Chapitre IV
Algorithmique

Table des matières

Dans ce chapitre, nous allons nous poser des questions sur les algorithmes que nous rencontrons ou créons, en particulier, trois questions fondamentales étant donné un algorithme :

- **L'algorithme se termine-t-il ?**
- **L'algorithme est-il efficace ?**
- **L'algorithme donne-t-il le résultat attendu ?**

Partie A

Terminaison

1. Introduction

Comme annoncé en introduction du chapitre, nous allons apprendre à répondre à la question de la terminaison d'un algorithme... qui d'ailleurs, par essence, doit se terminer pour avoir droit à ce nom ! Commençons par quelques exemples simples :

Il est aisé d'écrire un "algorithme" qui ne se termine pas :

```
a = 0
Tant que a >= 0
    a+=1
Fin Tant que
```

La condition "`a >= 0`" étant toujours vérifiée, le boucle est infinie et donc ne se termine pas.

Maintenant, si on considère alogrotihme avec une boucle `Pour ... allant de a à b`, on peut être sûr qu'il se termine : on connaît a priori le nombre d'itération de la boucle : `b-a+1` itérations. L'algorithme se termine donc !

Remarque 1.

Il faut tout de même faire attention, dans le cas de Python et de l'implémentation de ses listes, on peut avoir des boucles `for` qui ne se terminent pas :

Boucle for infinie avec Python

```
1 L=[0]
2 for element in L:
3     L.append(0)
```

A chaque itération, un élement est ajouté en fin de liste et la liste étant parcouru élément par élément dans l'ordre, la boucle devient infinie !
On peut simplement éviter ce comportement :

```
L=[0]
for i in range(len(L)):
    L.append(0)
```

La valeur `len(L)` étant calculée une fois pour toute au début de la boucle, on se retrouve dans le cas d'une boucle `Pour ... allant de a à b` décrit plus haut.

On se concentrera donc dans la suite à l'étude de la terminaison des boucles conditionnelles `Tant que`.

2. Variant de boucle

Dans cette partie, on considère un algorithme constitué d'une boucle conditionnelle.

Définition 1. *Variant de boucle*

On appelle **variant de boucle** une quantité dépendante des variables en jeu et qui est **entière**, **positive** au début de chaque itération de la boucle et qui **décroit strictement** à chaque itération.

Lorsqu'on parle d'une *itération* d'une boucle conditionnelle, on sous-entend que la condition associée est vraie !

Proposition 1.

Si une boucle possède un variant de boucle, alors cette boucle se termine.

Exemple 1.

On considère l'algorithme suivant, où l'argument `n` est un réel plus grand ou égal à 1.

```
def pgp2(n):
    p=1
    while p <= n:
        p = p*2
    return p
```

Cet algorithme renvoie la première puissance de 2 strictement plus grande que n.

Considérons la quantité $\lfloor n \rfloor - p$ dans l'algorithme précédent. Voyons comment montrer qu'il s'agit d'un variant de boucle :

On remarque tout d'abord que la variable p étant initialisée à 1 et qu'elle double à chaque itération, donc ses valeurs sont entières et $p \geq 1$; ensuite :

— Avant la boucle, comme $n \geq 1$ et $p = 1$, la quantité vaut $\lfloor n \rfloor - 1 \geq 0$; la boucle est donc bien initialisée ;

— Au début de chaque l'itération, la condition étant vraie, on a $p \leq n$, donc p étant entier, $p \leq \lfloor n \rfloor$, d'où la quantité $\lfloor n \rfloor - p$ est un entier positif.
— La quantité valant $\lfloor n \rfloor - p$ au début d'une itération ; à la fin de celle-ci, la quantité prend la valeur $\lfloor n \rfloor - 2p$;
or comme $p \geq 1$, on a $p < 2p$, d'où

$$\lfloor n \rfloor - 2p < \lfloor n \rfloor - p$$

Ainsi, la quantité décroit strictement à chaque itération.
Par suite, $\lfloor n \rfloor - p$ est bien un **variant de boucle** et donc la boucle se termine.

Remarque 2.

Si on exhibe un variant de boucle v, alors, une fois la boucle terminée, $v \leq 0$. Donc lorsqu'on veut montrer la positivité d'un variant, il faut se placer pendant une itération i.e. tant que la condition de la boucle est vraie.

3. Variant récursif

Pour montrer qu'un algorithme récursif se termine, on va utiliser raisonner de la même manière que pour les boucles conditionnels, en cherchant un variant récursif dont la définition est très proche de celle de variant de boucle :

Définition 2. *Variant récursif*

On appelle **variant récursif** d'une fonction récursive une quantité dépendante des arguments successifs à chaque appel récursif de la fonction, qui est **entière**, **positive** et qui **décroit strictement** à chaque appel récursif.

Proposition 2.

Si une fonction récursive possède un variant récursif, alors l'algorithme récursif associé se termine.

a. Récursivité simple

Exemple 2.

On considère l'algorithme suivant, où l'argument `n` est un réel plus grand ou égal à 1.

```python
def pgp2_rec(n):
    if n<2:
        return 2
    else:
        return 2*pgp2_rec(n-1)
```

Cet algorithme renvoie (encore !) la première puissance de 2 strictement plus grande que n.

Considérons la quantité $a = \lfloor n \rfloor - 1$ où n est la valeur de l'argument de `pgp2_rec` à chaque appel récursif. Voyons comment montrer qu'il s'agit d'un variant récursif :

Lors de l'appel initial `pgp2_rec(n)`, a vaut $\lfloor n \rfloor - 1$ qui est un entier positif car $n \geq 1$.

- Si $n < 2$, la fonction renvoie 2 donc l'algorithme se termine.
- Si $n \geq 2$, on a l'appel récursif `pgp2_rec(n-1)` d'où a vaut $\lfloor n-1 \rfloor - 1 = \lfloor n \rfloor - 2 < \lfloor n \rfloor - 1$ qui est la valeur précédente de a donc la quantité a décroit strictement à chaque appel récursif.

Par suite, a est bien un **variant récursif** et donc l'algorithme se termine.

b. Récursivité multiple

Exemple 3.

On considère l'algorithme suivant, où l'argument `n` est un entier naturel

```python
def fibo(n):
    if n<=1:
        return 1
    else:
        return fibo(n-1)+fibo(n-2)
```

Considérons la quantité $arg = n$ où n est la valeur de l'argument de `fibo` à chaque appel récursif. Voyons comment montrer qu'il s'agit d'un variant récursif :

Lors de l'appel initial `fibo(n)`, arg vaut n qui est un entier positif.

- Si $n \leq 1$, la fonction renvoie 1 donc l'algorithme se termine.
- Si $n > 1$, n étant entier, $n \geq 2$ et on a :
  - pour l'appel récursif `fibo(n-1)` ; arg vaut $n - 1 < n$ qui est la valeur précédente de arg donc la quantité arg est positive ($n - 1 \geq 1$) et décroit strictement pour cet appel récursif.
  - pour l'appel récursif `fibo(n-2)` ; arg vaut $n - 2 < n$ qui est la valeur précédente de arg donc la quantité arg est positive ($n - 2 \geq 0$) et décroit strictement pour cet appel récursif.

  Dans le deux cas, arg est un entier positif qui décroit strictement par rapport à l'appel précédent.

Par suite, arg est bien un **variant récursif** et donc l'algorithme se termine.

4. Exercices

Exercice 1.

Déterminer si les algorithmes suivants se terminent.

1. Algorithme 1 : l'argument `n` est un entier positif.

```
def f1(n):
    a=0
    while a <= n:
        a += 2
    return a
```

2. Algorithme 2 : l'argument `n` est un entier positif.

```
def f2(n):
    x=0
    while x <= n:
        x -= 2
    return x
```

3. Algorithme 3 : l'argument `n` est un entier positif.

```
def f3(n):
    a=0
    for i in range(n):
        a = n**2+a-1+30*i
    return a
```

4. Algorithme 4 : les arguments `a,b` sont des entiers strictement positifs.

```
def reste(a,b):
    r=a
    while r >= b:
        r = r-b
    return r
```

5. Algorithme 5 : l'argument `L` est une liste de nombres **triée** et `x` est un nombre.

```python
def recherche_dicho(x,L):
    d,f = 0,len(L)-1
    while d<f:
        m = (d+f)//2
        if L[m] == x:
            return True
        elif L[m] < x:
            d = m+1
        else:
            f = m-1
    return False
```

Correction.

1. On considère la quantité $v = n - a$. Comme la variable a est initialisée à 0 et augmente de 2 à chaque itération, la variable a est à valeurs entières (positives) et par hypothèse, n est à valeurs entières (positives). Par suite, $v = n - a$ est un entier.
   — Avant la boucle, on a $n \geq 0 = a$ donc la boucle est initialisée et on a $v = n - a = n$ est un entier positif.
   — Au début de chaque itération, la condition $a \leq n$ étant vraie, on a $v = n - a$ positif. Ainsi, au début de chaque itération, v est un entier positif.
   — La quantité valant $n - a$ au début d'une itération, à la fin de cette même itération, on a $v = n - (a + 2) = n - a - 2 < n - a$ donc v décroît strictement à chaque itération.
   Ainsi, v est un variant de boucle et donc l'algorithme se termine.

2. La variable x est initialisée à 0 est diminue de 2 à chaque itération donc $x \leq 0$. Or, par hypothèse, $n \geq 0$, donc on a pendant toute l'exécution $x \leq 0 \leq n$. Donc la boucle ne se termine pas.

3. Il s'agit d'une boucle `for` qui fait exactement n itérations donc l'algorithme se termine.

4. On considère la quantité $v = r - b$. Comme la variable r est initialisée à a entier et diminue de b entier à chaque itération, la variable r est à valeurs entières et par hypothèse, b est à valeurs entières. Par suite, $v = r - b$ est un entier.
   — Avant la boucle, on a $r = a$ donc si $a < b$ la boucle n'est pas initialisée et donc l'algorithme se termine ; sinon, si $a \geq b$ la boucle est initialisée et on a $v = r - b = a - b$ est un entier positif.
   — Au début début de chaque itération, la condition $r \geq b$ étant vraie, on a $v = r - b$ positif. Ainsi, au début de chaque itération, v est un entier positif.
   — La quantité valant $r - b$ au début d'une itération, à la fin de cette même itération, on a $v = (r - b) - b = r - 2b < r - b$ car $b > 0$ donc v décroît strictement à chaque itération.
   Ainsi, v est un variant de boucle et donc l'algorithme se termine.

5. On considère la quantité $v = f - d$. Comme les variables d et f sont initialisées en des valeurs enières (0 et la longueur de `L` -1) et qu'elles sont potentiellement incrémentée ou décrémentée de 1 à chaque itération, elles sont à valeurs entières, d'où v est un entier.
   — Avant la boucle, si $d \geq f$ alors la boucle n'est pas initialisée donc l'algorithme se termine ; sinon, si $d < f$, la boucle est initialisée et on a $v = f - d$ est un entier positif.

— Au début début de chaque itération, la condition $d < f$ étant vraie, on a $v = f - d$ positif. Ainsi, au début de chaque itération, v est un entier positif.
— La quantité valant $f - d$ au début d'une itération, à la fin de cette même itération, on a :
- si `L[m]` vaut `x`, on retourne `True` donc l'algorithme se termine ;
- si `L[m]<x`, alors $v = f - (d+1) = f - d - 1 < f - d$
- si `L[m]>x`, alors $v = (f-1) - d = f - d - 1 < f - d$

donc dans tous les cas où l'algorithme ne se termine pas directement, v décroît strictement à chaque itération.

Ainsi, v est un variant de boucle et donc l'algorithme se termine.

Exercice 2.

Déterminer si les algorithmes suivants se terminent.

1. Algorithme 1 : l'argument `n` est un entier positif.

```
def f1(n):
    if n==0:
        return 1
    else:
        return f1(n-1)*n
```

2. Algorithme 2 : l'argument `n` est un entier positif.

```
def f2(n):
    if n==1:
        return 0
    else:
        return f2(n)
```

3. Algorithme 3 : les arguments `a,b` sont des entiers positifs.

```
def pgcd(a,b):
    if b==0:
        return a
    else:
        return pgcd(b,a%b)
```

4. Algorithme 4 : les arguments `n,m` sont des entiers relatifs.

```python
def diff(n,m):
    if n==m:
        return 0
    elif n<m:
        return diff(n+1,m)+1
    else:
        return diff(n,m+1)+1
```

5. Algorithme 5 : l'argument `L` est une liste de nombres **triée** et `x` est un nombre.

```python
def recherche_dicho(x,L,d,f):
    if f-d<=1:
        return False
    else:
        m=(d+f)//2
        if L[m]==x:
            return True
        elif L[m]<x:
            return recherche_dicho(x,L,d,m)
        else:
            return recherche_dicho(x,L,m+1,f)
```

6. Algorithme 6 : les arguments `n,p` sont des entiers positifs.

```python
def c(n,p):
    if p == 0:
        return 1
    elif n==p:
        return 1
    return c(n-1,p-1)+c(n-1,p)
```

Correction.

1. On considère la quantité $a = n$ où n est la valeur de l'argument de `f1` à chaque appel récursif. Lors de l'appel initial `f1(n)`, a vaut n qui est un entier positif par hypothèse.
   — Si $n = 0$, la fonction renvoie 1 donc l'algorithme se termine.
   — Si $n > 0$, alors comme n est un entier, $n \geq 1$. Ainsi, lors de l'appel récursif `f1(n-1)`, on a $a = n - 1 \geq 0$ et $a = n - 1 < n$ qui est la valeur précédente de a. Ainsi a est une quantité entière positive qui décroit strictement à chaque appel récursif.
   Par suite, a est un variant récursif et donc l'algorithme se termine.

2. On remarque que si $n >\geq< 2$, alors l'appel de `f2(n)` conduit de nouveau à un même appel de `f2(n)` donc la suite des arguments des appels successifs est constante en $n > 1$. Ainsi, la condition initiale $n = 1$ n'est jamais réalisée. Cet algorithme ne se termine donc pas.

3. On considère la quantité $arg = b$ où b est la valeur du deuxième argument de `pgcd` à chaque appel récursif. Lors de l'appel initial `pgcd(a,b)`, arg vaut b qui est un entier positif par hypothèse.
   — Si $b = 0$, la fonction renvoie a donc l'algorithme se termine.
   — Si $b > 0$, alors comme b est un entier, $b \geq 1$. Ainsi, lors de l'appel récursif `pgcd(b,a%b)`, et si on note $r =$`a%b` (reste de la division euclidienne de a par b) on a $arg = r$ et donc, par définition du reste, arg est un entier tel que $0 \leq arg < b$ où b est la valeur précédente de arg. Ainsi arg est une quantité entière positive qui décroit strictement à chaque appel récursif.

   Par suite, arg est un variant récursif et donc l'algorithme se termine.

4. On considère la quantité $a = |n - m|$ où n, m sont les valeurs des arguments de `diff(n,m)` à chaque appel récursif. Lors de l'appel initial `diff(n,m)`, a vaut $|n - m|$ qui est positif et entier car n et m le sont par hypothèse.
   — Si $n = m$, la fonction renvoie 0 donc l'algorithme se termine.
   — Sinon :
   - si $n < m$ et donc $n + 1 \leq m$ (car n, m entiers), alors avant l'appel de `diff(n+1,m)`, a vaut $|n - m| = m - n$ et lors de cet appel, $a = |(n + 1) - m| = m - (n + 1) = m - n - 1 < m - n$ donc a est un entier positif et décroît strictement pour cet appel
   - si $n > m$ et donc $n \geq m + 1$ (car n, m entiers), alors avant l'appel de `diff(n,m+1)`, a vaut $|n - m| = n - m$ et lors de cet appel, $a = |n - (m + 1)| = n - (m + 1) = n - m - 1 < n - m$ donc a est un entier positif et décroît strictement pour cet appel

   Dans tous les cas, a est entier positif et décroît strictement à chaque appel récursif.

   Par suite, a est un variant récursif et donc l'algorithme se termine.

Partie B

Complexité temporelle

1. Introduction

Étant donné un algorithme, on aimerait savoir si celui-ci est "efficace" : si on implémente cet algorithme sur une machine (si on crée un programme correspondant à cet algorithme donc), va-t-il s'exécuter rapidement ? Va-t-il consommer beaucoup de mémoire pendant son exécution ? On peut quantifier ces problèmes grâce à la complexité temporelle qui nous donnera un ordre de grandeur du temps d'exécution de l'algorithme et à la complexité spatiale qui nous donnera un ordre de grandeur de la quantité de mémoire requise à l'exécution de l'algorithme. L'objet de cette partie est la complexité temporelle.

L'intérêt de la complexité temporelle est de pouvoir évaluer la rapidité d'un algorithme indépendamment de la machine sur laquelle il est exécuté. En effet, la première idée qui pourrait venir pour savoir si un algorithme est efficace, serait de mesurer le temps d'exécution de celui-ci : le problème est que ce temps dépend non seulement de la machine mais aussi des processus déjà en cours sur la machine : cette mesure de temps n'est donc pas fiable pour comparer universellement deux algorithmes.

Il nous vient alors l'idée de calculer le nombre de d'opérations élémentaires effectuées dans l'algorithme pendant son exécution. Ainsi, si on part du principe que chaque opération élémentaire (comme on va la définir) prend un temps identique pour s'effectuer, en évaluant le nombre de ces opérations, on obtiendra le temps d'exécution de l'algorithme. Voyons comment calculer le nombre d'opérations élémentaires d'un algorithme et surtout, définissons ce qu'est une opération élémentaire !

2. Le coût d'un algorithme

Définition 3.

On appelle **coût** d'un algorithme le nombre d'*opérations élémentaires* effectuées par l'algorithme.

Mais comment définit-on une opération élémentaire ?

Définition 4.

On appelle **opération élémentaire** une des actions suivantes (la liste n'est pas exhaustive) :
- Faire une opération mathématique : addition, soustraction, multiplication, division, multiplication matricielle, puissance... ;
- Lire le contenu d'une variable ;
- Affecter une valeur à une variable ;
- Effectuer un test : `==`, `>=`, `!=`,... ;
- Faire une opération booléenne : `and`, `not`,...

Voici un exemple :

```
1 def sommeCarre(n):
2     """ Somme des n premiers carres non nuls """
3     S=0
4     while n>0:
5         S=S+n**2
6         n=n-1
7     return S
```

Question. Apparté

Au fait, quelle est la formule pour la somme des carrés de n premiers entiers plus grand que 1 ?

Question 1.

Quel est le coût de cette algorithme ?

Si on compte toutes les opérations élémentaires citées plus haut, on obtient :
- $n+n+n$ opérations mathématiques ;
- $n+n+n+n+1$ lectures ;
- $1+n+n$ affectations ;
- n comparaisons

Donc le coût total est de $10n+2$ opérations élémentaires.

On remarque alors deux choses qui semblent peu cohérentes dans ces considérations :

- Parmi ces opérations, certaines prennent sûrement plus de temps que d'autres !
- Pour une même opération, le temps d'exécution est proportionnel à la taille des données !

Comment avoir un calcul de coût pertinent dans ces conditions ? Et bien on va faire les hypothèses suivantes afin d'une part, de rendre le calcul de coût plus facile, et d'autre part, de le rendre représentatif de l'efficacité de l'algorithme :

Hypothèse de calcul du coût

1. On fera toujours l'approximation que toutes les opérations élémentaires prennent le même temps ;
2. Dans le calcul du coût, on ne comptera que les opérations élémentaires les plus significatives de l'algorithme.

La deuxième hypothèse est peut sembler subjective : et bien elle l'est ! Mais dans la plupart des cas, il y aura bien une opération plus important que les autres, et qui est la plus pertinente à sélectionner en fonction du contexte de l'algorithme. Par exemple, pour l'algorithme précédent, ce sont les opérations mathématiques (on calcule une somme de carrés) qui constitue les opérations importantes pour cet algorithme. et ainsi, notre calcul de coût devient $3n$ pour ces opérations.

On voit bien dans notre exemple que le coût d'un algorithme dépend fortement de l'argument n. Ainsi, le coût dépend de la taille des données que traite l'algorithme ; mais qu'entend-on exactement par "taille des données" ?

Définition 5.

La taille des données d'un algorithme est un entier n (ou même plusieurs) qui mesure la données à traiter. Comme un algorithme est ici écrit comme une fonction, les données à traiter sont les paramètres d'appels de cette fonction.

Les cas les plus fréquents sont :

- donnée = entier $n \Rightarrow$ taille $= n$
- donnée = liste $\Rightarrow$ taille = longueur de la liste
- donnée = plusieurs entiers $\Rightarrow$ taille = le plus grand de ces entiers (par exemple).

Exemple 4. *Calcul de coût*

Considérons l'algorithme suivant qui trie dans l'ordre croissant une liste de nombres donnée en argument :

```python
def tri(L):
    n = len(L)
    for i in range(n - 1):
        for j in range(i, n):
            if L[i] > L[j]:
                x = L[i]
                L[i] = L[j]
                L[j] = x
    return L
```

On remarque que comme on parcourt les indices de la liste `L`, le coût va être directement dépendant de la taille n de la liste. Il est donc judicieux de considérer n comme la *taille de la donnée*.

Ici, les opérations qui semblent prédominantes sont les tests et les affections des variables. Calculons le nombre de ces opérations dans l'algorithme :

— Les tests : il y en a 1 pour chaque itération de la deuxième boucle `for` imbriquée dans la première donc il y en a :

$$\sum_{i=0}^{n-2}\sum_{j=i}^{n-1} 1 = \sum_{i=0}^{n-2} n-i = \frac{(n-2)(n+3)}{2}$$

— Les affectations : il y en a 1 au départ et en fonction de la véracité de chaque test il y en a entre 0 et

$$\sum_{i=0}^{n-2}\sum_{j=i}^{n-1} 3 = \frac{3(n-2)(n+3)}{2}$$

Ainsi, dans le cas des affectations, on se rend compte que le coût dépend non seulement de la taille n de la donnée, mais également des caractéristiques de la donnée : en effet, on peut voir que si on applique notre fonction à :

— une liste déjà triée dans l'ordre croissant, la comparaison `L[i] > L[j]` sera toujours fausse, et il n'y aura pas d'affectation au cours de l'algorithme ; il s'agit du meilleur des cas ;

— une liste triée dans l'ordre décroissant, la comparaison `L[i] > L[j]` sera toujours vraie, et il aura le nombre maximal d'affectation décrit plus haut ; il s'agit du pire des cas ;

— une liste quelconque, alors on n'est pas capable de donner précisément le nombre d'affectations autrement qu'en l'encadrant par les valeurs du meilleur et du pire des cas !

On voit dans l'exemple précédent que dans le cas général, il est peut-être difficile d'obtenir le coût exact pour des données quelconques.
Comme le but de ces calculs est d'obtenir la rapidité de l'algorithme, on va en fait chercher à savoir dans quel est sa rapidité lorsqu'on le "pousse dans ses retranchements" - quand, pour une taille de donnée fixée, on lui donne la pire donnée possible :

Définition 6. *Coût dans le pire des cas*

On appelle **coût dans le pire des cas** d'un algorithme, son coût maximal à taille de donnée fixée i.e. son coût dans le pire des cas d'exécution possibles.

Remarque 3.

On peut bien-sûr définir le coût dans le meilleur des cas, mais aussi le coût en moyenne d'un algorithme en utilisant des probabilités/dénombrements afin de mieux estimer la rapidité globale de cet algorithme ; en effet, si le pire des cas est très "improbable", il serait dommage de se baser sur le coût dans ce cas.

Exercice 3.

On considère l'algorithme suivant :

```
1 def recherche(x,L):
2     n=len(L)
3     for k in range(n):
4         if L[i] == x:
5             return i
6     return None
```

1. Quel est le rôle de cet algorithme ?
2. Décrire le meilleur et le pire des cas d'exécution pour des listes de même taille `n`.
3. Calculer le coût dans le pire des cas.

Correction.

1. Comme son nom l'indique, et nous l'avons déjà programmé, cet algorithme détermine l'indice de la valeur `x` dans le tableau `L` et renvoie `None` s'il n'y est pas .
2. L'opération élémentaire important ici est la comparaison `==` car c'est un algorithme de "recherche".
   - On remarque que si l'élément recherché `x` se trouve en première position dans le tableau `L` (à l'indice `0`), alors l'algorithme s'arrête à la première itération (`i`$= 0$) de la boucle `for` car dans ce cas, `L[0]==x` vaut `True` et donc l'indice `0` est retourné par la fonction.
   - On remarque que si l'élément recherché `x` ne se trouve pas dans le tableau `L` alors l'algorithme ne s'arrête qu'une fois la boucle `for` terminée en retournant `None`
3. Dans le pire des cas, on a 1 comparaison à chaque itération pour `i` allant de 0 à $n-1$ où $n =$`len(L)` (taille de la donnée ici), donc le coût de cet alogrithme dans le pire des cas est $c(n) = n$.

3. Calcul pratique du coût d'un algorithme

a. Coût d'un algorithme itératif - boucle `for`

Grâce à l'exemple précédent, on observe que le coût d'un bloc d'une boucle `for` revient s'obtient de la manière suivante :

Calcul du coût d'une boucle `for`

```
for i in range(N):
    instructions(i) #c_f(i) opérations
```

- Pour chaque `i` de la boucle **`for`**, on détermine le nombre $c_f(i)$ d'opérations élémentaires effectuées par `instructions(i)` dans le bloc de la boucle **`for`** ;
- Puis on somme ces nombres $c_f(i)$ d'opérations pour i allant de l'indice de début de la boucle **`for`** (ici 0) à l'indice de fin de la boucle (ici $N-1$).
  Dans le cas précédent, cela donne :

$$\sum_{i=0}^{N-1} c_f(i).$$

Exercice 4.

Calculer le coût des algorithmes suivants en termes d'opérations arithmétiques :

1.
```
def f1(n):
    a=1
    for i in range(n):
        a=a+i*3
    return a
```

2.
```
def f2(n):
    a=1
    for i in range(n):
        a=1+2*a
    for j in range(n):
        a=a//2
    return a
```

3.
```
def f3(n):
    x=1
    for i in range(n):
        for j in range(n):
            x=x+5
    return x
```

Correction.

1. Le coût $c(n)$, pour la fonction `f1(n)`, vérifie :

$$c(n) = \sum_{i=0}^{n-1} 2 = 2n.$$

2. Le coût $c(n)$, pour la fonction `f2(n)`, vérifie :

$$c(n) = \sum_{i=0}^{n-1} 2 + \sum_{j=0}^{n-1} 1 = 2n + n = 3n.$$

3. Le coût $c(n)$, pour la fonction `f3(n)`, vérifie :

$$c(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 = \sum_{i=0}^{n-1} n = n^2.$$

Exercice 5.

Calculer le coût, en terme d'opérations mathématiques i.e. additions, produits et racines de l'algorithme suivant en fonction de la donnée n :

```
1 def mystere(a,b,n):
2     for i in range(n):
3         a,b=0.5*(a+b),sqrt(a*b)
4     return b
```

Bonus : que renvoie cet algorithme pour deux nombres a, b positifs ?

Correction.

Cette algorithme renvoie une approximation de la moyenne arithmético-géométrique des nombres a et b. Sont coût $c(n)$ vérifie (en supposant que la fonction `sqrt` correspond à 1 opération arithmétique) :

$$c(n) = \sum_{i=0}^{n-1} 4 = 4n$$

b. Coût d'un algorithme itératif - boucle `while`

Dans le cas d'une boucle **`while`**, le principe est le même qu'avec une boucle **`for`** mais une difficulté apparaît : alors qu'on connaît le nombre d'itérations de boucle d'un **`for`**, le nombre d'itérations d'un **`while`** n'est pas connu a priori !
Il faut donc le déterminer afin de pouvoir effectuer le même calcul que pour une boucle **`for`**.

Calcul du coût d'une boucle `while`

```
u=valeur_initiale
while condition(u):
    instructions(u) #u n'est pas modifié ici
    u=modif(u)
```

- Pour chaque `i` de la boucle **`while`**, on détermine le nombre $c_w(u)$ d'opérations élémentaires effectuées par `instructions(u)` et `u=modif(u)` dans le bloc de la boucle **`while`** ;
- On modélise les valeurs de u **à la fin** de chaque itération k de la boucle **`while`** grâce à une suite récurrente $(u_k)_{k\in\mathbb{N}}$:

$$\begin{cases} u_0 &= \texttt{valeur\_initiale} \\ u_k &= \texttt{modif}(u_{k-1}) \text{ pour } k \in \mathbb{N}^*. \end{cases}$$

On utilise alors nos talents mathématiques pour déterminer une expression explicite de cette suite pour $k \in \mathbb{N}$:

$$u_k = f(k)$$

Grâce à cela, on peut déterminer le nombre d'itérations N de la boucle **`while`**. En effet, ce nombre N est déterminé par :

$$\texttt{condition}(u_{N-1}) \text{ est } \textbf{VRAI !} \text{ et } \texttt{condition}(u_N) \text{ est } \textbf{FAUX !}$$

Dans la pratique, on sera souvent amené à utiliser la fonction réciproque de la fonction f telle que $u_k = f(k)$ pour déterminer N (si tant est quelle soit bijective !).
De plus, il arrivera très souvent que l'on ne puisse pas déterminer exactement le nombre N ! On essaiera alors d'obtenir un encadrement de N en fonction de la taille de la donnée n.
On obtient ainsi toutes les valeurs de u lors de l'exécution, il s'agit de :

$$u_0, u_1, ..., u_N.$$

- Finalement, on somme les nombres $c_w(u)$ d'opérations pour chaque valeur de u.
Dans le cas précédent, cela donne :

$$\sum_{k=0}^{N-1} c_w(u_k).$$

Exemple 5.

Voici un exemple concret :

```python
def algo(n):
    a=1
    S=0
    while a<=n:
        S=S+a
        a=a*2
    return S
```

On calcule le coût de `algo(n)` en termes d'opérations arithmétiques :

- Il y a 2 opérations arithmétiques dans le bloc de la boucle `while`.
- Soit (a_k) la suite des valeurs de la variables a. Alors, on a :

$$\begin{cases} a_0 &= 1 \\ a_k &= 2a_{k-1} \text{ pour } k \in \mathbb{N}^*. \end{cases}$$

Ainsi, en calculant de deux manières le produit $\prod_{i=1}^{k} \frac{a_i}{a_{i-1}}$, on obtient :

$$a_k = \frac{a_k}{a_0} = \prod_{i=1}^{k} \frac{a_i}{a_{i-1}} = 2^k.$$

Le nombre de d'itération N de la boucle `while` vérifie alors

$$a_{N-1} \leq n \text{ i.e. } 2^{N-1} \leq n \text{ et } a_N > n \text{ i.e. } 2^N > n;$$

d'où l'encadrement :

$$\log_2(n) < N \leq \log_2(n) + 1.$$

bu On peut alors calculer le coût $c(n)$ de la boucle `while` et a fortiori de `algo(n)` puisqu'il n'y a pas opérations en dehors de cette boucle :

$$c(n) = \sum_{k=1}^{N-1} 2 = 2(N-1)$$

Or, d'après l'encadrement précédent, on a :

$$2\log_2(n) + 2 < c(n) \leq 2\log_2(n) + 4.$$

Remarque : Nous verrons dans la suite qu'un tel encadrement nous convient très bien !

Exercice 6.

Déterminer le coût des algorithmes suivant en termes d'opérations arithmétiques :

1. Algorithme 1

```
def f1(n):
    u=1
    while u<=2*n:
        u+=1
    return u
```

2. Algorithme 2

```
def f2(n):
    u=2**n
    P=1
    while u>1:
        P=P*u
        u=u//2
    return P
```

3. Algorithme 3

```
def f3(n):
    a=1
    S=0
    while a<=n:
        for i in range(a):
            S+=i
        a=a*2
    return S
```

Correction.

1. Soit (u_k) la suite des valeurs de la variables u. Alors, on a :

$$\begin{cases} u_0 &= 1 \\ u_k &= u_{k-1} + 1 \text{ pour } k \in \mathbb{N}^*. \end{cases}$$

Il s'agit d'une suite arithmétique de raison 1 donc, pour $k \in \mathbb{N}$

$$u_k = 1 + k \times 1 = 1 + k.$$

Le nombre de tours N de la boucle `while` vérifie alors

$$u_{N-1} \leq 2n \text{ i.e. } 1 + N - 1 \leq 2n \text{ et } u_N > 2n \text{ i.e. } 1 + N > 2n;$$

d'où l'encadrement :

$$2n - 1 < N \leq 2n.$$

On a alors :

$$c(n) = \sum_{k=1}^{N-1} 2 = 2(N-1)$$

Or, d'après l'encadrement précédent, on a :

$$4n - 4 < c(n) \leq 4n - 2.$$

2. Soit (u_k) la suite des valeurs de la variables u. Alors, on a :

$$\begin{cases} u_0 & = 2 * *n \\ u_k & = \frac{u_{k-1}}{2} \text{ pour } k \in \mathbb{N}^*. \end{cases}$$

Il s'agit d'une suite géométrique de raison $\frac{1}{2}$ donc, pour $k \in \mathbb{N}$

$$u_k = 2^n \times \frac{1}{2^k} = 2^{n-k}.$$

Le nombre de tours N de la boucle **while** vérifie alors

$$u_{N-1} > 1 \text{ i.e. } 2^{n-N+1} > 1 \text{ et } u_N \leq 1 \text{ i.e. } 2^{n-N} \leq 1;$$

d'où :

$$n - N + 1 > \log_2(1) = 0 \text{ et } n - N \leq \log_2(1) = 0.$$

et donc :

$$n \leq N < n + 1.$$

On a alors :

$$c(n) = \sum_{k=1}^{N-1} 2 = 2(N-1)$$

Or, d'après l'encadrement précédent, on a :

$$2(n-1) \leq c(n) \leq 2n.$$

3. Soit (a_k) la suite des valeurs de la variables a. Alors, on a :

$$\begin{cases} a_0 & = 1 \\ a_{k+1} & = 2a_k \text{ pour } k \in \mathbb{N}. \end{cases}$$

Il s'agit d'une suite géométrique de raison 2 donc, pour $k \in \mathbb{N}$

$$a_k = 2\kappa.$$

Le nombre de tours N de la boucle **while** vérifie alors

$$a_{N-1} \leq n \text{ i.e. } 2^{N-1} \leq n \text{ et } a_N > n \text{ i.e. } 2^N > n;$$

$$\log_2(n) < N \leq \log_2(n) + 1.$$

On a alors :

$$c(n) = \sum_{k=1}^{N-1} c_k$$

où c_k est le nombre d'opérations dans la boucle `while` au tour k i.e.

$$c_k = 1 + \sum_{i=0}^{a_k - 1} 1 = 1 + a_k = 1 = 2^k$$

Donc :

$$c(n) = \sum_{k=1}^{N-1} (1 + 2^k) = N - 1 + \frac{1 - 2^N}{1 - 2} = N - 1 + 2^N - 1 = 2^N + N - 2.$$

Or, d'après l'encadrement précédent, on a :

$$n + \log_2(n) - 2 < c(n) \leq 2n + \log_2(n) - 1.$$

c. Cas récursif

Dans le cas d'un algorithme récursif, on calcule le coût par récurrence sur la taille des données n.

Calcul du coût d'un algorithme récursif

```
def recursif(n):
    if cas_initiaux(n):
        instruction(n)
    else:
         recursif(f1(n))
         recursif(f2(n))
         ...
         recursif(fm(n))
```

On note $c(n)$ le coût de l'algorithme pour une taille de donnée n.

- Si n est dans les cas initiaux (i.e. si `cas_initiaux(n)` est vrai), alors $c(n)$ vaut le coût de `instruction(n)`
- Sinon, pour chaque appel `recursif(fk(n))`, le coût associée vaut $c(fk(n))$ (plus le coût de `fk(n)`) donc, dans ce cas :

$$c(n) = c(f1(n)) + ... + c(fm(n)) \text{ (+ } \textit{somme des coûts des } \texttt{fk(n)}\text{)}$$

Ainsi, la suite $c(n)$ est une suite récurrente, et encore une fois, nos talents de mathématiciens nous permettront de trouver la valeur explicite de $c(n)$ en fonction de n.

Comprenons ceci directement sur un exemple : considérons la suite récurrente

$$u_0 = 2 \text{ et } u_{n+1} = \tfrac{1}{2}(u_n + \tfrac{1}{u_n}).$$

Calculons le coût (en terme d'opérations arithmétiques) de deux implémentations récursives différentes de cette suite :

Programme récursif 1 (naïf)

```
def suite(n):
    if n == 0:
        return 2
    return 0.5 * (suite(n - 1) + 1 / suite(n - 1))
```

Soit $c(n)$ le coût de `suite(n)`. Alors on a :

- $c(0) = 0$
- pour $n \geq 1$, $c(n) = 2c(n-1) + 3$.

Donc, par récurrence, on obtient $c(n) = 3(2^n - 1)$. Ce qui semble plutôt croître très vite (expoenentiellement vite en fait) en fonction de n !

Essayons de faire mieux !

Programme récursif 2 (amélioré !)

```
def meme_suite(n):
    if n == 0:
        return 2
    x=meme_suite(n-1)
    return 0.5*(x+1/x)
```

Soit $c'(n)$ le coût de `meme_suite(n)`. Alors on a :

- $c'(0) = 0$
- pour $n \geq 1$, $c'(n) = c'(n-1) + 3$.

Donc, par récurrence, $c'(n) = 3n$. Le coût croit linéairement en fonction de n ; c'est nettement mieux !

Avant de s'exercer sur le calcul de complexité d'algorithmes récursifs, comparons les résultats précédents avec le cas itératif :

Exercice 7.

Donner une version itérative du programme de calcul de la suite précédente et calculer son coût.

```
def uI(n):
    u = 2
    for i in range(n):
        u = 0.5 * (u + 1 / u)
    return u
```

Exercice 8.

On donne la fonction suivante :

```
def s(m,n):
    """ n entier, n entier positif """
    if n==0:
        return m
    return s(m+1,n-1)
```

Quel résultat renvoie cette fonction ? Quelle est son coût en fonction de la donnée n

4. La complexité

Malgré la simplicité de nos exemples, on peut voir que le calcul du coût peut très vite devenir impossible pour des algorithmes très grands. Et on remarque que si n est très grand, le fait qu'il y ait n^2 opérations où n^2+5n+6 opérations revient quasiment au même puisque les autres termes sont négligeables devant n^2.

Ainsi on a l'idée de définir la complexité d'un algorithme de la façon suivante :

Définition 7. *Complexité*

La **complexité temporelle** d'un algorithme est une fonction simple telle que le coût de l'algorithme (pour des opérations élémentaires données) est dominée par quand la taille des données tend vers l'infini.

Ainsi, si on note n la taille des données, et $c(n)$ le coût ; la complexité $C(n)$ de l'algorithme vérifie :

$$c(n) = O(C(n)).$$

Remarque 4.

Pour rappeler que la complexité est une approximation asymptotique, on notera souvent $C(n) = O(n^2)$ par exemple au lieu de $C(n) = n^2$.

Quelques fonctions de complexité

$C(n) = O(1)$	complexité constante	extraordinaire !
$C(n) = O(\ln(n))$	complexité logarithmique	excellent
$C(n) = O(n)$	complexité linéaire	super bien
$C(n) = O(n\ln(n))$	complexité quasi-linéaire	très bien
$C(n) = O(n^2)$	complexité quadratique	moyen
$C(n) = O(n^3)$	complexité cubique	mauvais
$C(n) = O(n^p)$	complexité polynomiale ($p > 3$)	très mauvais
$C(n) = O(a^n)$	complexité exponentielle ($a > 1$)	horrible
$C(n) = O(n!)$	complexité factorielle	à proscrire

	log(n)	n	n log(n)	n^2	n^3	2^n	n !
10^2	7 ns	100 ns	0,7 µs	10 µs	1 ms	4.10^{13} ans	3.10^{141} ans
10^3	10 ns	1 µs	10 µs	1 ms	1 s	10^{292} ans	10^{2560} ans
10^4	13 ns	10 µs	133 µs	100 ms	17 s	6.10^{3002} ans	
10^5	17 ns	100 µs	2 ms	10 s	11,6 jours		
10^6	20 ns	1 ms	20 ms	17 min	32 ans		

5. Calcul de Complexité

a. Exemple :

Pour un algorithme itératifs ou récursifs, il s'agit donc simplement de calculer des sommes pour obtenir le coût dans le pire des cas, et de chercher une fonction simple qui domine ce coût pour obtenir la complexité :

Revenons à notre tri de l'exemple précédent :

```
def tri(L):
    n = len(L)
    for i in range(n - 1):
        for j in range(i, n):
            if L[i] > L[j]:
                x = L[i]
                L[i] = L[j]
                L[j] = x
    return L
```

On a vu que dans le pire des cas, $c(n) = 2 + 4\frac{n(n-1)}{2}$. Donc $C(n) = O(n^2)$! La complexité temporelle dans le pire des cas est quadratique pour cet algorithme.

b. Exercices basiques

Exercice 9. *Factorielle et binôme de Newton*

1. Écrire une fonction `facto(n)` qui renvoie la factorielle du nombre `n`
2. Calculer sa complexité.
3. Écrire une fonction `binom(n,p)` qui renvoie la valeur du binôme de Newton $\binom{n}{p}$ en vous servant de la fonction `facto` précédemment écrite.
4. Calculer sa complexité en fonction de n (et du pire des cas pour p).

Exercice 10.

Calculer la complexité des 6 algorithmes suivants (en terme d'opérations arithmétiques) :

```python
#Algo f1
def f1(n):
    x = 0
    for i in range(n):
        for j in range(n):
            x = x+1
    return x
```

```python
#Algo f2
def f2(n):
    x = 0
    for i in range(n):
        for j in range(i):
            x = x+1
    return x
```

```python
#Algo f3
def f3(n):
    x = 0
    for i in range(n):
        j = 0
        while j * j < i:
            x += 1
            j += 1
    return x
```

```python
#Algo f4
def f4(n):
    x = 0
    i = 1
    while i < n:
        x += 1
        i *= 2
    return x
```

```python
#Algo f5
def f5(n):
    x,i = 0,n
    while i > 1:
        x+=1
        i//=2
    return x
```

```python
#Algo f6
def f6(n):
    x,i = 0,n
    while i > 1:
        for j in range(i):
            x+=1
        i //=2
    return x
```

Exercice 11.

Considérons l'algorithme suivant : (remarque, si ce n'est pas déjà fait, n'oubliez pas le `from math import *` pour avoir accès au fonctions mathématiques de Python)

```python
def mystere(n):
    for d in range(2, floor(sqrt(n)) + 1):
        if n % d == 0:
            return False
    return True
```

1. Que détermine l'algorithme suivant ?
2. Calculer sa complexité en fonction de n en prenant en compte les congruences (les `%`) comme opérations élémentaires.

6. Mise en pratique : Exercices sur la complexité temporelle

Exercice 12.

Déterminer la complexité des algorithmes suivants en terme d'opérations arithmétiques :

```python
#Algo1
def f1(n):
    p = 0
    for i in range(n):
        for j in range(n):
            for k in range(n):
                s = 2 * s
```

```python
#Algo2
def f2(n):
    p = 0
    for i in range(n):
        s = s + 3
    for j in range(n):
        s = s + 3
```

```python
#Algo3
def f3(n):
    s = 0
    for i in range(n):
        for j in range(i,n):
            s = s + j**2
```

```python
#Algo4
def f4(n):
    p = 1
    for i in range(n):
        for j in range(i-5,i+5):
            p = 2*p
```

```python
#Algo5
def f5(n):
    s = 0
    for i in range(n):
        a = n
        while a > 1:
            a = a/2
        s += 1
```

```python
#Algo6
def f6(n):
    i = n
    s = 0
    while i > 1:
        for j in range(i):
            s = s + 1
        i = i/2
```

Exercice 13.

Soit $x \in \mathbb{R}$. On considère la suite récurrente $(u_n)_{n\in\mathbb{N}}$ telle que :

$$\begin{cases} u_0 = 1 \\ u_{n+1} = \frac{xu_n}{n+1} \quad \forall\, n \in \mathbb{N}^*. \end{cases}$$

1. Écrire une fonction `u(x,n)` qui renvoie le terme u_n d'indice `n` de la suite $(u_n)_{n\in\mathbb{N}}$.
2. Que fait la fonction `mystere` suivante ?

```python
def mystere(x,n):
    S=0
    for i in range(n+1):
        S=S+u(x,i)
    return S
```

3. Montrer que pour tout $n \in \mathbb{N}$, $u_n = \frac{x^n}{n!}$. En déduire une formule mathématique pour `mystere(x,n)`.
4. Importer la fonction ln du module `math` grâce à l'instruction `from math import log` (Attention `log` désigne ici le logarithme népérien et pas le logarithme en base 10 !).
   - Calculer `mystere(log(10),n)` pour $n = 1, 5, 10, 1000$. Vers quelle valeur `mystere(log(10),n)` semble-t-elle converger ?
   - Calculer `mystere(2*log(10),n)` pour $n = 1, 5, 10, 1000$. Vers quelle valeur `mystere(2*log(10),n)` semble-t-elle converger ?
   - Calculer `mystere(5*log(10),n)` pour $n = 1, 5, 10, 1000$. Vers quelle valeur `mystere(5*log(10),n)` semble-t-elle converger ?
   - Calculer `mystere(10*log(10),n)` pour $n = 1, 5, 10, 1000$. Vers quelle valeur `mystere(10*log(10),n)` semble-t-elle converger ?

   Conjecturer une formule pour la limite de `mystere(x*log(10),n)` quand n tend vers l'infini, puis pour `mystere(x,n)`. En "déduire" une formule mathématique reliant la fonction exponentielle et une somme infinie.

Exercice 14.

On considère les algorithmes suivants :

— Se terminent-t-ils ?

— Si oui, déterminer leurs coûts puis leurs complexités.

```python
#Algo1
def f1(n):
    a = 1
    while a < n:
        a = a + 5
```

```python
#Algo2
def f2(n):
    a = 1
    while a < n:
        a = 2 * a
```

```python
#Algo3
def f3(n):
    while n >= 0:
        n = n//2
```

```python
#Algo4
def f4(n):
    a = 1
    while a < n**2:
        a = 3*a+1
```

```python
#Algo5
def f5(n):
    if n==0:
        return 1
    else:
        return f5(n-1)+12
```

```python
#Algo6
def f6(n):
    if n==0:
        return 1
    else:
        return f5(n+1)+12
```

```python
#Algo7
def f7(n):
    if n==0:
        return 1
    else:
        a=0
        while a<n**2:
            a+=1
        return f7(n-1)+a/2
```

```python
#Algo8
def f8(n):
    if n<=1:
        return 1
    else:
        return f8(n//2)+1
```

Correction.

1. On remarque que $v = n - a$ est un variant de boucle donc l'algorithme se termine.
   On note $c(n)$ le coût de `f1(n)`. Alors, si N est le nombre d'itérations de la boucles `while`, on a :
   $$c(n) = \sum_{k=1}^{N} 1 = N$$
   Si on note $(a_k)_{k\in\mathbb{N}}$ la suite des valeurs de la variable `a`, on a :
   $$\begin{cases} a_0 = 1 \\ a_{k+1} = a_k + 5 \quad \text{si } k \geq 1 \end{cases} \text{ d'où, pour tout } k \in \mathbb{N}, \quad a_k = 5k + 1$$
   De plus, N le numéro de la dernière itération, on a :
   $$\begin{cases} a_N < n \\ a_{N+1} \geq n \end{cases} \Leftrightarrow \begin{cases} 5N + 1 < n \\ 5N + 6 \geq n \end{cases} \quad \text{d'où } \frac{n-6}{5} \leq N < \frac{n-1}{5}$$
   Ainsi, $\frac{n-6}{5} \leq c(n) < \frac{n-1}{5}$ et donc $c(n) \underset{n\to+\infty}{\sim} \frac{n}{5}$. Il en résulte que la complexité de `f1` est $C(n) = O(n)$ (linéaire).

5. On remarque que $a = n$ est un variant récursif donc l'algorithme se termine.
On note $c(n)$ le coût de `f5(n)`. Alors on a :

$$\begin{cases} c(0) = 0 \\ c(n) = c(n-1) + 2 \quad \text{si } n \geq 1 \end{cases}$$

Ainsi, $c(n) = 2n$. Il en résulte que la complexité de `f5` est $C(n) = O(n)$ (linéaire).

Exercice 15.

Ecrire des fonctions `compte_cara_it` et `compte_cara_rec` prenant chacun en arguments une variable `chaine` contenant une chaîne de caractères (non vide) et une variable `cara` qui contient une chaîne de caractère de longueur 1 (un caractère) et qui renvoie le nombre d'occurences de `cara` dans `chaine` où `compte_cara_it` sera programmé de manière itérative et `compte_cara_rec` de manière récursive.
Déterminer la complexité de vos algorithmes.

Correction.

```
#programme itératif
def compte_cara_it(chaine,cara):
    compteur = 0
    for i in range(len(chaine)):
        if chaine[i]==cara:
            compteur+=1
    return compteur

#programme récursif
def compte_cara_rec(chaine,cara):
    if len(chaine)==1:
        if chaine[0]==cara:
            return 1
    if chaine[0]==cara:
        return 1+compte_cara_rec(chaine[1:],cara)
    return compte_cara_rec(chaine[1:],cara)
```

Exercice 16.

Déterminer les complexités temporelles dans le pire des cas des algorithmes récursifs de recherche par dichotomie et d'exponentiation rapide.

Exercice 17.

Considérons l'algorithme suivant :

```
def myst(n):
    if n < 2:
        return 1
    return myst(n-1) + myst(n-2)
```

1. Que calcule l'algorithme suivant ?
2. Calculer sa complexité en fonction de n en prenant en compte les additions. Que dire de cette algorithme ?

Exercice 18. *Nombres de Bell*

Pour $n \in \mathbb{N}$, on définit le nombre de Bell B_n par récurrence :

$$\begin{cases} B_0 = 1 \\ B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k \end{cases}$$

1. Écrire une fonction `Bell(n)` qui prend pour argument un entier naturel `n` et qui renvoie la valeur de B_n. On pourra coder une fonction `binom(p,q)` qui renvoie la valeur de $\binom{p}{q}$.
2. Quelle est la complexité temporelle de votre algorithme en terme d'opérations arithmétiques
3. Soit E un ensemble. Une famille $(A_i)_{i\in I}$ de parties de E est appelée **partition** de E si, pour tout $i, j \in I$ avec $i \neq j$, $A_i \cap A_j = \emptyset$ et $\bigcup_{i\in I} A_i = E$. Par exemple,

   $$(\{1\}, \{2,5\}, \{3,4,6\}) \text{ est une partition de } \{1,2,3,4,5,6\}.$$

   Soit $n \in \mathbb{N}$. On considère l'ensemble :

   $$[\![1, n]\!] = \{1, \dots, n\}.$$

   On convient que pour $n = 0$, $[\![1, 0]\!] = \emptyset$.

   Montrer que le nombre de partitions de $[\![1, n]\!]$ est égal à B_n.
4. *Question Bonus très dure !* Écrire un algorithme `partition(n)` qui renvoie la liste des partitions de $[\![1, n]\!]$. Une partition sera représentée par une liste de listes : la partition $(\{1\}, \{2,5\}, \{3,4,6\})$ de l'exemple plus haut sera donc codée `[[1],[2,5],[3,4,6]]`

Partie C

Correction

1. Introduction

Dans cette partie, notre but est de trouver des outils qui permettent de montrer qu'un algorithme fait bien ce pour quoi il a été conçu. Prenons un exemple simple : on cherche à écrire un algorithme qui calcule la somme $1+2+\ldots+n$ des n premiers entiers strictement positifs.
Soit $n \in \mathbb{N}$. On remarque tout d'abord que, si on note, pour i entre 0 et n, $S_i = 1+\ldots+i$ (S_0 est une somme vide donc vaut 0), alors S_n vaut la quantité recherchée et la suite finie $(S_i)_{i\in[\![0,n]\!]}$ est uniquement définie par la relation :

$$\begin{cases} S'_0 = 0 \\ S'_i = S'_{i-1} + i \quad \text{si } i = 1,\ldots,n \end{cases}.$$

En effet, par récurrence finie, l'initialisation est vraie $S'_0 = 0 = S_0$ et pour $i \in [\![0, n-1]\!]$, si $S'_i = S_i = 1+\ldots+i$, on a $S'_{i+1} = S'_i + (i+1) = 1+\ldots+i+(i+1) = S_{i+1}$.
Cela nous donne l'idée de définir une variable `S` qui prendra les valeurs successives de la suite $(S_i)_{i\in[\![0,n]\!]}$ en nous appuyant sur la relation de récurrence précédente :

```
S=0
Pour i allant de 1 à n
    S=S+i
Fin Pour
Renvoyer S
```

La variable `S` vérifie bien la relation de récurrence de la suite (S_i) à savoir :

- Avant la première itération, la variable `S` vaut $0 = S_0$
- Si au début d'un itération $i \geq 1$, `S` vaut $S_{i-1} = 1+\ldots+(i-1)$, alors au début de la suivante, `S` vaut $S_{i-1} + i = 1+\ldots+(i-1)+i = S_i$.

Ceci montre (par récurrence, comme précédemment) que pour tout $i \geq 1$ à la fin de l'itération i, la variable `S` vaut $S_i = 1+\ldots+i$ et donc sa dernière valeur est celle pour $i = n$, c'est-à-dire $S_n = 1+\ldots+n$! Donc l'algorithme renvoie bien ce qu'on attendait de lui, à savoir $1+\ldots+n$!

Nous venons de mettre en évidence, pour concevoir notre algorithme et prouver qu'il donne le bon résultat, un **invariant de boucle**, comme nous le définirons dans la paragraphe suivant.

Définition 8.

On dit que la **correction** d'un algorithme est :
- **partielle** si lorsqu'on suppose que l'algorithme se termine, alors celui-ci renvoie le résultat attendu.
- **totale** si l'algorithme se termine et qu'il renvoie le résultat attendu.

2. Correction d'un algorithme itératif : invariant de boucle

Dans cette partie, on considère un algorithme constitué d'une boucle (`for` ou `while`).

Définition 9. *Invariant de boucle*

Un invariant de boucle est une propriété P de l'algorithme dépendante des valeurs des variables à chaque itération de la boucle telle que :

- **Initialisation :** P est vraie avant la première itération ;
- **Transmission :** si P est vraie au début d'une itération alors elle est vraie à au début de la suivante.

Remarque 5.

Un invariant de boucle bien choisi permet de montrer la correction d'un algorithme itératif.

Exemple 6.

— Dans l'exemple introductif, on a montré que la propriété P ="au début de l'itération i, la variable `S` vaut $1 + \ldots i - 1$" est un *invariant de boucle* pour l'algorithme présenté.
Ainsi, à la fin de la dernière itération `n`, la variable `S` vaut $1 + \ldots n - 1 + n$ ce qui est bien la valeur recherchée : notre algorithme est donc correct !

— Considérons l'algorithme de la division euclidienne :

```
1 def divsision(a,b):
2     """ a,b entiers naturels et b non nul, renvoie le couple (q,r) du
          quotient et reste de la division euclidienne de a par b """
3     q = 0
4     r = a
5     while r >= b:
6         r = r-b
7         q = q+1
8     return q,r
```

On considère la propriété P ="au début d'une itération, $a = bq + r$". Alors :

- **Initialisation :** Avant la première itération, on a : $q = 0$ et $r = a$ donc :

$$bq + r = b \times 0 + a = a$$

  P est donc vraie avant la première itération. ;
- **Transmission :** si P est vraie au début d'une itération i quelconque, on a $a = bq_i + r_i$ où q_i, r_i sont les valeurs de `q,r` au début de l'itération. A la fin de l'itération, on a $r = r_i - b$ et $q = q_i + 1$ donc :

$$bq + r = b(q_i + 1) + (r_i - b) = bq_i + b + r_i - b = bq_i + r_i = bq_i + r_i = a$$

  Donc P est vraie au début de l'itération suivante.

Ainsi, P est bien un invariant de boucle.

On en déduit que si la boucle se termine (et on peut le prouver ici !) alors la fin de la dernière itération et donc à la sortie de la boucle, on a $a = bq + r$.
De plus, si on note r' et r les valeurs de la variables `r` au début et à la fin respectivement de la **dernière** itération, alors $r' \geq b$ et $r = r' - b$, donc $r \geq 0$ et comme il s'agit de la dernière itération, on a $r < b$.

Par suite, on a $a = bq+r$ et $0 \leq r < b$ ce qui est bien le résultat attendu : notre algortihme de division euclidienne est donc correct !

— On aurait pu, dans l'algorithme précédent, poser $P =$" au début d'une itération, r est un entier". Cetté propriété est bien un invariant de boucle, mais il n'a aucune utilité quant à la correction de notre algorithme ! Le *bon* choix d'un invariant est donc déterminant !

Remarque 6.

Un invariant de boucle a plusieurs intérêts : comme vu dans les exemples précédents, il peut permettre, s'il est bien choisi (et c'est ça le plus dur !) de prouver la correction d'un algorithme itératif ; il peut également permettre de construire un algorithme : en effet, en exprimant un bon invariant de boucle avant la conception de l'algorithme, on peut s'appuyer sur celui-ci pour l'écrire ... *correctement* !

Exercice 19.

Écrire une n-ième fois un algorithme itératif qui permet de calculer $n!$ et prouver qu'il est correct grâce à un invariant de boucle bien choisi.

Correction.

```
1 def facto(n):
2     p=1
3     for i in range(1,n+1):
4         p=p*i
5     return p
```

On considère la propriété $P(i) =$"au début de l'itération i, $p = (i-1)!$" pour $i = 1, ..., n$. Alors :

- **Initialisation :** Au début de la première itération $i = 1$, on a $p = 1 = 0! = (1-1)! = (i-1)!$ donc $P(1)$ est vraie. ;
- **Transmission :** si $P(i)$ est vraie au début de l'itération i quelconque i.e. $p = (i-1)!$ au début de l'itération, alors, au début de la suivante, on a : $p = (i-1)! \times i = i!$, donc $P(i+1)$ est vraie.

Ainsi, P est bien un invariant de boucle.

On en conclut que pour la dernière itération $i = n$, $P(n)$ est vraie ; alors au début de l'itération n, $p = (n-1)!$. Donc à la fin de celle-ci p avut $(n-1)! \times n = n!$.
On a donc prouvé que `facto(n)` renvoie $n!$. L'algorithme est donc correct.

Exercice 20.

Prouver que l'algorithme itératif de recherche dichotomique (cf exercice 1, Algorithme 5) est correct.

3. Correction d'un algorithme récursif

Comme on l'a vu dans le cas récursif, l'utilisation d'un invariant de boucle revient à faire ce qu'on appelle en mathématiques, un raisonnement par récurrence. Dans le cas récursif, le principe est similaire : même si on ne parlera pas d'invariant de boucle, on fera un raisonnement par récurrence pour démontrer la correction. Voyons ceci directement sur un exemple :

Exemple 7.

Considérons un algorithme récursif de calcul de $n!$ et prouvons qu'il renvoie bien le bon résultat :

```
1 def facto(n):
2     if n==0:
3         return 1
4     else:
5         return facto(n-1)*n
```

Prouvons par récurrence sur $\mathbb{N}$ que, pour tout $n \in \mathbb{N}$, `facto(n)`$= n!$.

- **Initialisation :** On a `facto(0)` renvoie 1 qui est bien le résultat attendu.
- **Hérédité :** On suppose que pour $n \in \mathbb{N}^*$, `facto(n-1)`$= (n-1)!$. Alors on a :

$$\texttt{facto(n)} = \texttt{facto(n-1)} \times n = (n-1)! \times n = n!$$

Ce qui achève notre raisonnement : ainsi, pour tout $n \in \mathbb{N}$, `facto(n)`$= n!$; ce qui prouve la correction de notre algorithme.

Exercice 21.

Ecrire un algorithme de complexité linéaire de calcul de x^n pour $n \in \mathbb{N}$ et démontrer qu'il est correct.

Exercice 22.

Prouver que les algorithmes récursifs de recherche dichotomique et d'exponentiation rapide sont corrects.