeur d'un algorithme ne permet pas d'en déduire exactement le temps d'exécution de ce dernier, mais cette notion permet de comparer entre eux deux algorithmes résolvant le même problème.

Cependant, il importe de prendre conscience des différences d'échelle considérables qui existent entre les ordres de grandeurs définies précédemment. En se basant sur une machine réalisant une opération élémentaire par nanoseconde, on obtient le tableau ci-dessous donnant le temps d'exécution pour réaliser un nombre d'opérations élémentaires en fonction de la taille n de l'entrée.

Taille entrée	1	$\log_2(n)$	n	$n\log_2(n)$	n^2	n^3	2^n
$n=10^1$	1 ns	3 ns	10 ns	33 ns	100 ns	1 µs	1 us
$n=10^2$	1 ns	7 ns	100 ns	664 ns	10 µs	1 ms	$4\cdot10^{13}$ ans
$n=10^3$	1 ns	10 ns	1 µs	10 µs	1 ms	1 s	$3\cdot10^{284}$ ans
$n=10^4$	1 ns	13 ns	10 µs	133 µs	100 ms	17 min	×
$n=10^5$	1 ns	17 ns	100 µs	1,6 ms	10 s	11,6 jours	×
$n=10^6$	1 ns	20 ns	1 ms	20 ms	17 min	31,7 ans	×

Exercice 8 : Étudier la complexité temporelle de la fonction `maxi(t)` définie dans l'exercice 5.

Exercice 9 : Étudier la complexité temporelle de la fonction `div_euc(a, b)` définie dans l'exercice 1.

Exercice 10 : On considère la fonction `distintcs(t)` définie ci-dessous.

```python
def distincts(t:tuple) -> bool:
    res = True
    for i in range(len(t)):
        for j in range(i+1, len(t)):
            if t[i] == t[j]:
                res = False
    return(res)
```

1. Étudier la complexité temporelle de la fonction `distincts`.
2. Que fait la fonction `distincts` ?

Exercice 11 : Étudier la complexité temporelle de la fonction `tri_insertion(L)` définie dans l'exercice 3.

Exercice 12 : Étudier la complexité temporelle de la fonction `recherche_dicho(t, x)` définie dans l'exemple 3.

II.C - Complexité temporelle des algorithmes récursifs

L'étude de la complexité d'un algorithme récursif se ramène souvent à l'étude du comportement asymptotique d'une suite définie par récurrence. Nous nous limiterons à étudier un exemple simple.

Exemple 13 : Considérons la fonction récursive `produit(L)` définie ci-dessous prenant en argument une liste d'entiers `L` et qui renvoie le produit des éléments de la liste `L`. Notons $C(n)$ le nombre d'opérations élémentaires dans le pire des cas pour une liste `L` de longueur n en entrée.

```
def produit(L:list) -> int:
    n = len(L)                  # 2 instructions élémentaires
    if n == 0:                  # 2 instructions élémentaires
        res = 1                     # 1 instruction élémentaire
    else:
        L_2 = L[:n-1]               # 2(n-1) instructions élémentaires
        res = produit(L_2) * L[n-1]   # C(n-1) + 2 instructions élémentaires
    return(res)
```

Par analyse du code ci-dessus, on obtient $C(0) = 4$ et la relation de récurrence ci-dessous.

$$\forall n \in \mathbb{N}^*, \quad C(n) = 2 + 2(n-1) + C(n-1) + 2 = C(n-1) + 2n + 2.$$

On en déduit pour tout $n \in \mathbb{N}$ que

$$\begin{aligned} C(n) = (C(n) - C(0)) + C(0) &= \sum_{k=1}^{n} \big(C(k) - C(k-1)\big) + C(0) \\ &= \sum_{k=1}^{n} \big(2k+2\big) + C(0) = n(n+1) + 2n + 4 = O\big(n^2\big), \end{aligned}$$

donc la complexité temporelle de la fonction `produit` est quadratique.

Remarques 8 :

a) Avec une aisance suffisante sur les relations de domination, on pouvait écrire pour tout $n \in \mathbb{N}$ la relation de récurrence sous la forme $C(n) = C(n-1) + O(n)$, puis en déduire que

$$C(n) = \sum_{k=1}^{n} \big(C(k) - C(k-1)\big) + O(1) = \sum_{k=1}^{n} O(k) + O(1) = O\left(\frac{n(n+1)}{2}\right) + O(1) = O\big(n^2\big).$$

b) On peut être surpris d'avoir obtenu une complexité quadratique pour la fonction récursive `produit`, alors que sa version itérative, définie ci-dessous, a une complexité linéaire.

```
def produit_iter(L:list) -> int:
    res = 1         # 1 instruction élémentaire
    for x in L:     # La boucle fait n tours
        res *= x        # 3 instructions élémentaires
    return(res)
```

L'origine du problème se trouve dans l'utilisation de l'instruction `L[:n-1]` qui effectue une copie quasi-complète de la liste de départ à chaque appel de la fonction `produit`. Comme une copie est effectuée à chaque étape de la récursivité, le temps d'exécution augmente significativement en passant d'une complexité linéaire à une complexité quadratique.

c) Pour éviter le problème précédent, on pouvait écrire notre fonction `produit` de la manière suivante.

```python
def produit(L:list) -> int:

    def produit_rec(lst:list, n:int) -> int:
        if n == 0:
            res = 1
        else:
            res = produit_rec(lst, n-1) * L[n-1]
        return(res)

    return(produit_rec(L, len(L)))
```

Dans cette nouvelle version, plus aucune copie de la liste n'est effectuée et on obtient bien une complexité linéaire comme dans la version itérative.

II.D - Complexité spatiale

De la même façon qu'on définit la complexité temporelle d'un algorithme pour évaluer ses performances en temps de calcul, on peut définir sa **complexité spatiale** pour évaluer sa consommation en espace mémoire. Le principe est le même sauf qu'ici on cherche à évaluer l'ordre de grandeur de la quantité de mémoire utilisée dans le pire des cas d'un algorithme en fonction de la taille de ses entrées.

Exemple 14 : Considérons la fonction `binomial(n, p)` définie ci-dessous prenant en argument un couple d'entier naturel `(n, p)` et qui renvoie le coefficient binomial `p` parmi `n`.

```python
import numpy as np # Pour utiliser des tableaux à deux dimensions

def binomial(n,p):
    c = np.zeros((n+1, p+1), dtype=int) # Tableau (n+1) x (p+1) rempli de zéros

    for i in range(n+1): # On remplit les i parmi 0 avec 1
        c[i,0] = 1

    for i in range(1, n+1): # On remplit le triangle de Pascal
        for j in range(1, p+1):
            c[i,j] = c[i-1,j-1] + c[i-1,j] # Formule de Pascal

    return(c[n,p])
```

La complexité spatiale de cette fonction est donnée par le nombre de cases mémoires présentes dans le tableau `c`, donc sa complexité spatiale est $O\big((n+1)(p+1)\big) = O(np)$.