Chapitre (S2) 2 Preuve et complexité

Objectifs

- Savoir prouver la terminaison d'une boucle.
- Savoir prouver un programme simple en utilisant un invariant de boucle.
- Savoir calculer la complexité temporelle et/ou spatiale d'un programme.

1 INTRODUCTION

Un algorithme est une suite finie de règles et d'opérations élémentaires mises en oeuvre sur un nombre fini de données en vue de résoudre un problème spécifique. Si un algorithme est susceptible de résoudre un problème, il convient également de s'interroger sur sa validité et sur ses performances.

- L'algorithme termine-t-il ?
- L'algorithme répond-il aux spécifications ?
- De combien de temps et de ressources mémoires a-t-il besoin ?

Les deux premières questions soulèvent le problème de la *preuve d'un algorithme* en termes de *terminaison* et de *correction* de ce dernier. La troisième question a trait à la *complexité* de l'algorithme, en termes temporels et spatiaux.

Pour fonctionner, un algorithme reçoit généralement un jeu de données en entrée. Après leur traitement, il renvoie un résultat en sortie. Le traitement peut souvent se décomposer en blocs simples :

- opérations d'affectation, entrées/sorties, manipulations de variables ;
- structures conditionnelles `if`, `elif`, `else` ;
- structures répétitives `for`, `while`.

Ce découpage essentiel en blocs simples permet l'analyse de l'algorithme pour en établir la preuve et en évaluer la complexité.

Dans la suite du cours, tous les algorithmes sont exprimés sous la forme de programmes écrits en Python. Nous parlerons de preuve de programmes et de leur complexité.

2 TERMINAISON

2.1 Exemple

Prouver qu'un *programme termine*, c'est montrer que, quel que soit le jeu de données passé en entrée respectant la pré-condition, chaque bloc simple est traité en un nombre fini d'opérations.

Les opérations d'affectation, les entrées/sorties, les manipulations de variables terminent toujours. Il en est de même des structures conditionnelles (`if`) et des boucles inconditionnelles (`for`), sous réserve que la variable d'itération ne soit pas modifiée[1].

Les boucles conditionnelles `while` requièrent une attention particulière. Mal programmées, elles peuvent être à l'origine de boucles infinies. Prenons l'exemple suivant.

```
n = 5
while n != 0:
    n -= 1
```

Avant d'entrer dans la boucle `while`, la variable `n` est initialisée avec la valeur 5. Cette donnée d'entrée est traitée dans la boucle où sa valeur est décrémentée. `n` prend donc

1. Nous nous interdirons cette modification à l'avenir.

successivement les valeurs 4, 3, 2, 1 et 0. Pour cette dernière valeur, la condition `n != 0` n'étant plus vérifiée, la boucle se termine de sorte que le nombre d'opérations effectuées par ce programme est fini.

L'exemple suivant présente une situation *a priori* proche mais de comportement radicalement différent.

```
n = -1
while n != 0:
    n -= 1
```

La valeur initiale de n étant négative, la condition `n != 0` sera vraie à chaque tour de boucle car n ne fait que diminuer, et la boucle ne s'arrête jamais[2].

Ainsi, l'algorithme précédent termine si n est initialement un entier positif, c'est donc une pré-condition possible pour la boucle. Nous avons vu dans un précédent chapitre que cette pré-condition peut-être testée à l'aide d'une assertion avant la boucle, comme ceci :

```
assert n >= 0, "la valeur de n doit être positive"
while n != 0:
    n -= 1
```

2.2 Une propriété mathématique

La preuve de la terminaison d'un algorithme repose généralement sur le résultat suivant[3].

Rappel (mathématiques)

- Si a est un réel et (u_i) une suite d'entiers strictement croissante (*resp.* strictement décroissante), alors il existe un rang i_0 pour lequel $u_{i_0} > a$ (*resp.* $u_{i_0} < a$).
- Autrement dit, une suite d'entiers strictement croissante (*resp.* strictement décroissante), ne peut pas être majorée (*resp.* minorée).

Ce résultat exprime en particulier qu'il n'existe pas de suite infinie strictement décroissante dans $\mathbb{N}$. Ainsi, pour établir la terminaison d'un programme, on peut par exemple exhiber une suite d'entiers positifs, dépendant des données du programme, à valeurs dans $\mathbb{N}$, qui décroît strictement à chaque passage dans la boucle, ou bien plus généralement, une suite d'entiers minorée (*resp.* majorée) qui décroît (*resp.* croît) strictement à chaque passage dans la boucle.

2. En pratique, les limites physiques de la machine vont mener le programme à se terminer.
3. Voir le cours de mathématiques pour une preuve.

Dans l'exemple précédent, en posant u_k la valeur de n à la fin de l'itération n° k, on a $u_0 = n$ et, pour tout entier naturel k strictement positif, $u_k = u_{k-1} - 1$ (s'il y a une itération k), la suite (u_k) est donc une suite d'entiers strictement décroissante, et plus précisément $u_k = n - k$. Si $n \geqslant 0$, le passage dans la boucle se fait n fois et le programme termine (avec $u_n = 0$). Dans le cas contraire, la boucle est infinie et le programme ne termine pas.

Notation Évolution d'une variable dans une boucle

Le contenu d'une variable évoluant généralement à chaque passage dans une boucle, on utilisera la notation suivante : si `var` désigne le nom d'une variable, on note :

- $\texttt{var}_i$ le contenu de cette variable **à la fin** du i^{e} passage dans la boucle (itération i).
- Par convention, $\texttt{var}_0$ désigne le contenu de la variable juste **avant** la première exécution de la boucle (itération 0).

Exemple 1 Par exemple, considérons le programme :

```
S = 10
for k in range(2,5) :
    S += k
```

Les valeurs successives de la variable S sont $S_0 = 10$, $S_1 = 12$, $S_2 = 15$, $S_3 = 19$.

Remarque 1 L'indice i positionné sous la variable fait référence au numéro de l'itération dans la boucle, et non aux éléments de la liste décrite par la boucle. Dans notre exemple, i parcourt les valeurs $0, 1, 2, 3$ alors que k décrit les valeurs $2, 3, 4$.

2.3 Exemple

Considérons l'algorithme d'*exponentiation rapide* qui calcule x^n, pour un réel x et un entier naturel n, en utilisant uniquement des produits, des soustractions par 1 et des divisions par 2. Cet algorithme effectue généralement beaucoup moins de multiplications que les n attendues par la définition.

$$x^n = \underbrace{1 \times x \times x \times \cdots \times x}_{n \text{ mutliplications}}.$$

```
def expR(x: float, n: int) -> float :
    """ Renvoie x^n pour x réel et n entier naturel. """
    X = x
    N = n
    R = 1
```

```
    while N != 0 :
        if N%2 == 0 :
            N = N//2
        else :
            R = R*X
            N = (N-1)//2
        X = X*X
    return R
```

Avec la notation introduite plus haut, désignons par N_i la valeur prise par la variable `N` à la fin de l'itération i. Avant d'entrer dans la boucle **`while`**, $N_0 = n \in \mathbb{N}$ par hypothèse (pré-condition). Lors de la $(i+1)^e$ itération de la boucle **`while`** :

- si N_i est pair, alors $N_{i+1} = N_i//2$;
- si N_i est impair, alors $N_{i+1} = (N_i - 1)//2$

Dans les deux cas, si $N_i \in \mathbb{N}$, alors $N_{i+1} \in \mathbb{N}$ (récurrence).
D'autre part, la suite $(N_i)_{i\in\mathbb{N}}$ est strictement décroissante. En effet, pour $i \in \mathbb{N}$, supposons qu'il y ait une itération $i+1$, c'est à dire que $N_i \neq 0$:

- si N_i est pair, alors $N_{i+1} = N_i//2 < N_i$ puisque $N_i > 0$ par hypothèse ;
- si N_i est impair, alors $N_{i+1} = (N_i - 1)//2 < N_i$.

Supposons que la boucle ne se termine pas (raisonnement par l'absurde), alors, la suite $(N_i)_{i\in\mathbb{N}}$ est formée d'entiers positifs et elle est strictement décroissante, ce qui absurde. La boucle **`while`** termine donc.

Exercice 1

[Sol 1] On considère un entier $n \geqslant 0$ et x un réel quelconque.

1. Que fait la fonction suivante dans le code ci-dessous ? Établir sa terminaison et la documenter.

```
def f(x: float, n: int)->float :
    y = 1
    for i in range(n) :
        y *= x
    return y
```

2. Mêmes questions pour la fonction ci-dessous.

```
def f(x: float, n: int)->float :
    y = 1
    i = 0
    while i < n :
        y *= x
        i += 1
    return y
```

2.4 Problème de l'arrêt

Établir la terminaison d'une boucle n'est pas toujours simple. Il est des problèmes pour lesquels on ne peut que conjecturer le résultat. C'est le cas de la suite de SYRACUSE, rappelée ci-après.

Soit $(u_p)_{p\in\mathbb{N}}$ une suite définie par son premier terme $u_0 = n$, où n est un entier naturel non nul et par la relation de récurrence suivante :

$$\forall p \in \mathbb{N} \qquad u_{p+1} = \begin{cases} \frac{u_p}{2} & \text{si } u_p \text{ est pair,} \\ 3u_p + 1 & \text{si } u_p \text{ est impair.} \end{cases}$$

La conjecture de SYRACUSE affirme que cette suite finit toujours par une répétition de la séquence $4, 2, 1$, quelque soit l'entier naturel non nul n choisi. Mais ce résultat n'est pas prouvé à ce jour.

En Python, la fonction suivante renvoie l'indice du premier terme égal à 1 :

```
def syracuse(n: int)->int:
    """ Calcule les termes de la suite de Syracuse commençant par \
    ↪ l'entier strictement positif n jusqu'à ce qu'un terme \
    ↪ vaille 1 et renvoie l'indice de ce dernier """
    u = n
    indice = 0
    while u != 1 :
        if u%2 == 0 :
            u = u//2
        else :
            u = 3*u + 1
        indice += 1
    return indice
```

Selon la conjecture, cette fonction termine pour toute valeur de n. Mais sa terminaison n'est que conjecturée et donc non démontrée !

3 CORRECTION

3.1 Invariant de boucle

Définition 1 | Correction d'un algorithme

- Un algorithme est dit *correct*, si quels que soient les valeurs des paramètres d'entrée compatibles avec le fonctionnement de l'algorithme, ce dernier renvoie le résultat attendu. En découpant l'algorithme en blocs simples, cela revient à montrer que chaque bloc remplit une fonction bien identifiée.
- Établir la *correction d'un algorithme*, c'est montrer qu'il est correct.

Dans le cas des opérations d'affectation, des entrées/sorties, des manipulations de variables, des structures conditionnelles, l'analyse de leur action est relativement aisée. Celle des boucles `for` et `while` l'est moins. La notion d'*invariant de boucle*, qui a été introduite lors du chapitre précédent, permet d'établir la correction des boucles.

Rappel (Invariant de boucle) Un *invariant de boucle* est une propriété qui dépend des données de l'algorithme et qui est vérifiée juste avant la boucle, et après chaque passage dans la boucle (c'est-à-dire après chaque itération).

Nous allons mettre en évidence et utiliser les invariants de boucle sur quelques exemples d'algorithmes.

3.2 Moyenne d'une liste de nombres

Définition 2 | Moyenne d'une liste

La *moyenne* m d'un ensemble $L = \{\ell_0, \ell_1, \dots, \ell_{n-1}\}$ de n valeurs est définie par la relation suivante : $m = \dfrac{1}{n}\sum_{k=0}^{n-1} \ell_k$.

Compléter la fonction suivante pour qu'elle renvoie la moyenne des nombres contenus dans la liste `L` :

■■ Moyenne des éléments d'une liste

```
def moyenne(L: list)->float :
    """
    Calcule la moyenne des éléments de la liste de nombres L
    """
    S = 0
    for e in L:
        S += e
    return S/len(L)
```

PREUVE DE CORRECTION. On se propose de montrer par récurrence simple, que la propriété suivante est un invariant de boucle :

$\forall\, i \in \{0, \dots, n\}$, « $S_i = \sum_{k=0}^{i-1} L[k]$ » où i désigne le numéro de l'itération.

Initialisation. Par convention, si $a > b$, on convient que $\sum_{k=a}^{b} u_k = 0$. Ainsi, pour $i = 0$, on a bien : $S_0 = 0 = \sum_{k=0}^{-1} L[k]$.

Hérédité. Supposons la propriété vraie en un certain rang $i \in \{0, \dots, n-1\}$. Lors de la $(i+1)^{\text{e}}$ itération de la boucle, la variable `e` contient $L[i]$[4]. L'instruction `S += e` exécutée conduit à :

$S_{i+1} = S_i + L[i] = \sum_{k=0}^{i-1} L[k] + L[i] = \sum_{k=0}^{i} L[k]$. Ce qui achève la récurrence.

Comme la dernière itération de la boucle a lieu pour $i = n$, la valeur renvoyée par la fonction est bien : $S_n = \sum_{k=0}^{n-1} L[k]$ et la dernière ligne renvoie bien la moyenne des nombres contenus dans la liste `L`. Comme nous l'avons déjà conseillé, on écrit l'invariant dans le code sous forme de commentaires :

■■ Moyenne des éléments d'une liste

```
def moyenne(L: list)->float :
    """
    Calcule la moyenne des éléments de la liste de nombres L
    """
    S = 0
    for e in L:
        # Invariant: S_i = L[0]+...+L[i-1]
        S += e
    return S/len(L)
```

4. Les listes sont indicées à partir de 0.

3.3 Algorithme de HORNER

On s'intéresse à l'évaluation en un réel x d'une fonction polynomiale P à coefficients réels $(a_i)_{i\in\{0,1,\dots,n\}}$, n étant un entier naturel donné.

$$P(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0$$

Le *schéma de HORNER* organise les calculs en minimisant le nombre de multiplications.

$$P(x) = (((a_n x + a_{n-1})x + \cdots + a_2)x + a_1)x + a_0$$

Les coefficients étant stockés dans une liste $LC = [a_n, a_{n-1}, \dots, a_2, a_1, a_0]$, la fonction suivante met en oeuvre ce schéma.

```
def evalP(x: float, LC: list)->float :
    """Evalue en x le polynôme dont les coefficients sont donnés \
    ↪ dans la liste LC (par ordre de puissances \
    ↪ décroissantes)."""
    P = 0
    for c in LC :
        P = P * x + c
    return(P)
```

PREUVE DE CORRECTION. Pour prouver que cette fonction renvoie bien la valeur attendue, nous allons d'abord prouver, par récurrence simple, que la propriété suivante est un invariant de boucle :

$$\forall\, i \in \{0,\dots,\ell\},\quad \text{« } P_i = \sum_{k=0}^{i-1} LC[k] \times x^{i-1-k} \text{ »}$$

où ℓ désigne la longueur de la liste LC et i le numéro de l'itération.

Initialisation. De même que précédemment, pour $i = 0$, on a bien :

$$P_0 = 0 = \sum_{k=0}^{-1} LC[k] \times x^{-1-k}.$$

Hérédité. Supposons la propriété vraie en un certain rang $i \in \{0,\dots,\ell-1\}$. Lors de la $(i+1)^e$ itération de la boucle, la variable c contient $LC[i]$[5]. L'instruction `P = P*x+c` exécutée conduit à :

$$P_{i+1} = P_i \times x + LC[i] = \left(\sum_{k=0}^{i-1} LC[k] \times x^{i-1-k}\right) \times x + LC[i] = \sum_{k=0}^{i} LC[k] \times x^{i-k}.$$

Ce qui achève la récurrence.

5. Les listes sont indicées à partir de 0.

Comme la dernière itération de la boucle a lieu pour $i = \ell$, la valeur renvoyée par la fonction est bien :

$$\begin{aligned} P_\ell &= \sum_{k=0}^{\ell-1} LC[k] \times x^{\ell-1-k} \\ &= LC[0] \times x^{\ell-1} + LC[1] \times x^{\ell-2} + \cdots + LC[\ell-2] \times x + LC[\ell-1] \\ &= \boxed{a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0} \end{aligned}$$

si on note $LC = [a_n, a_{n-1}, \dots, a_1, a_0]$ et $\ell = n+1$.

3.4 Exponentiation rapide

Pour une boucle `while`, à la correction s'ajoute une étape préliminaire consistant à établir la terminaison de l'algorithme. On suppose cette étape validée.

Pour illustrer notre propos, établissons la correction de l'algorithme d'exponentation rapide.

```
def expR(x: float, n: int)-> float :
    """
    Renvoie x^n pour x réel et n entier naturel.
    """
    X = x
    N = n
    R = 1
    while N != 0 :
        if N%2 == 0 :
            N = N//2
        else :
            R = R*X
            N = (N-1)//2
        X = X*X
    return R
```

PREUVE DE CORRECTION. Montrons que la propriété suivante est un invariant de boucle, où ℓ désigne le nombre d'itérations de la boucle `while` :

$$\forall\, i \in \{0,\dots,\ell\},\quad \text{« } R_i \times X_i^{N_i} = x^n \text{ »}$$

Initialisation. Pour $i = 0$, on a $R_0 \times X_0^{N_0} = 1 \times x^n = x^n$.

Hérédité. Supposons la propriété vraie en un certain rang $i \in \{0,\dots,\ell-1\}$. Lors de la $(i+1)e$ itération de la boucle :

- si N_i est pair, alors les exécutions de `N = N/2` et `X = X*X` conduisent à $R_{i+1} = R_i$, $N_{i+1} = N_i//2$ et $X_{i+1} = X_i^2$ donc :
$$\begin{aligned} R_{i+1} \times X_{i+1}^{N_{i+1}} &= R_i \times (X_i^2)^{\frac{N_i}{2}} \\ &= R_i \times X_i^{N_i} \\ &= x^n. \end{aligned}$$
- si N_i est impair, alors les exécutions de `R = R*X`, `N = (N-1)/2` et `X = X*X` conduisent à $R_{i+1} = R_i \times X_i$, $N_{i+1} = \dfrac{N_i - 1}{2}$ et $X_{i+1} = X_i^2$ donc :
$$\begin{aligned} R_{i+1} \times X_{i+1}^{N_{i+1}} &= R_i \times X_i \times (X_i^2)^{\frac{N_i-1}{2}} \\ &= R_i \times X_i^{N_i} \\ &= x^n. \end{aligned}$$

Ce qui achève la récurrence. Comme la dernière itération de la boucle a lieu pour $i = \ell$, et que l'on a $N_\ell = 0$ sans quoi le programme continuerait à boucler, la valeur renvoyée par la fonction est bien :
$$R_\ell = R_\ell \times \underbrace{X_\ell^{N_\ell}}_{=1} = \boxed{x^n}$$

Exercice 2 **Partie entière** [Sol 2] Pour x réel positif, on rappelle que la partie entière de x est le plus grand entier naturel inférieur ou égal à x. La fonction suivante en effectue le calcul :

```
def ParEnt(x: float)->float :
    """ Calcule la partie entière du réel positif x """
    n = 0
    while n + 1 <= x :
        n += 1
    return n
```

Faire la preuve de cette fonction.

3.5 Et avec une fonction récursive ?

Comme on peut s'en douter, avec une fonction récursive on peut envisager une preuve (terminaison + correction) à l'aide d'un raisonnement par récurrence. Par exemple, soit la fonction :

```
def f(a: int, b: int)->int :
    """ Calcul récursif du pgcd, a et b sont supposés naturels \
    ↪ """
    if b == 0:
        return a
    else:
        return f(b, a%b)
```

On peut établir la terminaison et la correction en montrant par récurrence sur le paramètre b :

$\mathscr{P}(b)$ «$\forall a \in \mathbb{N}$, $f(a,b)$ se termine et renvoie pgcd(a,b)».

Initialisation. Il est clair que $\mathscr{P}(0)$ est vrai (c'est le cas terminal et pgcd$(a,0) = a$).

Hérédité. Supposons la propriété vraie pour tous les entiers jusqu'à un naturel b. Soit $a \in \mathbb{N}$. Lorsqu'on appelle $f(a, b+1)$, comme $b+1 \neq 0$, on renvoie la valeur de $f(b+1, r)$ où r est le reste de la division de a par $b+1$: $a = (b+1)q + r$. Or, $0 \leqslant r \leqslant b$ et on sait par hypothèse que $f(b+1, r)$ se termine et renvoie pgcd$(b+1, r)$, donc $f(a, b+1)$ se termine et renvoie pgcd$(b+1, r)$. Or, d'après le cours de mathématique, pgcd$(a, b+1)$ = pgcd$(b+1, r)$, donc $\mathscr{P}(b+1)$ est vraie, ce qui termine la récurrence.

3.6 Tout n'est pas si simple

Il y a des exemples de codes simples en apparence mais dont la preuve peut être très difficile. En voici un exemple : un théorème mathématique dit que tout nombre premier congru à 1 modulo 4 est une somme de deux carrés. La fonction suivante fournit une telle décomposition :

■■ Décomposition en somme de deux carrés

```
def decompPremier(p: int)-> (int,int):
    """ Renvoie deux entiers u et v tels que p = u^2+v^2
        p doit être un nombre premier congru à 1 modulo 4 """
    # fonction locale
    def f(a: int, b: int, c: int)->(int, int, int) :
        """ La fonction magique """
        if a > b+c:
            return (a-b-c, b, 2*b+c)
        else:
            return (b+c-a, a, 2*a-c)
    # corps de la fonction principale
    a, b, c = (p-1)//4, 1, 1
    while a != b:
        a, b, c = f(a, b, c)
```

```
    return (2*a,c)
>>> decompPremier(601)
(24, 5)
```

On a bien : $601 = 24^2 + 5^2$.

La terminaison de la boucle et la preuve de cet algorithme ne sont absolument pas triviales !

4 COMPLEXITÉ

4.1 Complexité temporelle

Étudier la *complexité temporelle* d'un algorithme permet d'en mesurer l'« efficacité » en terme de temps de calcul. Bien sûr, la durée d'exécution d'un programme dépend de la « taille » des données sur lesquelles il est appelé. Par exemple, on s'attend à ce qu'un programme calculant la moyenne d'une liste de nombres ait un temps de calcul d'autant plus long que la liste est longue. C'est précisément cette relation entre taille des données et durée d'exécution que va exprimer la complexité temporelle.

L'intérêt de cette évaluation est multiple. D'une part, connaître la complexité d'un algorithme recevant en entrée un certain volume de données permet d'en discuter la pertinence temporelle : le programme s'exécutera-t-il en un temps « raisonnable » ? D'autre part, plusieurs algorithmes différents peuvent résoudre un même problème. Une étude de leur complexité permet d'identifier le plus rapide d'entre eux, pour un une taille de données fixée.

Remarque 2 Dans ce cadre, la notion de complexité ne correspond pas à la difficulté ressentie à concevoir un algorithme. Un programme peut être très facile à écrire, mais avoir un temps d'exécution très long, alors qu'un autre programme beaucoup plus sophistiqué pourra être nettement plus efficace.

4.2 Méthode

Pour évaluer la complexité temporelle d'un algorithme, on commence par déterminer un entier naturel n mesurant la « taille » des données fournies au programme. Cet entier n'a pas besoin d'être la mesure précise en octets de l'espace mémoire nécessaire à stocker ces données, mais peut être une grandeur plus simple qui lui soit corrélée. Par exemple, si le programme doit traiter une liste, n peut être son nombre d'éléments.

On détermine ensuite, en fonction de la taille n des données, le nombre d'« instructions significatives » exécutées par le programme. Ces instructions peuvent dépendre du type d'algorithme étudié, et sont généralement des instructions élémentaires du langage : stockage d'une valeur dans une variable, comparaison de deux valeurs, addition, multiplication, etc.

Les durées d'exécution de chacune de ces instructions élémentaires ne sont pas identiques[6], mais on estime qu'elles restent d'un même « ordre de grandeur » . Le nombre d'opérations élémentaires effectuées par le programme est donc approximativement proportionnel à son temps d'exécution. Pour estimer le temps d'exécution, il faut estimer la durée d'une opération élémentaire, qui dépend de la puissance du microprocesseur et donc varie d'un ordinateur à un autre.

Se contenter de comptabiliser le nombre d'opérations élémentaires, et non leurs durées, permet donc de s'affranchir de l'ordinateur sur lequel le programme est exécuté : la complexité temporelle mesure réellement la performance de l'algorithme, et non celle de l'ordinateur sur lequel il est exécuté. Si un algorithme a une meilleur complexité qu'un autre, son temps d'exécution sera plus faible quelque soit l'ordinateur sur lequel on l'exécute.

Remarque 3 Notez bien que la complexité temporelle sera une fonction de n, exprimée sans unité puisqu'il s'agit d'un nombre d'instructions, et non en unité de temps comme le serait un temps d'exécution.

4.3 Un premier exemple

Intéressons-nous à la fonction qui permet de calculer la moyenne des éléments d'une liste de nombres donnée en argument.

■■ Moyenne des éléments d'une liste

```
def moyenne(L: list)->float :
    """Calcule la moyenne des éléments
    de la liste de nombres L"""
    S = 0
    for e in L:
        S += e
```

6. Le temps requis pour effectuer une multiplication est supérieur à celui d'une addition par exemple.

```
    return S/len(L)
```

Pour calculer la complexité temporelle de cette fonction, notons n le nombre d'éléments contenus dans la liste `L`. Le nombre d'affectations et d'opérations arithmétiques (additions et divisions) effectuées par le programme est :

- une affection avant la boucle `for`,
- dans la boucle `for` : une addition et une affectation, qui sont répétées pour les n éléments de la liste,
- une division dans la dernière ligne (on suppose que la fonction `len` s'effectue en un temps négligeable devant les autres.

La complexité temporelle de cet algorithme est :

$$C(n) = 1 + (1+1) \times n + 1 = \boxed{2n+2}$$

Attention

Pour les syntaxes ramassées `+=`, `-=`, `*=` *etc.*, pensez à bien décomposer dans le calcul de la complexité : 1 affectation et 1 opération (`+`, `-`, `*`).

Exercice 3 [Sol 3] Calculer la complexité des deux fonctions suivantes :

```
def f1(n: int)->int:
    x = 0
    for i in range(n):
        for j in range(n):
            x = x+1
    return x
def f2(n: int)->int:
    x = 0
    for i in range(n):
        for j in range(i):
            x += 1
    return x
```

4.4 Classification de la complexité

Les algorithmes sont classés suivant leurs complexités temporelles en les comparant à certaines formes de référence. Le tableau suivant présente plusieurs complexités usuelles et, pour différentes valeurs de n, une estimation du temps d'exécution correspondant si le processeur exécute chaque opération élémentaire en une nanoseconde ($1\text{ ns} = 10^{-9}$ s).

n	10	100	1000	10000	100000
$\ln n$	2 ns	5 ns	7 ns	9 ns	12 ns
n	10 ns	0.1 µs	1 µs	10 µs	0.1 ms
$n \ln n$	20 ns	0.5 µs	7 µs	90 µs	1 ms
n^2	0.1 µs	10 µs	1 ms	0.1 s	10 s
n^3	1 µs	1 ms	1 s	17 h	12 j
2^n	1 µs	3.10^{13} a	…	…	…

Un des enseignements de ce tableau est que pour certaines formes de complexité, la croissance du temps d'exécution est telle qu'il devient prohibitif même pour des valeurs assez faibles de n. C'est spectaculairement le cas pour la complexité 2^n dont la durée de calcul s'estime pour seulement $n = 100$ à trente mille milliards d'année[7] ! Un algorithme possédant cette complexité ne pourra donc être utilisé que pour de très petites données, ce qui en limite énormément l'intérêt pratique.

4.5 Notation de LANDAU

Pour comparer la complexité temporelle d'un algorithme aux expressions de référence, on utilise la relation de domination entre suites. Si (u_n) et (v_n) sont deux suites réelles, on note $u_n = O(v_n)$ si et seulement si on peut écrire, à partir d'un certain rang, $u_n = \alpha_n v_n$ où (α_n) est une suite bornée. Si la suite (v_n) ne possède aucun terme nul (ce qui sera le cas lorsqu'on manipulera des complexités), cela revient à :

$$\boxed{u_n = O(v_n) \iff \text{la suite } \left(\frac{u_n}{v_n}\right) \text{ est bornée.}}$$

Le plus souvent, la suite u sera polynomiale et on dira que u est un grand O de son plus grand monôme.

Exemple 2 (Fonction moyenne)

1. On a vu que la complexité temporelle de la fonction `moyenne` s'exprime par $C(n) = 2n+2$. Or $2n+2 = O(n)$; en effet $2n+2 = \frac{2n+2}{n} \times n$, avec $\frac{2n+2}{n}$ borné puisque converge vers 2. On dira alors que la fonction `moyenne` « a une complexité en $O(n)$ ».
2. On aurait pu tout aussi bien affirmer que la fonction `moyenne` a une complexité en $O(n^2)$. En effet $2n+2 = \frac{2n+2}{n^2} \times n^2$, avec $\frac{2n+2}{n^2}$ borné puisque converge vers 0. Mais cette information est moins intéressante que la pré-

7. À titre de comparaison, on estime l'âge de l'univers à 13 milliards d'années.

cédente puisque n^2 a une croissance plus rapide que n. Une classification pertinente consiste donc à comparer la complexité à une expression de référence ayant la croissance la plus faible possible.

3. À l'inverse, on ne pouvait pas dire que la fonction `moyenne` a une complexité en $O(\ln n)$ puisque $\frac{2n+2}{\ln n}$ tend vers $+\infty$ et donc n'est pas bornée.

$O(1)$	complexité constante	$O(n \ln n)$	complexité quasi-linéaire
$O(\ln n)$	complexité logarithmique	$O(n^k)$	complexité polynomiale
$O(n)$	complexité linéaire	$O(2^n)$	complexité exponentielle

Définition 3 | Vocabulaire

- *Déterminer la complexité exacte* d'un programme, c'est donner l'expression explicite du nombre d'opérations en fonction de la taille des données.
- *Déterminer la complexité asymptotique* d'un programme, c'est donner le nombre d'opérations en fonction de la taille des données sous la forme d'un grand O.

Exercice 4 **Listes de Césaro** [Sol 4] Étant donnée une liste de réels $u = [u_0, u_1, \dots, u_n]$, on appelle liste de Césaro associée la liste $v = [v_0, v_1, \dots, v_n]$ dont chaque terme est égal à la moyenne des premiers termes de u :

$$v_0 = \frac{u_0}{1}, \quad v_1 = \frac{u_0 + u_1}{2}, \quad \dots \quad v_n = \frac{u_0 + u_1 + \cdots + u_n}{n+1}.$$

Le terme v_k apparaît ainsi comme la moyenne des termes $u_0, \dots, u_k$ de la suite u. La fonction suivante prend comme argument une liste `u` et renvoie la liste correspondante pour la suite v associée.

```
def cesaro(u: list)->list :
    """Calcul de la liste de Césaro associée à la liste u"""
    v = []
    for k in range(len(u)) :
        m = moyenne(u[:k+1])
        v += [m]
    return v
```

1. Calculer la complexité temporelle de la fonction `cesaro` en fonction du dernier indice n de la liste `u`, et classifier celle-ci.

2. Écrire une autre version de la fonction `cesaro` ayant une complexité temporelle « significativement meilleure » .

Exercice 5 **Cas d'un while** [Sol 5] On considère la fonction ci-après :

```
def compte_it(n:int)->int:
    i = n
    x = 0
    while i > 1:
        i = i//2
        x += 1
    return x
```

1. Prouver la terminaison de cette fonction.
2. Justifier l'existence et l'unicité de p entier tel que : $2^p \leqslant n < 2^{p+1}$.
3. Donner et prouver un invariant sur i_k de la forme $i_k \in [a_k, b_k[$ où a_k, b_k dépendent de k et p.
4. En déduire la complexité temporelle de `compte_it`. Quel est le rôle de x ?

Dans l'exercice précédent, x compte le nombre d'appels récursifs.

4.6 Complexité dans le meilleur ou dans le pire des cas

Considérons maintenant la fonction `nbPositifs` suivante, qui renvoie le nombre d'éléments strictement positifs d'une liste de nombres.

```
def nbPositifs(lst: list)->int :
    """Nombre d'éléments positifs d'une liste"""
    nb = 0
    for e in lst:
        if e > 0 :
            nb = nb + 1
    return nb
```

Pour en calculer la complexité en fonction de la taille n de la liste, on est confronté à une difficulté : l'affection `nb = nb+1` de la ligne 6 n'est effectuée que si la condition `e > 0` est vraie, ce qui dépend de la liste fournie. Autrement dit, contrairement aux exemples précédents où le nombre d'opérations élémentaires ne dépendait que de la taille des données, ici ce nombre peut varier entre deux données de même taille. On introduit alors les notions de complexité dans le meilleur et dans le pire des cas, consistant à calculer respectivement le nombre minimal et maximal d'opérations élémentaires parmi l'ensemble des données de taille n. Ces deux valeurs fournissent alors un encadrement de la complexité pour une donnée quelconque de taille n.

Sur notre exemple, la complexité dans le meilleur des cas est obtenue lorsque l'affectation `nb = nb+1` n'est jamais effectuée, ce qui est le cas lorsqu'il n'y a pas d'éléments positifs. On dénombre alors les opérations élémentaires :

- une affectation pour l'instruction `nb = 0`,
- pour chacun des n éléments de `liste`, un test `e > 0`, mais aucune affectation `nb = nb+1`.

On trouve donc : $C_{\text{meilleur}}(n) = 1 + n \times 1 = n + 1$.

Le calcul de la complexité dans le pire des cas (une liste dont les termes sont strictement positifs) se fait bien sûr en ajoutant une addition et une affectation `nb = nb+1` pour chacun des n éléments de `liste`, soit $C_{\text{pire}}(n) = 1 + n \times 3 = 3n + 1$.

Remarquons qu'ici la complexité dans le meilleur et dans le pire des cas sont tous deux en $O(n)$.

Exercice 6 [Sol 6] Considérons la fonction suivante qui permet de rechercher un indice à partir duquel la somme des éléments d'une liste de nombres positifs dépasse un seuil fixé. Le programme renvoie -1 si cet indice n'existe pas.

```
def depasse(L: list, seuil: float)->int:
    """Renvoie l'indice où la somme des éléments de L dépasse \
    ↪ seuil, -1 en cas de non-existence"""
    S = 0
    trouve = False
    k = 0
    while not(trouve) and (k < len(L)):
        S = S + L[k]
        if S > seuil:
            trouve = True
        else:
             k += 1
    if not(trouve):
            k = -1
    return k
```

Calculer la complexité de cette fonction dans le meilleur et dans le pire des cas.

4.7 Complexité d'une fonction récursive

Une fonction récursive s'appelant elle-même, le calcul de sa complexité temporelle se fait naturellement par récurrence. Pour mieux le comprendre, intéressons-nous à la fonction `FactorielleRec` suivante, qui calcule la factorielle d'un entier naturel n en récursif.

```
def FactorielleRec(n: int)->int :
    if n == 0:
        return 1
    else:
        return n*FactorielleRec(n-1)
```

Notons $C(n)$ la complexité temporelle de cette fonction lorsque le paramètre d'entrée est n. On trouve $C(0) = 1$ (car si $n = 0$, on compare juste n à 0) et $C(n+1) = 2 + C(n)$ (on fait une comparaison à 0, une multiplication puis $C(n)$ opérations élémentaires en appelant la fonction avec le paramètre n). On obtient donc une suite arithmétique de raison 2, ce qui donne après calculs $C(n) = 1 + 2n = O(n)$.

Exercice 7 [Sol 7] Considérons la fonction `SuiteU` récursive suivante :

```
def SuiteU(n: int)->float :
    if n == 0:
        return 1
    else:
        return 2*SuiteU(n-1)+1/SuiteU(n-1)
```

1. Que calcule cette fonction ?
2. Calculer sa complexité temporelle.
3. Proposer une fonction `SuiteU2` ayant une meilleure complexité.

4.8 Complexité spatiale

De la même façon que l'on définit la complexité temporelle d'un algorithme pour évaluer sa performance en temps de calcul, on peut définir sa *complexité spatiale* pour évaluer sa consommation en espace mémoire. Le principe est le même sauf qu'au lieu de compter les opérations élémentaires, on compte les entités élémentaires de mémoire allouée pour l'exécution du programme, toujours en fonction de la taille n des données. Ces entités élémentaires de mémoire sont celles qui stockent les valeurs de type élémentaires (entier, flottant, caractère, etc). Un type complexe comme une liste de réels, par exemple, comptera pour autant d'entité élémentaires qu'elle contient de réels. Notez qu'ici encore il s'agit de compter des entités élémentaires, et pas de mesurer la mémoire utilisée. La complexité spatiale sera exprimée sans unité, et pas en octets. Cependant, on notera que la complexité spatiale est bien moins que la complexité temporelle un frein à l'utilisation d'un algorithme : on dispose aujourd'hui le plus souvent d'une quantité pléthorique de mémoire vive, ce qui rend moins important la détermination de la complexité spatiale.

SOLUTIONS DES EXERCICES

Solution 1

1. La fonction calcule x^n si n est un entier naturel, cette fonction contient une unique boucle `for`, elle se termine donc forcément.

2. La fonction calcule également x^n si n est un entier naturel. La suite (i_k) des valeurs contenues dans la variable i vérifie $i_0 = 0$ et $i_{k+1} = i_k + 1$. On a donc $i_k = k$ pour $k \geqslant 0$. Cette suite d'entiers étant strictement croissante, il existe un rang k_0 tel que $i_{k_0} \geqslant n$, ce qui assure la terminaison de la boucle (sinon on aurait une suite d'entiers strictement croissante et majorée par n).

Solution 2

1. Prouvons d'abord la terminaison de la boucle `while`. Pour cela, raisonnons par l'absurde et supposons que celle-ci ne se termine pas. Montrons alors l'invariant de boucle :

$$\forall\, i \in \mathbb{N}, \quad \text{«}n_i \in \mathbb{N}\text{»}.$$

Initialisation. Pour $i = 0$, on a $n_0 = 0 \in \mathbb{N}$.

Hérédité. Supposons la propriété vraie en un certain rang $i \in \mathbb{N}$. Lors de la $(i+1)^e$ itération de la boucle `while`, l'exécution de `n += 1` conduit à $n_{i+1} = n_i + 1 \in \mathbb{N}$. Ce qui achève la récurrence.

D'autre part, la suite $(n_i)_{i\in\mathbb{N}}$ est strictement croissante puisque pour tout $i \in \mathbb{N}$ on a $n_{i+1} = n_i + 1 > n_i$.

La suite $(n_i)_{i\in\mathbb{N}}$ est formée d'entiers et est strictement croissante, donc il existe un rang i_0 pour lequel elle sera strictement supérieure à $x-1$. On aura alors $n_{i_0}+1 > x$ et la boucle `while` s'arrête, contrairement à l'hypothèse.

Ce qui achève la preuve de terminaison de la boucle `while`.

Reste à montrer que le programme renvoie bien le résultat attendu. Pour cela, montrons l'invariant de boucle, où ℓ désigne le nombre d'itérations de la boucle `while` :

$$i \in \{0, \dots, \ell\}, \quad \text{«}n_i \leqslant x\text{»}.$$

Initialisation. Pour $i = 0$, on a $n_0 \leqslant x$ puisque $n_0 = 0$ et que x est positif par hypothèse.

Hérédité. Supposons la propriété vraie en un certain rang $i \in \{0, \dots, \ell - 1\}$. Lors de la $(i+1)^e$ itération de la boucle `while`, comme on est entré dans cette boucle, c'est que la condition exprimée dans le `while` était vraie, à savoir que $n_i + 1 \leqslant x$. Or $n_{i+1} = n_i + 1$, donc on a bien $n_{i+1} \leqslant x$.

Ce qui achève la récurrence.

Comme la dernière itération de la boucle `while` a lieu pour $i = \ell$, on a à la fois $n_\ell \leqslant x$ d'après la propriété précédente, et $n_\ell + 1 > x$ puisque la boucle s'arrête à cette étape. Ainsi, la valeur n_ℓ renvoyée par la fonction est bien le plus grand entier naturel inférieur ou égal à x.

Solution 3

- Pour la fonction $f1$, on commence par une affectation à la ligne 2. On a ensuite deux opérations élémentaires à la ligne 5 répétées n fois dans la boucle `for` de la ligne 4, soit $2n$ opérations élémentaires pour les lignes 4 et 5. Ces $2n$ opérations sont répétées n fois chacune dans la boucle `for` de la ligne 3, soit $2n^2$ opérations élémentaires pour les lignes 3 à 5. Au total, on a donc une complexité de $f1$ égale à $1 + 2n^2$.
- Pour la fonction $f2$, on commence également par une affectation à la ligne 2. On a ensuite deux opérations élémentaires à la ligne 5 répétées i fois dans la boucle `for` de la ligne 4, soit $2i$ opérations élémentaires pour les lignes 4 et 5. Avec la boucle `for` de la ligne 3, on a donc :

$$2*1 + 2*2 + 2*3 + 2*4 + \dots + 2*(n-1) = 2 \times \frac{(n-1)n}{2} = (n-1)n$$

opérations élémentaires pour les lignes 3 à 5. Au total, on a donc une complexité de $f2$ égale à $1 + (n-1)n$.

Solution 4

1. On dénombre :

- une affectation `v = []` ;
- pour l'indice k de la boucle `for` (k allant de 0 à n), l'appel `moyenne(u[:k+1])` sur une liste de longueur $k+1$ se fait en $2 + 2(k+1)$ instructions élémentaires auxquelles s'ajoute l'affectation dans la variable `m` et la ligne suivante contient deux instructions élémentaires (une concaténation et une affectation). On compte donc $7 + 2k$ instructions élémentaires pour l'indice k.

Ainsi, la complexité temporelle est :

$$C(n) = 1+\sum_{k=0}^{n}(7+2k) = 1+7\sum_{k=0}^{n}1+2\sum_{k=0}^{n}k = 1+7(n+1)+2\frac{n(n+1)}{2} = \boxed{8+8n+n^2}.$$

Comme $\dfrac{8+8n+n^2}{n^2}$ converge vers 1, cette fonction a une complexité en $\boxed{O(n^2)}$.

2. Pour éviter de re-sommer à chaque fois les premiers termes de la suite, on peut constater que l'on a la relation suivante :

$$\forall k \in [\![0\,,\,n]\!], \quad v_k = \frac{S_k}{k+1}, \text{où : } S_k = u_0+\cdots+u_k.$$

La variable S_k pouvant être créée à l'aide d'une boucle `for`.

```
def cesaro2(u: list)->list :
    """Calcul de la liste de Césaro associée à la liste u"""
    v = []
    S = 0
    for k in range(len(u)) :
        S = S + u[k]
        v = v + [S/(k+1)]
    return v
```

Calculons la nouvelle complexité :

- deux affectations pour `v = []` et `S = 0` ;
- pour chacune des $n+1$ itérations de la boucle `for`, une somme et une affectation pour `S = S+u[k]` et 4 opérations élémentaires (une somme, une division, une somme de listes et une affectation.

On trouve donc une complexité en $\boxed{O(n)}$ puisque :

$$C(n) = 2+6(n+1) = \boxed{8+6n}.$$

Solution 5

1. Supposons que le `while` ne s'arrête pas. Alors la suite $(i_k)_{k\in\mathbb{N}}$ est une suite strictement décroissante d'entiers. En effet, soit $k \in \mathbb{N}$:

- si i_k est pair, alors $i_{k+1} = \dfrac{i_k}{2} < i_k$ puisque $i_k \neq 0$ (en effet, si on rentre dans la boucle alors c'est que $i_k > 1$).
- Si i_k est impair, alors $i_{k+1} = \dfrac{i_k-1}{2} < i_k$.

Ainsi, puisque $(i_k)_{k\in\mathbb{N}}$ est une suite strictement décroissante d'entiers, il existe un rang à partir duquel $i_k \leqslant 1$ — **Contradiction**.
La boucle termine.

2. On reformule en passant au $\log_2$:

$$\begin{aligned} 2^p \leqslant n < 2^{p+1} &\iff p \leqslant \log_2(n) < p+1 \\ &\iff \log_2(n)-1 < p \leqslant \log_2(n). \end{aligned}$$

On reconnait là la définition de la partie entière : $\boxed{p = \lfloor \log_2(n) \rfloor}$.

3. On peut montrer que : $2^{p-k} \leqslant i_k < 2^{p+1-k}$ pour tout $k \in \{0,\dots,N\}$ où N désigne le nombre d'itérations de la boucle `while`.

Initialisation. Pour $k = 0$, puisque $i_0 = n$ on applique simplement la question précédente.

Hérédité. Supposons que $2^{p-k} \leqslant i_k < 2^{p+1-k}$ pour $k \in \{0,\dots,N-1\}$ fixé. Il y a donc une itération $k+1$. On fait là encore deux cas :

- si i_k est pair, alors $i_{k+1} = \dfrac{i_k}{2}$ donc d'après l'invariant :

  $$2^{p-k-1} \leqslant i_{k+1} = \frac{i_k}{2} < 2^{p-k}.$$

  C'est exactement ce que l'on voulait montrer.
- Si i_k est impair, alors $i_{k+1} = \dfrac{i_k-1}{2}$, donc d'après l'invariant :

  $$2^{p-k-1} - \frac{1}{2} = \frac{2^{p-k}-1}{2} \leqslant i_{k+1} = \frac{i_k-1}{2} < \frac{2^{p+1-k}-1}{2} = 2^{p-k} - \frac{1}{2} < 2^{p-k}.$$

  Comme i_{k+1} est un entier, l'encadrement donne aussi :
  $2^{p-k-1} \leqslant i_{k+1} < 2^{p-k}$.

L'invariant est donc prouvé par principe de récurrence.

4. L'invariant précédent donne $i_p \in [1,2[$ alors que i_{p-1} ne peut valoir 1. Ainsi, le nombre d'itérations du `while` est exactement $p = \lfloor \log_2(n) \rfloor$. Faisons maintenant le bilan des opérations : on a $C(n) = 2+5p = 2+5\lfloor \log_2(n) \rfloor = O(\log_2(n))$. Les 5 opérations venant des 4 affectations/addition/quotient et du test d'entrée dans la boucle.

Solution 6

Là encore, le nombre d'opérations élémentaires ne dépend pas uniquement de la taille n de la liste fournie. Dans le meilleur des cas, le premier élément est supérieur à seuil et la boucle `while` est répétée une seule fois. Les opérations élémentaires sont :

- trois affectations avant la boucle ;
- trois tests `not(trouve)`, `k < len(liste)` (condition du `while`), `S > seuil` (condition du `if`), une addition, une affectation, et une affectation `trouve = True` ; le tout est effectué une seule fois, mais n'oublions pas un dernier test `not(trouve)` pour déterminer que la boucle s'arrête (l'autre test `k < len(liste)` n'est pas effectué puisque le précédent est faux) ;
- un test `if not(trouve)`.

Ainsi, $C_{\text{meilleure}}(n) = 11$. Ici, la complexité dans le meilleur des cas est indépendante de la taille de la liste, elle est donc en $O(1)$.

Le pire des cas s'obtient quand la somme des éléments de la liste est inférieure au seuil. Il faut alors parcourir la liste en entier :

- toujours trois affectations avant la boucle ;
- les trois tests, deux affectations et deux additions sont répétés n fois, puis les deux tests `not(trouve)` et `k < len(liste)` une dernière fois pour déterminer que la boucle s'arrête ;
- puis enfin un test et une affectation `k = -1`

On trouve donc $C_{\text{pire}}(n) = 3 + (3+2+2)n + 2 + 2 = 7n + 7$. La complexité dans le pire des cas est donc en $O(n)$.

Solution 7

1. Cette fonction permet de calculer le terme d'indice n de la suite (u_n) définie par : $u_0 = 1$ et $\forall n \in \mathbb{N}$, $u_{n+1} = 2u_n + \dfrac{1}{u_n}$.
2. Notons $C(n)$ la complexité de cette fonction lorsque le paramètre d'entrée est n. On trouve $C(0) = 1$ et $C(n+1) = C(n) + C(n) + 4$ (on fait une multiplication, une addition, une division et $2 \times C(n)$ opérations élémentaires en appelant deux fois la fonction avec le paramètre n). On obtient donc une suite arithmético-géométrique ce qui donne après calculs $C(n) = 5 \times 2^n - 4$.
3. Considérons la fonction suivante :

```
def SuiteU2(n:int)->float :
    if n == 0:
        return 1
    else:
        a = SuiteU2(n-1)
        return 2*a+1/a
```

On trouve $C'(0) = 1$ (seulement un test lorsque $n = 0$). Si $n \geq 1$, $C'(n) = 5 + C'(n-1)$ (on fait un test, une affectation, une multiplication, une addition, une division et $C'(n-1)$ opérations élémentaires en appelant la fonction avec le paramètre $n-1$). On obtient donc une suite arithmétique qui donne après calculs $C'(n) = 5n + 1$. On obtient une complexité linéaire : le gain est énorme !