Preuves d'algorithmes

C. Charignon

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 voit 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 arriver à i 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.

Dit autrement, si u est une suite d'entier naturels, strictement décroissante, alors elle n'est pas définie sur un ensemble infini, autrement dit *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 predn ici comme variant de boucle la quantité `fin-deb`. En effet, cette quntté est entière (différence de deux entiers), positive (par la condition même de la boucle) et strictement décroissamte car elle dimnue de 1 à chaque itération.

-
```python
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.
2. Il manque quelque chose pour que le résultat concorde avec la description. Voyez-vous quoi ?

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

Remarque : En Python le reste de la division euclidienne de x par y s'obtient ainsi : `x%y`.

Ici, nous prenons comme suite 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 Principe de base

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 ».

2.3 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. Ainsi, pour une boucle de type `for i in range(0,n)`, on écrira « l'itération i » pour dire « lorsque le compteur `i` prend la valeur i »[2].

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 $i-1$». Par contre, 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».

2.3.1 Premier exemple : factorielle

Reprenons la fonction suivante :

```python
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 :

```python
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 $(i-1)!$ ».

- **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 $P(i+1)$.

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

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

2.3.2 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
```

2.3.3 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 notation pourra être utile de temps en temps dans les programmes, mais surtout sera une notation très pratique sur papier 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.

2.3.4 Exemple avec un tableau : `estTrié`

2.4 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.

Maintenant, commençons par quelques exemples.

2.4.1 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, \infini[|.
    """
    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 :

```
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[.
```

```python
    """
    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.
    return res
```

On choisir donc l'invariant de boucle suivant : « À la fin de chaque tour de boucle, les éléments $[\![2, res[\![$ ne divisent pas n ».

Démontrons-le par récurrence. J'adopte les notations complètes suivantes :

- 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. Il est convenu que r_0 est le contenu de `res` avant de commencer la boucle. En outre, si par hasard $N = \infty$, 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 » .
- **Initialisation :** Avant de rentrer dans la boucle, on a `res`=2. Donc $r_0 = 2$, et $[\![2, r_0[\![= \varnothing$, et la phrase « les éléments $[\![2, r_0[\![$ ne divisent pas n » est vrai.
- **Terminaison :**

  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 $(n - r_i)_i$.

  - 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 $\mathtt{n} - \mathtt{r_i} \geqslant 0$.

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

- **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 $P(i+1)$.

Revenons à la preuve de la correction de notre algorithme. À la fin du programme, nous avons les informations suivantes :

- $[\![2, \mathtt{res}[\![$ ne contient pas de diviseur de n. Nous le savons grâce à l'invariant de boucle.
- $\mathtt{res}|n$, ceci grâce au fait que la boucle s'est finie, donc sa condition n'est plus vraie.

Ces deux points prouvent bien que `res` est le plus petit élément de $[\![2, \mathtt{res}[\![$ qui divise n. □

2.4.2 Version améliorée de la recherche dans un tableau

cf exercice: ??

```python
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 $x \in t[0:i]$ ».

   (b) *Preuve par récurrence de l'invariant de boucle.*

   (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.

2.4.3 Division euclidienne

cf exercice: 2

2.4.4 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. *Soit $(a, b, q, r) \in \mathbb{Z}^4$ tels que $a = b.q + r$. 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 ← a
    y ← b
    tant que b ≠ 0 :
        x ← y
        x ← 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. 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 $y \leqslant 0$ (la condition du "tant que" n'étant plus vérifiée), mais aussi $y \geqslant 0$ car un reste de division euclidienne est toujours $\geqslant 0$. 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.

2.4.5 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 deux choses :
      - L'invariant de boucle ;
      - 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

L'algorithme que nous allons maintenant étudier est un peu plus complexe, et figure au programme officiel. Il a une importance cruciale dès qu'on a besoin de gérer une grande quantité de données.

3.1 Présentation de l'algorithme

Nous avons déjà vu comment chercher si un tableau contient un certain élément. Supposons maintenant que ce tableau est un tableau de nombres (plus généralement, d'un type muni d'une relation d'ordre total...) et que ces nombres sont triés dans l'ordre croissant. On peut alors employer un algorithme bien plus efficace.

3.2 Terminaison

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.

Entrées : un tableau T d'entiers trié dans l'ordre croissant et un entier x
Sorties : un booléen indiquant si $x \in T$
Variables locales :

- d et f qui indiquent la plage de recherche actuelle. Précisément, on recherchera dans $T[d:f]$.
- m milieu de d et f
- trouvé qui indique si on a trouvé x

```
début
    d ← 0
    f ← longueur de T
    trouvé ← Faux
    tant que non trouvé et d < f :
        #On cherche dans T[d : f]
        m ← (d + f)//2
        si T[m] = x :
            trouvé ← vrai
        sinon:
            si T[m] < x :
                d ← m + 1
            sinon:
                f ← m
            fin
        fin
    fin
fin
```

Algorithme 3 : chercheDicho

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 :

$$\texttt{trouvé} \text{ est vrai ssi } x \in \texttt{T} \setminus \texttt{T}[\texttt{d} : \texttt{f}]$$

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

Passons à la rédaction formelle. J'utilise ci-dessous la notation un peu lourde mais rigoureuse consistant à noter, pour tout i et toute variable `x`, x_i le contenu de `x` au début de l'itération i.

Soit `T` un tableau, n sa longueur, et x un élément. On effectue `chercheDicho(T,x)`.
Soit n_0 le nombre d'itérations de la boucle. Notons pour tout $i \in [\![0, n_0]\!]$, ti, d_i, f_i et m_i le contenu des variables `trouvé, d, f` et `m` après i itérations. Convenons que d_0 et f_0 sont le contenue de `d` et `f` avant de commencer la boucle.
On pose alors pour tout $i \in [\![0, n_0]\!]$, $P(i)$:« t_i est vrai ssi $x \in \setminus T[d_i : f_i]$. ».

- **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 $T[0], \ldots T[m_{i+1}]$ 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` *est vrai :*
  D'après $P(n_0)$ c'est que $x \in T$.
- *cas 2, si* `trouvé` *est faux :*
  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.

4 Bonus : exemple d'étude complète d'un algorithme : tri bulle

Soit T un tableau de nombres. Le but est de trier les éléments de T dans l'ordre croissant.

C'est un exemple classique d'algorithme un peu plus difficile que ceux vus jusqu'à présent (on va utiliser une boucle "tant que "**et** une boucle "pour"). Vous étudierez en seconde année des algorithme tris plus efficaces, mais pour cette année nous utiliserons une méthode naïve.

4.1 Objets modifiables et effets de bord

On profite de l'occasion pour revenir sur la notion d'objet modifiable. On rappelle qu'en Python, les tableaux (c'est-à-dire le type "list") est la seule structure de donnée au programme qui soit modifiable ("mutable" en anglais).

Ainsi, les opérations "append()", "pop()" et "T[i]=..." modifient directement le tableau sur le disque dur. Au contraire une opération sur un type non mutable, par exemple `m=m+"abc"` si m est une chaîne de caractères, va créer un nouvel objet, à un nouvel emplacement du disque dur, qui sera encore désigné par la lettre `m`.

Mais de plus, lors du passage d'un objet modifiable en argument d'une fonction, la valeur reçue en argument pointera encore sur la même zone mémoire. Exemple :

```
T=[1,2,3,4]

def f(X):
    X.append(33)

f(T)
```

Lorsqu'on appelle f avec comme argument T, le X utilisé dans le corps de la fonction f pointera vers la même zone mémoire que T. (Pour une variable non modifiable, le X aurait été une **copie** de T.) Concrètement, cela signifie que toute modification du X au sein de la fonction va aussi modifier le T à l'extérieur.

Ainsi, à l'issue du code ci-dessus, T contient $[1, 2, 3, 4, 33]$.

La fonction f ne renvoie rien (pas de "return"), mais elle a quand même un effet : elle rajoute 33 dans le tableau donné en entrée. On dit que f crée des "effets de bord".

4.2 Échange de deux cases du tableau

On propose ici d'écrire un algorithme qui trie le tableau T "en place", c'est-à-dire qu'on ne va pas renvoyer un nouveau tableau contenant les éléments de T triés, mais qu'on va directement modifier T. On va fonctionner par "effets de bord".

Pour commencer, écrivons un algorithme qui échange deux cases dans T :

Entrées : i, j entiers, T un tableau modifiable
Sorties : rien ! échange les cases i et j du tableau par effet de bord.
Variables locales : temp

début
  $temp \leftarrow T[i]$
  $T[i] \leftarrow T[j]$
  $T[j] \leftarrow temp$
fin

Algorithme 4 : échange de cases

4.3 Le tri bulles

Le principe de ce tri est simple : on parcourt le tableau T, et à chaque fois que deux cases consécutives ne sont pas dans le bon ordre, on les échange. On continue jusqu'à ce que le tableau soit trié.

Le parcourt de T ressemblera à ceci (attention à la valeur finale de la boucle !) :

Et on continue "tant que le tableau n'est pas trié". Pour savoir quand s'arrêter, nous utiliserons un booléen `trié`[3].

Pour savoir si le tableau est trié, c'est simple : si dans la boucle "pour" ci-dessus on n'a eu aucun inversion à faire, c'est que le tableau était trié.

début
  pour *i de 0 à* $n-2$ **:**
    si *T[i]>T[i+1]* **:**
      échange(T,i,i+1)
    fin
  fin
fin

Entrées : un tableau T modifiable
Sorties : rien ! T sera trié par effet de bord.
Variables locales : trié : booléen

début
  trié ← Faux
  tant que *non trié* **:**
    trié ← Vrai
    pour *i de 0 à* $n-2$ **:**
      si *T[i]>T[i+1]* **:**
        échange(T,i,i+1)
        trié ← Faux
      fin
    fin
  fin
fin

Algorithme 5 : tri bulle

On arrive à l'algorithme :

Ci-dessous on donne une traduction Python. On a rajouté quelques lignes d'affichages permettant d'afficher T à chaque étape pour suivre le fonctionnement du tri. Le `debug=False` en argument signifie que l'utilisateur peut rentrer un paramètre qui s'appellera "debug", mais que s'il ne rentre rien, la valeur par défaut sera "False".

Ainsi, par défaut le programme n'affiche rien, mais pour affichera le déroulement du tri, il faudra taper `triBulles(T, debug=Tru`

```
def echange(T,i,j):
    """ échange les éléments i et j du tableau T."""
    tmp=T[i]
    T[i]=T[j]
    T[j]=tmp

def triBulles(T, debug=False):
    """tri le tableau T en place par la méthode des bulles."""
    trie=False
    while not trie:
        trie=True
        for i in range(len(T)-1):
            if debug:
                os.system("clear") #effacer l'écran. Remplacer "clear" par "cls" sous windows.
                print(T)
                time.sleep(0.1) #attendre 0.1s
            if T[i]>T[i+1]:
                trie=False
                echange(T,i,i+1)
```

4.4 Terminaison et correction

Dans cet exemple, on justifie de manière élémentaire que l'algorithme s'arrête et fonctionne correctement.

3. Au fait, un booléen de ce genre qui indique quand la boucle va s'arrêter s'appelle traditionnellement un "drapeau".

Notons n la longueur de T.

À l'issue du premier parcourt du tableau, le maximum de T est arrivé en dernière position, c'est-à-dire dans sa position finale. En effet, soit M ce maximum et i_0 son emplacement initiale (si M apparaît plusieurs fois dans le tableau, prenons pour i_0 le plus grand indice tel que $M = T[i_0]$).

Dans la boucle, lorsque i vaudra i_0, la condition $T[i] > T[i+1]$ sera remplie, donc $T[i]$ et $T[i+1]$ seront échangés. Donc M arrivera à la position $i_0 + 1$.

Mais à l'étape suivante de la boucle "pour", i vaudra $i_0 + 1$. Donc encore une fois l'échange $T[i] \leftrightarrow T[i+1]$ aura lieu, et M arrivera en case d'indice $i_0 + 2$.

Ainsi de suite, on constate que M va ainsi remonter de case en case jusqu'à la dernière case.

Lors du second parcourt du tableau, c'est le deuxième plus grand élément qui va arriver à l'avant dernière case.

Ainsi de suite, pour tout $k \in [\![1, n]\!]$, après k parcours du tableau, les k plus grands éléments de T seront arrivés à leur place.

En particulier, après n parcours du tableau, le tableau est entièrement trié, et l'algorithme s'arrête.

4.5 Amélioration

On a vu qu'après k parcours du tableau, les k plus grands éléments du tableau sont déjà en place. Il est donc inutile de pousser la boucle "pour" jusqu'au bout : il suffit d'aller jusqu'à la case $n - k$.

Entrées : un tableau T modifiable
Sorties : rien ! T sera trié par effet de bord.
Variables locales : trié : booléen, k entier

```
début
    trié ← Faux
    k ← 2
    tant que non trié :
        trié ← Vrai
        pour i de 0 à n − k :
            si T[i]>T[i+1] :
                échange(T,i,j)
                trié ← Faux
            fin
        fin
        k ← k + 1
    fin
fin
```

Algorithme 6 : tri bulle optimisé

4.6 Complexité

Les opérations élémentaires utilisées dans cet algorithme sont les comparaisons et les affectations. Nous choisissons de compter les comparaisons pour les raisons suivantes :

- il y en a plus que des affectations ou alors du même ordre de grandeur.
- leur nombre est plus facile à compter, puisque dans la boucle "pour" les comparaisons sont toujours effectuées, au contraire des affectations.

Cependant, on pourrait aussi compter les affectations dans le pire des cas.

Chaque passage de la boucle "pour" effectue $n - k + 1$ comparaison. Et cette boucle "pour" va s'effectuer au pire pour k variant de 2 à n. Le nombre total de comparaisons est donc au pire des cas :

$$\begin{aligned} \sum_{k=2}^{n} n - k + 1 &= \sum_{j=1}^{n-1} j \qquad \text{(penser } j = n - k + 1\text{)} \\ &= \frac{n(n-1)}{2} \end{aligned}$$

$$\underset{n\to\infty}{\sim} \frac{n^2}{2}$$
$$= O_{n\to\infty}(n^2)$$

4.7 Tri d'un tableau à deux dimensions selon différentes colonnes

On propose maintenant une variante permettant de travailler un peu plus la structure de tableau modifiable.

Le tableau `enfants` contient les données suivantes concernant les enfants à qui le père noël va donner des cadeaux : le nom des enfants, la distance où ils habitent, s'il ont été sage ou pas, et leur âge.

Nous allons écrire une fonction `triSelon(T,c)` qui va trier le tableau T selon la $c$ème colonne. Ainsi l'appel `triSelon(enfants, 2)` triera le tableau `enfants` en mettant les plus sages en premiers. Mais `triSelon(enfants, 1)` mettra ceux qui habitent le plus près en premier [4].

On commence par écrire une fonction `échangeLignes` pour remplacer la fonction `échange` précédente :

Entrées : T tableau de tableaux, i, j entiers
Sorties : rien. Échange les lignes i et j dans T par effet de bord.
Variables locales : temp : tableau

début
  temp $\leftarrow T[i]$
  pour *k de 0 à (longueur de T)-1* **:**
    T[i][k]$\leftarrow$ T[j][k]
    T[j][k]$\leftarrow$ T[i][k]
  fin
fin

Algorithme 7 : échangeLignes

Dès lors, il nous suffit de quelques modifications :

Entrées : un tableau T modifiable, c l'indice d'une colonne
Sorties : rien ! T sera trié par effet de bord selon la $c$ème colonne.
Variables locales : trié : booléen, k entier

début
  trié $\leftarrow$ Faux
  $k \leftarrow 2$
  tant que *non trié* **:**
    trié $\leftarrow$ Vrai
    pour *i de 0 à $n-k$* **:**
      si *T[i][c]>T[i+1][c]* **:**
        échangeLignes(T,i,j)
        trié $\leftarrow$ Faux
      fin
    fin
    $k \leftarrow k+1$
  fin
fin

Algorithme 8 : tri bulle pour un tableau à deux dimension

Après quelques essais on constate... que ça ne marche pas. En ciblant un peu plus les essais, c'est l'échange de lignes qui est en cause : au lieu d'échanger les lignes i et j elle copie la ligne j dans la ligne i.

Explication ci-dessous :

4.8 Autre conséquence du caractère modifiable d'un tableau

C'est du au fait que $T[i]$ et $T[j]$ sont eux-mêmes des tableaux, et donc des objets modifiables. Et la commande `temp=T[i]` ne copie pas vraiment $T[i]$ dans *temp*, mais se contente de créer une variable `temp` qui pointe sur le même

4. à propos, la problème de déterminer le plus court trajet pour passer par la cheminée de chaque enfant est un problème connu très difficile...

tableau que $T[i]$ (on dit que `temp` est un simple « alias » pour $T[i]$). Dès lors, toute modification de $T[i]$ modifie en même temps *temp*.

$$\begin{tikzcd} temp \arrow[dr] & \\ T[i] \arrow[r] & {[\ldots]} \\ T[j] \arrow[r] & {[\ldots]} \end{tikzcd}$$

C'est bien entendu pour des question de vitesse, mais aussi pour permettre la programmation par effet de bord que Python choisit d'effectuer des copies superficielles pour le type "list".

Pour effectuer une vraie copie (c'est-à-dire une copie "profonde", "deep copy" en anglais), le plus simple est d'utiliser la commande `deepcopy` de la bibliothèque `copy`.

Le code suivant corrige la fonction `echangeLigne` :

```
import copy
def echangeLigne(T):
    temp=copy.deepcopy(T[i])
    for k in range(len(T[i])):
        T[i][k]=T[j][k]
        T[j][k]=temp[k]
```

En résumé, voici les trois conséquences concrètes principales du fait qu'une structure de données soit modifiable :

- Il existe des commandes permettant de modifier en place l'objet (`append(...)`, `pop()`, `T[i]=...` pour le type "list" de Python). Ces commandes seront plus rapide que des opérations consistant à recréer un nouvel objet (`T.append(x)` sera plus rapide que `T=T+[x]`).
- Action par effets de bords possible : une fonction appelée avec T comme argument peut modifier le contenu de T.
- Les copies sont superficielles. Après une commande `T2=T1`, $T2$ et $T1$ pointent vers le **même** tableau. Toute modification de l'un modifie l'autre également.

On constate que ces spécificités ont toute pour objectif d'accélérer les opérations. Elles sont implémentées dans les langages pour les types de données appelés à contenir beaucoup de données.

Enfin, dernière remarque : rien n'empêche que dans certains langage de programmation, seuls certains de ces trois points soient vérifiés pour certains types de données. Mais la plupart du temps, ils vont ensemble.

Deuxième partie

Exercices

Exercices : preuves d'algorithmes

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.
6. La fonction `sansZéro` de l'exercice sur le crible d'Ératosthène.

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

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

```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 :* Réécrire la fonction avec des noms de variables plus clairs, rajouter une aide, et indiquer le rôle des variables en commentaire.
4. Quelles sont les conditions sur les arguments pour que cet algorithme termine ?
5. Dans le cas où ces conditions sont vérifiées, démontrer que l'algorithme termine.
6. 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`.
7. Démontrer que cette fonction renvoie bien le résultat espéré.
8. Combien de fois s'exécute la boucle ?
9. Modifier cet algorithme pour qu'il fonctionne aussi lorsque $b < 0$ ou $a < 0$.

Exercice 3. ** Une fonction inutile

Soit la fonction `f` suivante :

```python
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`.

Quelques indications

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 :

```python
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**

```python
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.

```python
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 :

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

   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 :

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

    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 contenue de `facteurs` à la fin du programme donc la valeur renvoyée.