Preuves d'algorithmes

C. Charignon

Il n'est en général pas raisonnable de vouloir démontrer complètement que chaque fonction que l'on écrit est correcte. Cependant, les techniques vues dans ce chapitre seront extrêmement utiles pour réussir à faire une fonction correcte, et si besoin pour débugger une fonction fausse.

Table des matières

Première partie
Cours

Lorsqu'on veut prouver qu'un algorithme fonctionne, on sépare souvent en deux étapes : d'abord vérifier que l'algorithme se termine, puis que le résultat donné est le bon.

1 Terminaison d'une boucle

1.1 Boucles inconditionnelles

A priori une boucle « pour » pourrait planter si on modifiait l'indice de la boucle. Exemple :

pour *i de 1 à 10* **:**
  $i \leftarrow i - 1$
fin

C'est pourquoi nous adoptons la règle : *Dans une boucle "pour", ne jamais faire varier l'indice de la boucle.* Si vous voulez contrôler vous-même l'indice, faites une boucle « tant que » !

Cependant, essayez donc sous Python les instructions suivantes :

```
for i in range(0,10):
    i-=1
    print(i)
```

En fait, Python calcule à l'avance (dès qu'il évalue le **range**) les valeurs que devra prendre i. Ici, le premier calcul effectué est le `range(0,10)` dont le résultat est $[\![0, 10[\![$ [1]. Et le code devient donc :

```
for i in [0,1,2,3,4,5,6,7,8,9]:
    i-=1
    print(i)
```

L'indice i avance alors dans cette liste de valeurs, indépendamment de ce qui a pu lui arriver pendant la boucle.

On pourrait aussi imaginer ceci :

```
n=3
for i in range(0,n):
    n+=1
    print(i)
```

Mais là encore, la boucle va bien terminer. En effet, une fois qu'on a évalué le `range(0,n)`, le code devient :

```
n=3
for i in [0,1,2]:
    n+=1
    print(i)
```

Ainsi, i parcourt la liste de valeurs calculées à l'avance, même si entre temps n a changé.

D'autre langages, comme Caml, interdisent purement et simplement de modifier l'indice de la boucle.

Bref, en résumé, une boucle « pour » de type `for variable in range(...)` termine toujours en Python (et en Caml) !

Bonus : Si vous voulez absolument faire planter une boucle pour, il suffit de modifier directement la liste des valeurs parcourues par l'indice :

1. Il y a des subtilité quant au type de l'objet renvoyé par `range`, dont je ne parlerai pas ici.

```python
l=[1]
for i in l:
    l.append(1)
    print(l)
```

1.2 Boucles conditionnelles

1.2.1 Le théorème de base

Le théorème principal pour montrer la terminaison d'une boucle « tant que » est celui-ci :

Théorème 1.1. *Il n'existe pas de suite $u \in \mathbb{N}^{\mathbb{N}}$ qui soit strictement décroissante.*
Plus généralement il n'existe pas d'ensemble $I \in \mathcal{P}(\mathbb{N})$ infini et de suite $u \in \mathbb{N}^I$ strictement décroissante.

Démonstration : Supposons par l'absurde l'existence d'un tel I et d'une telle suite u. L'ensemble des valeurs prise par u (c'est-à-dire $u(I)$ c'est-à-dire $\{u_n \ ; \ n \in I\}$) est un ensemble d'entier naturel, et non vide. Il admet donc un minimum, que nous notons m.

Par définition d'un minimum, m est atteint par u. Soit $n_0 \in I$ tel que $u_{n_0} = m$. Soit ensuite $n \in I$ tel que $n > n_0$ (existe car I est infini). Alors $u_n < u_{n_0}$ car u est strictement décroissante. Ceci contredit le fait que m est le minimum de u ! □

Autrement dit, pour une suite u, la conjonction des propriétés suivantes est impossible :

- u est définie sur un ensemble infini ;
- u est à valeur dans $\mathbb{N}$;
- u est strictement décroissante.

Ou encore, si u est une suite d'entier naturels, strictement décroissante, alors elle n'est pas définie sur un ensemble infini, c'est-à-dire qu'*elle s'arrête au bout d'un nombre fini de termes.*

Dès lors, une méthode pratique pour démontrer la terminaison d'une boucle « tant que » est la suivante : trouver une quantité entière positive qui diminue strictement à chaque itération de la boucle. Vu le théorème ci-dessus, une telle quantité ne peut pas continue à décroître strictement une infinité de fois, donc la boucle ne peut pas continuer à s'exécuter une infinité de fois.

Une telle quantité sera appelée un « variant de boucle ».

Théorème 1.2. *(théorème de terminaison des boucles conditionnelles)*
On suppose que lors de l'exécution d'une boucle conditionnelle, il existe une quantité :

- *entière ;*
- *positive ;*
- *et qui décroît strictement lors de chaque itération.*

Alors cette boucle termine.

1.2.2 Exemples

```python
import numpy as np

def f(t):
    deb=0
    fin=len(t)-1
    res=[]
    while deb<=fin:
        if np.random.randint(0,2)==1:
            res.append(t[deb])
            deb+=1
        else:
            res.append(t[fin])
            fin-=1
    return res
```

On prend ici comme variant de boucle la quantité `fin-deb`. En effet, cette quantité est entière (différence de deux entiers), positive (par la condition de la boucle) et strictement décroissante car elle diminue de 1 à chaque itération.

```
def fusion(t1, t2):
    """ t1 et t2 doivent être deux tableaux triés. Ceci les fusionne en un tableau trié."""

    n1, n2 = 0, 0
    res=[]

    while n1< len(t1) and n2< len(t2) :
        if t1[n1] > t2[n2]:
            res.append( t1[n1])
            n1-=1
         else:
            res.append( t2[n2])
            n2-=1
    return res
```

1. Montrons que cette fonction termine. Il suffit de considérer la quantité `len(t1)-n1 + len(t2)-n2`.
   - C'est un nombre entier.
   - À chaque itération $n1$ ou $n2$ diminue de un et les autres valeurs ne changert pas, donc `len(t1)-n1 + len(t2)-n2` diminue strictement (de 1).
   - Il est positif par la condition de la boucle.

   Par le théorème de terminaison des boucde « while », cet algorithme termine.
2. Il manque quelque chose pour que le résultat concorde avec la description. Voyez-vous quoi ? La boucle s'arrête lorsqu'on a transféré un des deux tableaux entièrement dans `res`. Mais il reste alors à transférer le reste du deuxième tableau. La manière la plus simple est de rajouter

```
res.extend(t[n1:len(t1)])
res.extend(t2[n2:len(t2)])
```

   Inutile de faire un test pour savoir lequel des deux tableaux a été entièrement transféré dans `res` : dans les deux lignes précédente une des deux n'aura aucun effet car le tableau rajouté à `res` sera vide.

1.2.3 Algorithme d'Euclide

Entrées : a,b entier, tel que $b \neq 0$
Sorties : $a \wedge b$, le pgcd de a et b
Variables locales : x,y entiers

début
  $x \leftarrow a$
  $y \leftarrow b$
  tant que $b \neq 0$ **:**
    $x \leftarrow y$
    $x \leftarrow$ reste de la division euclidienne de x par y.
  fin
  renvoyer x
fin

Algorithme 1 : algorithme d'Euclide

Ici, nous prenons comme variant de boucle la suite des valeurs successives de y.

- Ce sont des nombres entiers.
- Une fois la première itération de la boucle passée, y contient un nombre positif. En effet, le reste d'une division euclidienne est un entier positif.
- Enfin, on sait que le reste d'une division euclidienne par y est $< |y|$. À partir de la deuxième itération, $|y| = y$, et on obtient donc que la nouvelle valeur de y est $< y$.

Ainsi, la suite des valeurs prises par y vérifie les trois hypothèses : c'est une suite d'entiers positifs, strictement décroissante. Elle ne peut donc pas prendre une infinité de valeurs, et la boucle ne peut pas s'effectuer une infinité de fois.

2 Correction d'un algorithme

2.1 Introduction

Remarque : Le mot « correction » signifie ici « le fait d'être correct », et pas « l'action de corriger ».

La preuve de la correction d'un algorithme peut être compliquée, et surtout longue à rédiger. En pratique, on ne la demandera en devoir que pour une fonction raisonnablement simple. Cependant, l'étude de ce chapitre est cruciale pour réussir à produire un code correct. En effet, dès que vous aurez le moindre doute lors de la rédaction d'une fonction (« dois-je mettre `i` ou `i+1` ? », « Faut utiliser `<` ou `<=` ? » etc.), vous pourrez utiliser les techniques que l'on va voir ci-dessous.

2.2 Boucles

En général, on utilise une récurrence pour chaque boucle du programme. On identifie une propriété telle que :

- *initialisation :* Elle est vraie au moment d'entrer dans la boucle
- *hérédité :* Si elle est vraie au moment de commencer une itération, alors les opérations effectuées pendant l'itération font qu'elle est encore vraie à la fin de l'itération.

D'après le théorème de la récurrence, une telle propriété est alors encore vraie en sortant de la boucle. On l'appelle alors un « invariant de boucle ».

2.2.1 Cas d'une une boucle « pour »

Dans une boucle inconditionnelle, on dispose déjà d'un compteur. Il semble donc naturel de l'utiliser comme variable de récurrence.

Pour une boucle de type `for i in range(0,n)`, j'écrirai ci-après « l'itération i » pour dire « lorsque le compteur `i` prend la valeur i ». Pour être rigoureux il faudrait faire la distinction entre la *variable* `i` et la valeur i qu'elle prend, qui est un nombre entier.

Remarque : Donc dans l'exemple présent où le compteur part de 0, cela signifiera « lors de la $(i+1)$-ème itération ».

En outre, pour tout $i \in [\![0, n-2]\!]$ la fin de l'itération i correspond au début de l'itération $i+1$. Pour $i = n-1$ par contre l'itération $i+1$ n'existe pas.

Ainsi, on définira nos prédicats de récurrence de la manière suivante : « $\forall i \in [\![0, n]\!]$, notons $P(i)$: " au début de l'itération i, …" ».

Étant entendu que pour tout $i \in [\![1, n-1]\!]$, le « au début de l'itération i » signifie aussi « à la fin de l'itération $i-1$ ». Pour $i = 0$, «au début de l'itération 0» signifie aussi «juste avant d'entrer dans la boucle». Enfin, pour $i = n$, «au début de l'itération n» signifiera en fait «à la fin de l'itération $n-1$», ou encore « à la sortie de la boucle».

Premier exemple : factorielle Reprenons la fonction suivante :

```
def facto(n):
    res=1
    for i in range(1,n+1):
        res*=i
    return res
```

La terminaison de cette fonction est évidente puisqu'elle ne contient qu'une brave boucle « pour ».

Passons à sa démonstration. Il s'agit d'analyser ce que contiennent à chaque instant les variables utilisées. Ici il n'y en a qu'une : `res`. Il n'est pas difficile de voir ce que contient cette variable à chaque instant, je le rajoute en commentaire dans le code :

```
def facto(n):
    res=1
    for i in range(1,n+1):
        # ici, res contient (i-1)!
        res*=i
        # Maintenant, res contient i!
    return res
```

Notons pour tout $i \in [\![1, n+1]\!]$, $P(i)$: « au début de l'itération i, `res` contient $(i-1)!$ ».

Pour aller plus vite, on peut utiliser la notation Python : « `res == (i-1)!` » signifie « `res` contient $(i-1)!$ ».

- **Initialisation :** Avant de commencer la boucle, `res` contient 1, or $(1-1)! = 1$, donc $P(0)$.
- **Hérédité :** Soit $i \in [\![1, n]\!]$, supposons $P(i)$. Ainsi, au début de l'itération i, `res` contient $(i-1)!$. On effectue l'itération i, qui consiste à exécuter `res*= i`. Alors `res` contient $(i-1)! \times i$, c'est-à-dire $i!$.
  Donc à la fin de l'itération i, c'est-à-dire au début de l'itération $i+1$, `res` contient $i!$, d'où $P(i+1)$.

En conclusion, pour tout $i \in [\![1, n+1]\!]$, $P(i)$.
En particulier, d'après $P(n+1)$, en sortant de la boucle, c'est-à-dire à la fin de l'itération n, `res` contient $n!$ d'après $P(n+1)$. Et c'est le résultat finalement renvoyé.

Calcul de e On reprend un exercice du premier TD, qui consiste à calculer, pour $n \in \mathbb{N}$, le nombre $\sum_{i=0}^{n-1} \frac{1}{i!}$. On utilise le code suivant :

```
def approx_e(n):
    res=0
    i_fact=1
    for i in range(n):
        res+=1/i_fact
        i_fact*=(i+1)
    return res
```

Il y a ici deux variables : `i_fact` et `res`. L'invariant de boucle consiste ici à indiquer ce que contiennent ces deux variables à chaque instant.

```
def approx_e(n):
    res=0
    i_fact=1
    for i in range(n):
        # Ici, i_fact== i!
        # et res == 1 + 1/2! + ... + 1/(i-1)!
        res+=1/i_fact
        i_fact*=(i+1)
        # Maintenant, i_fact == (i+1)!
        # et res == 1 + 1/2! + ... + 1/i!

    # Sortie de boucle : res == 1 + 1/2! + ... + 1/n!
    return res
```

N.B. Sans même rédiger une preuve complète, le fait d'écrire et de vérifier ces invariants de boucle permet de s'assurer que cette fonction est correct (le cas échéant de la corriger).

Remarque : Soit $i \in [\![0, n[\![$. $1 + 1/(2!) + ... + 1/((i-1)!)$ s'écrit aussi $\sum_{j=0}^{i-1} \frac{1}{j!}$. Lorsque $i = 0$, c'est une somme d'aucun terme, qui vaut donc 0.

Interlude Python : le « slicing » Soit `t` un tableau. Alors pour tout $(d, f) \in [\![0, len(t)]\!]^2$, `t[d:f]` renvoie le sous-tableau `[t[d], t[d+1], ..., t[f-1]]`.
Cette syntaxe pourra être utile de temps en temps dans les programmes, mais surtout sera une notation très pratique sur papier, ou dans les commentaires, pour analyser un programme.

En programmation, on n'abusera pas de cette commande car elle effectue une *copie* de la partie du tableau concernée, ce qui consomme du temps et de la mémoire. L'exécution de `t[deb:fin]` nécessite de recopier `fin-deb` valeurs ailleurs dans la mémoire.

Exemple avec un tableau : `estTrié`

```
def estTrié(t):
    res=True
    for i in range(1,n):
        # Ici, res ssi t[0:i] est trié
        if t[i]<t[i-1]:
            res=False
    return res
```

Soit `t` un tableau et n sa longueur.

Notons pour tout $i \in [\![1, n]\!]$, $P(i)$: « En entrée d'itération i, on a `res` $\Leftrightarrow$ `t[0 : i]` est trié ».

Remarque : `res` $\Leftrightarrow$ `t[0 : i]` est trié » signifie que « `res` » et « `t[0:i]` est trié » ont la même valeur de vérité. Si l'un est vrai, l'autre doit l'être aussi, si l'un est faux, l'autre doit l'être aussi. En fait le symbole $\Leftrightarrow$ est juste l'égalité entre deux booléens. En Python on écrirait juste `res == estTrié(t[0 :i])`.

- **Initialisation :** Initialement, `res` est vrai, et « `t[0:1]` est trié » l'est aussi. D'où $P(1)$.
- **Hérédité :** Soit $i \in [\![1, n[\![$. Supposons $P(i)$. Distinguons selon les deux cas du «if».
  - ◇ Si `t[i]<t[i-1]` : alors `t[0 :i+1]` n'est pas trié. Or `res` passe à Faux. D'où $P(i+1)$.
  - ◇ Si `t[i-1] <= t[i]` : dans ce cas, `res` ne change pas. Or :
    - ◦ Si `t[0 :i]` était trié, `res` était vrai par $P(i)$. Le fait que $t[i-1] \leqslant t[i]$ nous assure alors que `t[0 :i+1]` est trié, et `res` est resté vrai, donc $P(i+1)$ est vrai.
    - ◦ Sinon `t[0 :i]` n'était pas trié, donc `t[0 :i+1]` ne l'est pas non plus. Or `res` est resté Faux. Donc $P(i+1)$ est vrai aussi.

  Dans tous les cas, $P(i+1)$.

Dès lors par récurrence, $P(n)$ est vrai, ce qui signifie que `t[0 :n]`, càd `t`, est trié ssi `res` est vrai.

2.2.2 Cas d'une boucle « tant que »

Pour introduire cette partie, citons ce proverbe :

Théorème 2.1. *(loi des boucles « tant que »)*
Une boucle « tant que » est fausse.

puis sa version complète :

Théorème 2.2. *(loi des boucles « tant que »)*
Une boucle « tant que » est fausse, à moins que son invariant de boucle n'ait été explicité.

Bien que dans le fond le principe soit le même que pour une boucle « pour », c'est un peu plus compliqué à rédiger car on ne dispose pas du compteur de boucle sur lequel faire notre récurrence. On peut la faire sur le nombre d'itérations effectuées. Une des difficultés est qu'on ne sait pas à l'avance combien d'itérations il y a. On peut donner un nom au nombre d'itérations ; si on n'a pas encore prouvé que la boucle termine, ce nombre peut valoir ∞.

Une autre particularité importante : pour conclure la démonstration, il y a deux informations à utiliser :

1. L'invariant de boucle ;
2. Le fait que la condition de la boucle ne soit plus vrai.

En conséquence, au moindre doute lors de l'écriture d'une boucle while, on conseille d'écrire en commentaire au moment de la sortie de boucle la négation de la condition de la boucle, et de vérifier si la conjonction de ceci et de l'invariant de boucle permet bien de conclure que le résultat final renvoyé par la fonction est correct.

Maintenant, traitons quelques exemples.

Plus petit diviseur

```
def plusPetitDiviseur(n):
    """ n doit être un entier n'appartenant pas à [-1,1].
    Ceci renvoie le plus petit diviseur de n dans l'intervalle [2,∞[.
    """
    res=2
```

```
    while n%res!=0:
        res+=1
    return res
```

Il s'agit de trouver une propriété, vérifiée à chaque tour de boucle, qui explique le rôle des variables utilisées (ici, il n'y a que `res`). Dans cet exemple, ce que nous savons à chaque instant sur la variable `res` c'est que tous les entiers testés auparavant, c'est-à-dire les éléments de $[\![2, \infty[\![$, ne divisait pas n. Je l'indique précisément en commentaire dans la fonction. J'indique également les informations disponibles en sortie de boucle.

```
def plusPetitDiviseur(n):
    """ n doit être un entier n'appartenant pas à [-1,1].
    Ceci renvoie le plus petit diviseur de n dans l'intervalle [2, \infini[.
    """
    res=2
    while n%res!=0:
        # Ici, les éléments de [| 2, res |] ne divisent pas n.
        res+=1
        # Ici, les éléments de [| 2, res [| ne divisent pas n.
    # Sortie de boucle :
    # les éléments de [|2;res[| ne divisent pas n
    # et res divise n
    return res
```

On choisit donc l'invariant de boucle suivant : « À la fin de chaque tour de boucle, les éléments $[\![2, res[\![$ ne divisent pas n ».
Remarque : Je trouve plus commode d'utiliser un invariant de boucle en entrée pour une boucle « pour » et en sortie pour un « tant que ».

- **Notations :** Je propose ci-dessous une rédaction complète, qui nécessite de prendre un certain nombre de notations :
  - N est le nombre d'itérations de la boucle. A priori, $N \in \mathbb{N} \cup \{\infty\}$.
  - $\forall i \in [\![0, N]\!]$, r_i est le contenu de la variable `res` après i itération. En particulier, r_0 est le contenu de `res` avant de commencer la boucle. En outre, si (par malchance) $N = \infty$, alors nous ne définissons pas r_N.
  - $\forall i \in [\![0, N]\!]$, on pose $P(i)$: « Les éléments de $[\![2, r_i[\![$ ne divisent pas n » .

- **Récurrence :** Passons à la démonstration par récurrence.
  - *Initialisation :* Avant de rentrer dans la boucle, on a `res == 2`. Donc $r_0 = 2$, et $[\![2, r_0[\![= \emptyset$, et la phrase « les éléments $[\![2, r_0[\![$ ne divisent pas n » est vraie.
  - *Hérédité :* Soit $i \in [\![0, N-1]\!]$, supposons $P(i)$. On exécute l'itération $i+1$ de la boucle (on sait que l'itération $i+1$ a lieu car on a pris $i < N$). Le simple fait qu'elle s'exécute signifie, vu la condition de la boucle, que $r_i \not| n$. En outre, d'après $P(i)$, nous savons que $[\![2, r_i[\![$ ne contient pas de diviseur de n.
    Ces deux informations donnent que $[\![2, r_i + 1[\![$ ne contient pas de diviseur de n.
    Mais $r_i + 1 = r_{i+1}$, d'où $P(i+1)$.

  Par le théorème de récurrence, pour tout $i \in [\![0, N]\!]$, $P(i)$.

- **Terminaison de la boucle :**

  Remarque : Cette exemple est un peu spécial concernant l'ordre des différentes étapes. En effet, c'est maintenant, grâce à la preuve de l'invariant de boucle que nous venons de faire, que nous pouvons prouver la terminaison. Faisons-le :

  On considère la suite $\left(n - r_i\right)_i$.
  - Elle est entière.
  - Elle diminue strictement (de 1) à chaque tour de boucle.
  - Nous savons que pour tout i, $[\![2, r_i[\![$ ne contient pas de diviseur de n. Donc cet intervalle ne contient pas n. Donc $r_i \leqslant n$, et $\mathtt{n - r_i} \geqslant 0$.

Nous avons bien trouvé un variant de boucle convenable, donc d'après le théorème sur l'arrêt des boucles conditionnelles, la boucle termine.

- **Conclusion :** Nous savons maintenant que la boucle termine. En sortie de boucle, nous savons alors :
  - $\forall i \in [\![2, \texttt{res}[\![, \ i \not| n$;
  - $\texttt{res} | n$.

  Cela signifie précisément que `res` est le plus petit diviseur de n.

Version améliorée de la recherche dans un tableau

```
def appartient(x,t):
    trouvé=False
    i, n = 0, len(t)
    while not trouvé and i<n:
        if t[i]==x:
            trouvé=True
        i+=1
    return trouvé
```

1. **Terminaison :** La quantité $n - i$ est entière, positive, et strictement décroissante. Donc la boucle termine.
2. **Correction :**
   (a) *Définition de l'invariant de boucle :*

   Les deux variables utilisées sont `trouvé` et `i`. Comme `trouvé` est un booléen, il s'agit de savoir quand est-ce qu'il est vrai.

   On va utiliser l'invariant de boucle suivant : « À la fin de chaque itération, `trouvé` est vrai ssi $x \in t[0 : i]$ ».

   En plus court, `trouvé` $\Leftrightarrow x \in t[0 : i]$.
   En fait, dire que deux booléens sont équivalents revient à dire qu'ils sont soit tous les deux vrais soit tous les deux faux, autrement dit qu'ils sont égaux. Au final le symbole $\Leftrightarrow$ n'est rien d'autre que l'égalité pour des booléens...

   (b) *Preuve par récurrence de l'invariant de boucle.*
   Notons :
   - N le nombre d'itérations.
   - Pour tout $k \in [\![0, N]\!]$, t_k, et i_k le contenu de `trouvé` et de `i` après k itérations.

     Remarque : En réalité, on a $\forall k \in [\![0, N]\!]$, $i_k = k$, mais ce n'est pas nécessaire de le prouver.
   - $\forall k \in [\![0, N]\!]$, $P(k)$: « $t_k \Leftrightarrow x \in t[0 : i_k]$ »

   Passons à la preuve de l'invariant de boucle :
   - **Initialisation :** On a $i_0 = 0$, donc $t[0 : i_0] = \emptyset$, donc $x \in t[0 : i_k]$ est faux. Par ailleurs, t_k est faux. Donc t_k et $x \in [\![0 : i_k]\!]$ ont la même valeur de vérité. Donc l'équivalence $P(0)$ est vérifiée ici.
   - **Hérédité :** Soit $k \in [\![0, N - 1]\!]$, supposons $P(k)$. On effectue l'itération k.
     - *Si* `t[i]==x` : Alors `trouvé` devient Vrai, c'est-à-dire $t_{k+1} = \top$, et $x \int [0 : i + 1]$ est vrai aussi. On a bien $t_{k+1} \Leftrightarrow x \int [0 : i + 1]$.
     - *Sinon* : Dans ce cas, $x \in t[0 : i + 1] \Leftrightarrow x \in t[0 : i]$ puisque x n'est pas dans $t[i]$. Et par ailleurs, $t_{k+1} = t_k$ puisqu'on n'a pas touché à `trouvé`. Or par hypothèse de récurrence, $t_k \Leftrightarrow x \int [0 : i]$.
       En assemblant ces équivalences, on arrive bien à $t_{k+1} \Leftrightarrow x \int [0 : k + 1]$.

   Par récurrence, pour tout $k \in [\![0, N]\!]$, $P(k)$.

   (c) *Conclusion de la preuve :* À la fin du programme, on sait que :
   - (H1) `trouvé` est vrai ssi $x \in t[0 : i]$ *(invariant de boucle)*
   - (H2) `trouvé` est vrai ou $i = n$ *(car la boucle est finie).*

   On va considérer les deux cas possibles selon la valeur de `trouvé` :
   - Si `trouvé` alors $x \in t[0 : i]$ par (H1) et donc en particulier $x \in t$.
   - Si pas `trouvé`, alors $i = n$ par (H2), et $x \notin t[0 : i]$ par (H1). De (H2) on déduit que $t[0 : i] = t$, et alors (H1) donne $x \notin t$.

   En conclusion, on a bien `trouvé` $\Leftrightarrow x \in t$, donc `trouvé` contient le résultat attendu de la fonction.

Division euclidienne **cf exercice :** 2

Algorithme d'Euclide Reprenons l'exemple de l'algorithme d'Euclide. Pour tout $(a,b) \in \mathbb{Z}^2 \setminus \{(0,0)\}$, on notera $a \wedge b$ le pgcd de a et b.

L'algorithme d'Euclide est basé sur les deux résultats suivants (vus en spécialité math en terminale, et dans le chapitre d'arithmétique pour les MPSI).

Proposition 2.3. *Soit $a \in \mathbb{Z}^*$. Alors $a \wedge 0 = a$.*

Proposition 2.4. *(lemme d'Euclide)*
Soit $(a,b,q,r) \in \mathbb{Z}^4$ tels que $a = bq + r$. Alors :

$$a \wedge b \quad = \quad b \wedge r.$$

Rappel de l'algorithme :

Entrées : a,b entier, tel que $b \neq 0$
Sorties : $a \wedge b$, le pgcd de a et b
Variables locales : x,y entiers
début
  $x \leftarrow a$
  $y \leftarrow b$
  tant que $b \neq 0$ **:**
    $x \leftarrow y$
    $x \leftarrow$ reste de la division euclidienne de x par y.
  fin
  renvoyer x
fin

Algorithme 2 : algorithme d'Euclide

Fixons $(a,b) \in \mathbb{Z}^2$ tel que $b \neq 0$, et démontrons à présent que l'algorithme d'Euclide appliqué à a et b renvoie bien $a \wedge b$.

Nous allons démontrer que la propriété « $x \wedge y = a \wedge b$ » est un invariant de la boucle « tant que » de l'algorithme d'Euclide.

Pour tout n inférieur au nombre d'itérations effectuées, notons x_n, y_n le contenu des variables x et y après n itérations. Notons également q_n, r_n quotient et reste de la division euclidienne de x_n par y_n.

- **Initialisation :** $x_0 \wedge y_0 = a \wedge b$ d'après l'initialisation de `x` et `y`.
- **Hérédité :** Soit $n \in \mathbb{N}$ inférieur au nombre d'itérations effectuées. On a : $\begin{cases} x_{n+1} = y_n \\ y_{n+1} = r_n \\ x_n = q_n y_n + r_n \end{cases}$
  D'après la proposition 2, on a $x_n \wedge y_n = y_n \wedge r_n$. Ce qui donne précisément que $x_{n+1} \wedge y_{n+1} = x_n \wedge y_n$. Alors $x_{n+1} \wedge y_{n+1} = a \wedge b$ d'après l'hypothèse de récurrence.

En conclusion, la propriété « $x \wedge y = a \wedge b$ » est bien un invariant de boucle. En particulier, elle est encore vraie en sortie de boucle.

Or, à l'issue de la boucle, on a $y \leq 0$ (la condition du « tant que » n'étant plus vérifiée), mais aussi $y \geq 0$ car un reste de division euclidienne est toujours ≥ 0. Donc $y = 0$. Donc $x \wedge y = x \wedge 0 = x$ par la proposition 1. Comme on renvoie x, on a bien renvoyé le bon résultat.

Commentaires, bilan En résumé, voici les grandes étapes lors de l'analyse d'une fonction basée sur une boucle conditionnelle :

1. Démontrer la terminaison. Pour ce, on exhibe une quantité entière, positive et strictement décroissante (un « variant de boucle »).

2. Démontrer la correction. Ceci peut être découpé en plusieurs étapes :
   (a) Trouver l'invariant de boucle pertinent. En général, il s'agit de dire à quoi servent vos variables. Plus précisément, que contiennent vos variables à chaque étape.
   (b) Démontrer qu'il s'agit bien d'un invariant de boucle.
   (c) Conclure. Attention à ne pas sauter cette étape ! La conclusion utilisera en général :
      i. L'invariant de boucle ;
      ii. Le fait qu'on est sorti de la boucle : donc la condition derrière le « while » est fausse.
3. Nous verrons plus tard une étape supplémentaire : en analysant plus en détail la manière dont le variant de boucle décroît, il est possible d'estimer le nombre d'itérations effectuées par la boucle, et donc d'estimer le nombre d'opérations effectuées.

3 Exemple plus complexe : recherche dichotomique

3.1 Rappel du programme

```
def appartient_dicho(x, t):
    deb, fin = 0, len(t)
    trouvé = False
    while not trouvé and deb<fin:
        m = (deb+fin)//2
        if t[m]<x:
            deb=m
        elif t[m]>x:
            fin=m
        else:
            trouvé=True
    return trouvé
```

3.2 Terminaison

Fixons un tableau t et en élément x. On exécute l'algorithme sur t et x.
On utilise la quantité `f-d` comme variant de boucle.

- C'est toujours un nombre entier.
- Il est toujours positif à cause de la condition de la boucle "tant que".
- Reste à vérifier qu'il décroît strictement à chaque tour de boucle.

Concernant la division euclidienne, ce petit lemme figure dans le cours de math de MPSI :

Lemme 3.1. *Pour tout* $(a,b) \in \mathbb{N} \times \mathbb{N}^*$, $a//b = \left\lfloor \frac{a}{b} \right\rfloor$.

Vérifions que `f-d` décroît strictement à chaque itération.
Notations :

- n_0 le nombre d'itérations (peut-être ∞ a priori) ;
- Pour tout $i \in [\![0, n_0]\!]$ (exclure n_0 si c'est ∞), f_i, d_i, m_i le contenu des variables `d, f, m` après i itérations.

Calcul :
Soit $i \in [\![0, n_0[\![$. On a $m_{i+1} = \left\lfloor \frac{d_i+f_i}{2} \right\rfloor$.

- *Si* $t[m_{i+1}] = x$: la boucle s'arrête. Ce cas est en fait impossible ici puisque nous avons pris $i < n_0$.
- *Si* $t[m_{i+1}] > x$: on a $d_{i+1} = d_i$ et $f_{i+1} = m_{i+1} = \left\lfloor \frac{d_i+f_i}{2} \right\rfloor$. Ainsi :

$$\begin{aligned} f_{i+1} - d_{i+1} &= \left\lfloor \frac{d_i+f_i}{2} \right\rfloor - d_i \\ &\leqslant \frac{d_i+f_i}{2} - d_i \\ &= \frac{f_i-d_i}{2} \end{aligned}$$

Or, puisqu'on est encore dans la boucle while, $f_i - d_i > 0$, d'où on déduit que $\frac{f_i-d_i}{2} < f_i - d_i$.

- *Si* $t[m_{i+1}] < x$ *:* on a $d_{i+1} = m_{i+1} = \frac{d_i+f_i}{2}$ et $f_{i+1} = f_i$. D'où :

$$\begin{aligned} f_{i+1} - d_{i+1} &= f_i - \left\lfloor \tfrac{d_i+f_i}{2} \right\rfloor \\ &< f_i - \tfrac{d_i+f_i}{2} + 1 \\ &< \tfrac{f_i-d_i}{2} + 1 \end{aligned} \quad \Big) \forall x \in \mathbb{R},\ \lfloor x \rfloor > x - 1$$

  On ne voit pas comment arriver à $f_{i+1} - d_{i+1} < f_i - d_i$... Et de fait, lorsque $f_i - d_i = 1$, l'inégalité ci-dessus donne $f_{i+1} - d_{i+1} < \frac{3}{2}$: cette inégalité est insuffisante pour conclure.
  Mais la situation est même pire que ça : lorsque $f_i - d_i = 1$, on a $m_{i+1} = d_i$. Et donc pour peu qu'on soit dans le cas où $t[m_{i+1}] < x$, on aura $d_{i+1} = d_i$ et $f_{i+1} = f_i$. À partir de là, la situation n'évolue plus jamais : la programme plante.

De faire la preuve de la terminaison nous a donc permi de nous apercevoir que l'algorithme ne termine pas dans certains cas. Corrigeons-le. L'idée est que lorsque $t[m] < x$, nous savons que $x \neq t[m]$, et nous pouvons poursuivre la recherche dans $t[m+1 : f]$ plutôt que dans $t[m : f]$. Ceci permet d'être sûr d'exclure au moins une case de la zone de recherche.

Voici alors le programme corrigé.

```
def appartient_dicho(x, t):
    deb, fin = 0, len(t)
    trouvé = False
    while not trouvé and deb<fin:
        m = (deb+fin)//2
        if t[m]<x:
            deb=m+1
        elif t[m]>x:
            fin=m
        else:
            trouvé=True
    return trouvé
```

Et voici la fin de la preuve de la terminaison. Nous étions dans le cas où $t[m_{i+1}] < x$. Dansle code corrigé, $f_{i+1} = f_i$ et $d_{i+1} = m_{i+1} + 1 = \left\lfloor \frac{d_i+f_i}{2} \right\rfloor + 1$. Alors :

$$\begin{aligned} f_{i+1} - d_{i+1} &= f_i - \left\lfloor \tfrac{d_i+f_i}{2} \right\rfloor - 1 \\ &< f_i - \tfrac{d_i+f_i}{2} \\ &= \tfrac{f_i-d_i}{2} \end{aligned}$$

Et comme $f_i - d_i > 0$, on a bien $f_{i+1} - d_{i+1} < f_i - d_i$.

Ceci achève la preuve du fait que `f-d` est un variant de boucle, et donc que l'argorithme termine.

3.3 Correction

Le point est d'exprimer clairement quand est-ce que `trouvé` est vrai, et quand est-ce qu'il est faux.
On utilise la propriété suivante comme invariant de boucle :

$$\begin{cases} \texttt{trouvé} \Rightarrow x \in \texttt{t}. \\ \neg\texttt{trouvé} \Rightarrow x \notin t \setminus t[d : f] \end{cases}$$

Remarque : `t[d:f]` est la zone qu'il reste encore à étudier.

Passons à la rédaction formelle. On garde les notations n_0, d_i, f_i, m_i précédentes.
On pose alors pour tout $i \in [\![0, n_0]\!]$, $P(i)$:« $\begin{cases} t_i \Rightarrow x \in \setminus t \\ \neg t_i \Rightarrow x \notin t \setminus t[d_i : f_i] \end{cases}$. ».

- **Initialisation :** En entrée de boucle :

- t_0 est Faux
- $d_0 = 0$ et $f_0 = n$, donc $t \setminus t[d_0 : f_0] = []$, donc $x \in t \setminus t[d_0 : f_0]$ est faux.

Donc $P(0)$ est vrai.

- **Hérédité :** Soit $i \in [\![0, n_0 - 1]\!]$, supposons $P(i)$. On effectue l'itération $i+1$.
  Le fait même qu'on rentre dans la boucle signifie que `trouvé` est faux, et donc d'après $P(i)$, $x \notin t \setminus t[d_i : f_i]$. Vu la première ligne exécutée, on a $m_{i+1} = \left\lfloor \frac{d_i + f_i}{2} \right\rfloor$. traitons séparément les trois cas qui se présentent alors :
  - *cas 1 :* $t[m_{i+1}] == x$
    Dans ce cas t_{i+1} est vrai, or $x \in t$, donc $P(i+1)$ est bien vérifié.
  - *cas 2 :* $t[m_{i+1}] < x$
    Dans ce cas, x est également strictement supérieur à $t[0], \dots t[m_{i+1}]$ puisque t est trié. Donc $x \notin t[0 : m_{i+1}+1]$.
    Et par hypothèse de récurrence, il n'est pas non plus dans $t[f_i : n]$.
    Au final, il n'est pas dans $t \setminus t[m_{i+1} : f_i]$.
    Or, vu la seule instruction exécutée dans ce cas, on a $d_{i+1} = m_{i+1}$ et $f_{i+1} = f_i$.
    Donc $x \notin t \setminus t[d_{i+1} : f_{i+1}]$. Et donc $P(i+1)$
  - *cas 3 :* $t[m_{i+1}] > x$
    similaire au cas 2.

  En conclusion de la récurrence, pour tout $i \in [\![0, n_0]\!]$, $P(i)$. En particulier par $P(n_0)$, la propriété est vraie à la fin du programme. Voyons en quoi ceci nous assure que la valeur renvoyée est la bonne. Plaçons-nous à la fin du programme, nous avons alors :
  - *cas 1, si* `trouvé` *:*
    D'après $P(n_0)$ c'est que $x \in t$.
  - *cas 2, si non* `trouvé` *:*
    D'après $P(n_0)$, c'est que $x \notin t \setminus t[d : f]$. Mais d'après le fait que la condition de la boucle n'est plus vérifiée, on a $d = f$. Donc $t[d : f] = []$, et $t \setminus t[d : f] = t$.
    Au final, $x \notin t$.

  On constate ainsi, que `trouvé` est vrai si et seulement si $x \in t$, et comme c'est la valeur renvoyée, la fonction `chercheDicho` a bien le comportement prévu.

3.4 Fonction récursive

L'étude d'une fonction récursive est plus simple : on prouve directement par récurrence un prédicat du type $P(n)$: « La fonction appelée sur une entrée de taille n termine et renvoie le bon résultat. »

3.4.1 Factorielle

3.4.2 Calcul de puissance par dichotomie

Voir la preuve faite au chapitre sur la dichotomie.

3.4.3 Recherche dans un tableau trié

Pour écrire une version récursive de la recherche dichotomique dans un tableau trié, on écrit une fonction auxiliaire qui prend deux arguments supplémentaires `deb` et `fin` qui indiquent la plage du tableau où on recherche.

```
def appartientEntre(x, t, deb, fin):
    """
    Indique si x ∈ t[deb : fin].
    """
    if deb>=fin:
        return False
    else:
        m = (deb+fin)//2
        if x<t[m]:
            return appartientEntre(x, t, m+1, fin)
        elif x>t[m]:
            return appantient_Entre(x, t, deb, m)
        else:
            return True
```

Pour prouver la correction de cette fonction, on procède par récurrence sur la taille de la zone de recherche, c'est-à-dire `fin-deb`.

Fixons un tableau t, sa longueur n et un élément x. On définit le prédicat suivant : $P : k \mapsto$ « Pour tout $(\texttt{deb}, \texttt{fin}) \in [\![0, n[\![^2$ tel que `fin-deb==k`, `appartientEntre(x, t, deb, fin)` termine et renvoie `True` ssi $x \in t[deb : fin]$.

- **Initialisation :** Soit $(\texttt{deb, fin}) \in [\![0, n[\![^2$ tel que $\texttt{fin-deb} = 0$. Alors `t[deb:fin]==[]` donc $x \in t[deb : fin]$ est faux. Or `appartientEntre(x, t, deb, fin)` renvoie `False`. Donc cet appel termine et renvoie le bon résultat. Ainsi $P(0)$ est vrai.
- **Hérédité :** Soit $k \in \mathbb{N}$. Supposons $\forall j \in [\![0, k]\!]$, $P(j)$. Soit $(deb, fin) \in [\![0, n[\![^2$ tel que $fin - deb = k + 1$. Soit $m = \left\lfloor \frac{deb+fin}{2} \right\rfloor$ comme dans le code. On traite les trois cas :
  - ◇ Si $x < t[m]]$:

    Déjà, la programme renvoie alors `appartientEntre(t, deb, m)`. Or $m - deb = \left\lfloor \frac{deb+fin}{2} \right\rfloor - deb \leqslant \frac{fin-deb}{2} < fin - deb$ car $fin - deb \neq 0$. Ainsi par hypothèse de récurrence, $P(m - deb)$ est vrai. En particulier, `appartientEntre(t, deb, m)` termine. Et donc `appartientEntre(t, deb, fin)` termine aussi dans ce cas. Voyons maintenant si le bon résultat est renvoyé.

    Comme t est trié, nous savons alors que x n'est pas dans $t[m :]$. Par conséquent :

    $$\begin{aligned} x \in t[deb : fin] &\Leftrightarrow x \in t[deb : m] \\ &\Leftrightarrow appartientEntre(t, deb, m) \end{aligned}$$ *Par hypothèse de récurrence $P(m - deb)$*

    Or `appartientEntre(t, deb, m)` est précisément le résultat renvoyé par `appartientEntre(t, deb, fin)` dans ce cas. C'est donc bien le bon résultat.
  - ◇ Si $x > t[m]$: similaire.
  - ◇ Si $x = t[m]$: alors $x \in t[deb : fin]$. Or le résultat renvoyé est `True` : c'est le bon.

Ainsi par le théorème de récurrence, pour tout $k \in \mathbb{N}$, $P(k)$. D'où on déduit que `appartientEntre` termine et est correct pour toutes ses entrées telles que $fin - deb \in \mathbb{N}$.

Deuxième partie

Exercices

Exercices : preuves d'algorithmes

1 Boucles conditionnelles

Exercice 1. * Exemples d'invariants de boucle « pour »

Pour chacune des fonctions suivantes, vues dans les TP précédents, écrire un invariant de boucle. On pourra le taper en commentaire, directement dans le code.

Puis faire la rédaction complète de la correction d'une de ces fonctions.

1. Calcul de factorielle.
2. Tester si un nombre est premier.
3. Tester si un tableau est trié dans l'ordre croissant.
4. Calcul du maximum d'un tableau.
5. Recherche d'un élément dans un tableau.

2 Boucles inconditionnelles

Exercice 2. **! Étude complète d'une fonction mystère

On considère l'algorithme suivant, donné ici dans le langage Python :

```
def f(a,b):
    x=0
    y=a
    while y>=b:
        x+=1
        y-=b
    return(x,y)
```

1. Choisir des valeurs simples pour les arguments, et suivre l'exécution de cet algorithme pas à pas.
2. Deviner ce que calcule cette fonction. Quels sont les arguments et les variables employées ? Deviner leur type et leur utilité.
3. *(!) Rendre le code lisible :* Changer le nom de la fonction, le nom des variables et rajouter une documentation.
4. Quelles sont la ou les conditions sur les arguments pour que cet algorithme termine ? Dans le cas où ces conditions sont vérifiées, démontrer que l'algorithme termine.
5. Rajouter dans le corps de la boucle un commentaire indiquant l'invariant de boucle permettant de démontrer que cette fonction renvoie bien le résultat conjecturé. Ce sera une égalité qui doit être vérifiée à chaque instant entre `a`, `b`, `x` et `y`.
6. Démontrer que cette fonction renvoie bien le résultat espéré.
7. Combien de fois s'exécute la boucle ?
8. (bonus) Modifier cet algorithme pour qu'il fonctionne aussi lorsque $b < 0$ ou $a < 0$.

Exercice 3. ** Une fonction inutile

Soit la fonction `f` suivante :

```
import numpy.random as rd
def f(t):
    r=0
    i=0
    while i< len(t):
        r = 2*r + 3 *r**2
        if rd.randint(0,2)==0:
            t.pop()
        else:
            i+=1
    return r
```

1. Démontrer que `f` termine.

2. Combien de fois s'exécute la boucle ?
3. Que renvoie `f` ? Le démontrer.

Exercice 4. * Décomposition en facteurs premiers**

1. Écrire ou récupérer du TP précédent une fonction `plusPetitDiviseur` renvoyant le plus petit diviseur supérieur à 2 d'un entier, et une fonction `decomposition` prenant en entrée un entier n et renvoyant la liste des facteurs premiers de n.
2. Démontrer que cet algorithme termine.
3. Démontrer que cet algorithme renvoie le résultat correct. Dans un premier temps, on pourra admettre que `plusPetitDiviseur` fonctionne et se concentrer sur `decomposition`.

3 Fonctions récursives

Exercice 5. ** maximum par diviser pour régner

Démontrer que la fonction de calcul de maximum basé sur le principe « diviser pour régner » du TD sur la dichotomie termine et renvoie la bon résultat.

Exercice 6. * Une fonction inutile

On considère la fonction `f` suivante :

```
def f(n):
    if n==0:
        return 0
    else:
        res=0
        for i in range(n):
            res += f(n-1)
        return res
```

1. Que renvoie f ? Le démontrer par récurrence.
2. On note pour tout $n \in \mathbb{N}$, C_n le nombre d'additions pour exécuter `f(n)`.
   (a) Donner C_0 ainsi que la relation de récurrence vérifiée par la suite C.
   (b) Démontrer que pour tout $n \in N^*$, $C_n \geqslant n!$. Commentaires sur cette fonction ?
   (c) *Bonus :* écrire une fonction plus simple pour calculer la même chose que `f`.

Quelques indications

1 Il s'agit essentiellement d'indiquer ce que contiennent les variables utilisées.

3
1. Utiliser la quantité `len(t)-i`.
2. Vérifier que $r = 0$ est un invariant de boucle.

4
1.
2. Avec les notations de l'algorithme, la suite des valeurs successives de $|k|$ est une suite d'entiers positifs strictement décroissante.
3. Utiliser l'invariant de boucle suivant (en remplaçant `res` par le nom de variable que vous avez utilisé) : « *res* ne contient que des nombres premiers et $n = k \times \prod_{x \in \texttt{res}} x$ ».

Quelques solutions

1 1. Rédaction complète pour le calcul de puissance, en se basant sur la fonction suivante :

```
def puissance(x,n):
    res=1
    for i in range(0,n):
        # ici, res contient x**i
        res*=x
        #maintenant, res contient x**(i+1)
    #sortie de boucle: res contient x**(n-1+1) càd x**n
    return res
```

Posons pour tout $i \in [\![0, n]\!]$, $P(i)$: « au début de l'itération i de la boucle, et à la fin de l'itération $i-1$, `res` contient x^i. »

- *Initialisation :* En entrée du premier tour de boucle (pour $i = 0$), `res` contient 1. Or $1 = x^0$ donc $P(0)$.
- *Hérédité :* Soit $i \in [\![0, n-1]\!]$, supposons $P(i)$. Donc en entrée du tour de boucle i, `res` contient x^i. On effectue alors le corps de la boucle : `res*=x`, à la suite de quoi `res` contient x^{i+1}. Donc au début du tour de boucle $i+1$, `res` contient x^{i+1}, d'où $P(i+1)$.

En conclusion, pour tout $i \in [\![0, n]\!]$, $P(i)$. En particulier, $P(n)$ nous apprend qu'à la fin du tour $n-1$, c'est-à-dire du dernier tout de boucle, `res` contient x^n. Or c'est ce résultat qui est renvoyé par la fonction, et c'est bien le bon résultat.

2. **Calcul de factorielle**

```
def factorielle(n):
    res=1
    for i in range(1,n+1):
        # ici, res contient (i-1)!
        res*=i
        # maintenant, res contient i!
    return res
```

Soit $n \in \mathbb{N}$, effectuons `factorielle(n)`. Pour tout $i \in [\![1, n+1]\!]$, posons $P(i)$: « en entrée de l'itération i, et en sortie de l'itération $i-1$, `res` contient $(i-1)!$ ».

- **initialisation :** en entrant dans la boucle `res` contient 1. Or $(1-1)! = 1$. Donc $P(1)$ est vrai.
- **hérédité :** Soit $i \in [\![1, n]\!]$, supposons $P(i)$. Donc en entrée de l'itération i, `res` contient $(i-1)!$. On effectue alors `res*=i` : `res` contient maintenant $(i-1)! \times i$, c'est-à-dire $i!$.
  Ainsi, en fin d'itération i, et donc en début d'itération $i+1$, `res` contient $i!$. D'où $P(i+1)$.

Par récurrence, $\forall i \in [\![1, n+1]\!]$, $P(i)$. En particulier, d'après $P(n+1)$, à la fin de la dernière itération (l'itération n), `res` contient $n!$, qui est le bon résultat.

2 1.

2. Cette fonction calcule quotient et reste de la division euclidienne de a par b. La variable `x` est appelée à contenir le quotient, et `y` le reste.

3.
```
def divisionEuclidienne(a,b):
    """
    précondition : b>0
    renvoie (quotient,reste) de la division euclidienne de a par b.
    """

    quotient=0
    reste=a
    while reste>=b:
        # ici, a = a*quotient + reste
        quotient+=1
        reste-=b
    return(x,y)
```

4. Cette fonction termine à condition que $b > 0$.

5. La quantité `reste -b` est entière, strictement positive, et diminue de b à chaque itération. Comme $b > 0$, elle diminue donc strictement à chaque itération.
   Par le théorème de terminaison des boucles « tant que », la fonction `divisionEuclidienne` termine.

6.

7. Notons n_0 le nombre d'itérations de la boucle. Pour tout $i \in [\![0, n_0]\!]$, notons q_i, r_i le contenu de `quotient` et `reste` en entrée de l'itération i, et à la fin de l'itération $i-1$, et posons $P(i)$: « $a = b \times q_i + r_i$ » .
   - Initialement, $q_0 = 0$ et $r_0 = a$, donc l'égalité $a = b \times q_0 + r_0$ est vraie.
   - Soit $i \in [\![0, n_0 - 1]\!]$, supposons $P(i)$. On effectue l'itération i. Vu les deux lignes exécutées, on a $q_{i+1} = q_i + 1$ et $r_{i+1} = r_i - b$. Dès lors :

$$\begin{aligned} b \times q_{i+1} + r_{i+1} &= b \times (q_i + 1) + r_i - b \\ &= b \times q_i + r_i \\ &= a \end{aligned} \quad \text{par } P(i)$$

     On constate que $P(i+1)$ est vraie.

   Ainsi, pour tout $i \in [\![0, n_0]\!]$, $P(i)$. En particulier, d'après $P(n_0)$, on a en sortie de la boucle `a = b*quotient + reste`.
   De plus, comme la condition de la boucle n'est plus vraie, `reste < b`.
   Enfin, comme la condition de la boucle était vérifiée au début de la dernière itération, on avait $r_{n_0-1} \geqslant b$. On a ensuite soustrait b à r_{n_0-1}, de sorte qu'en sortant de la boucle on a `reste >=0`.
   Au final, on a $\begin{cases} \texttt{a=b*quotient+reste} \\ \texttt{reste} \in [\![0, b[\![\end{cases}$, donc `(quotient, reste)` est bien le résultat de la division euclidienne de a par b.

3

1. La quantité `len(t)-i` est entière, positive, et diminue de 1 à chaque itération (dans un cas c'est `i` qui augmente de 1, dans l'autre c'est `len(t)` qui diminue de 1).
   Donc par le théorème de terminaison des boucles « tant que », `f` termine.
2. Notons n_0 la longueur initiale du tableau `t` passé en argument. La quantité `len(t)-i` part de n_0, diminue de 1 à chaque itération, et vaut 0 à la fin du programme. Donc il y a eu n_0 itérations.
3. Pour tout $i \in [\![0, n_0]\!]$, notons r_i le contenu de `r` en entrée de l'itération i et en sortie de l'itération $i-1$, et posons $P(i)$:« $r_i = 0$ » .
   - Initialement, $r_0 = 0$, d'où $P(0)$.
   - Soit $i \in [\![0, n_0 - 1]\!]$, supposons $P(i)$. Donc $r_i = 0$. Vu les instructions exécutées lors de l'itération i, on a $r_{i+1} = 2r_i + 3r_i^2 = 0$. D'où $P(i+1)$.

   Ainsi, $\forall i \in [\![0, n_0]\!]$, $P(i)$. En particulier, à la fin du programme `r` contient 0. Donc cette fonction renvoie 0.

4

```python
def plusPetitDiviseur(n):
    """ Renvoie le plus petit diviseur de n supérieur à 2."""
    r=2
    while n%r != 0:
        # invariant de boucle : les éléments de [2,r[ ne divisent pas n
        r+=1
        # ici aussi

    # sortie de boucle:
    #    - par l'invariant, les éléments de [2,r[ ne divisent pas n
    #    - par la condition du tant que, r divise n
    # donc r est bien le plus petit diviseur de n.
    return r
```

```python
def decomp(n):
    facteurs=[]
    quotient =n
    while quotient != 1:
        # invariant de boucle : n = quotient * produit des éléments de facteurs
        #                       et facteur ne contient que des nombres premiers.
        d= plusPetitDiviseur(quotient)
        quotient/=d
        facteurs.append(d)

    # sortie de boucle:
    #      -  n = produit des éléments de facteurs, par la condition du while et l'invariant
    #      - facteurs ne contient que des nombres premiers, par l'invariant
    return facteurs
```

- **Terminaison :** La suite des valeurs successives de `quotient` est entière, positive, et strictement décroissante (car à chaque étape, d est un diviseur de `quotient` supérieur ou égal à 2).
  Correction : Notons n_0 le nombre d'itération, et pour tout $i \in [\![0, n_0]\!]$, q_i, F_i, d_i le contenu de `quotient`, `facteur` et `d` en entrée d'itération i, et posons $P(i)$: « F_i ne contient que des nombres premiers et $n = q_i \times \prod_{k \in F_i} k$ » .
  - ◇ Initialement, $q_0 = n$ et $F_i = []$ d'où $P(0)$.
  - ◇ Soit $i \in [\![0, n_0 - 1]\!]$ tel que $P(i)$. On a $q_i = d_{i+1} \times q_{i+1}$ et $F_{i+1} = F_i + [d_{i+1}]$ et d_{i+1} est un nombre premier.
    Le fait que F_{i+1} ne contient que des nombres premiers vient de $P(i)$ et du fait que d_{i+1} est premier car c'est le plus petit diviseur de q_i. (Détail : si d_{i+1} avait un diviseur non trivial k, k diviserait q_i tout en étant strictement inférieur à d_{i+1} : absurde).
    Ensuite, on a d'après $P(i)$, $n = q_i \times \prod_{k \in F_i} k$, d'où :

$$\begin{aligned} n =& q_{i+1} \times d_{i+1} \times \prod_{k \in F_i} k \\ =& q_{i+1} \times \prod_{k \in F_{i+1}} k. \end{aligned}$$

    d'où $P(i+1)$.

  Ainsi, P est bien un invariant de boucle.
  En sortie de boucle, on a donc $n = 1 \times \prod_{k \in F_{n_0}} k = \prod_{k \in F_{n_0}} k$ et F_{n_0} ne contient que des nombres premiers. Ainsi, F_{n_0} est bien une décomposition de n en produit de nombres premiers, et c'est le contenu de `facteurs` à la fin du programme donc la valeur renvoyée.

1 Pour tout $n \in \mathbb{N}$ posons $P(n)$: « $f(n)$ termine et renvoie 0. ».

- *Initialisation :* Vu le cas de base, $f(0) = 0$.
- Soit $n \in \mathbb{N}^*$, supposons $P(n-1)$. Exécutons $f(n)$. Par hyothèse de récurrence $f(n-1)$ termine et renvoie 0. Alors la boucle consiste à sommer n fois 0. Donc `res` contient 0 à son issue. Et $f(n)$ termine et renvoie 0.

2a C_0 et pour tout $n \in \mathbb{N}^*$, $C_n = n + n \times C_{n-1}$. Le $n+$ vient des n additions dans la boucle, et le $n \times C_{n-1}$ vient des n appels récursifs à $f(n-1)$ (ce qui est parfaitement idiot au passage : il aurait fallu calculer $f(n-1)$ une seule fois et enregistrer le résultat.).

2b Récurrence évidente. Attention à l'initialiser à 1 car pour $n = 0$ la formule est fausse.

2c

```
def f_mieux(n):
    return 0
```