Preuves d'algorithmes

Une fois qu'on a écrit un programme, il faut vérifier qu'il fait bien ce qu'on attend de lui. On peut effectuer une série de tests bien choisis, mais ces derniers ne peuvent recouvrir tous les cas possibles et il est préférable de démontrer rigoureusement que le programme est correct.

Prouver un algorithme consiste à résoudre les deux problèmes suivants :

– Démontrer sa **terminaison**, i.e. prouver que, quand on l'exécute, il finit par s'arrêter.

– Démontrer sa **correction**, i.e. prouver qu'il renvoie effectivement le résultat attendu.

Terminaison

Le problème de terminaison d'un algorithme se pose dans deux situations : les boucles `while` et les fonctions récursives (cf cours correspondant).

Pour démontrer qu'une boucle `while` se termine, on essaiera généralement de trouver un **variant de boucle**, c'est-à-dire un entier positif qui décroît strictement à chaque itération. Si la boucle était infinie, on pourrait alors construire une suite strictement décroissante d'entiers naturels, ce qui est impossible.

Par exemple, dans le programme suivant, la quantité $10 - n$ est un variant de boucle :

```
n = 0
while n < 10:
    print(n)
    n = n + 1
```

La fonction suivante reçoit deux entiers naturels a et b, avec b non nul, et renvoie le quotient et le reste de la division euclidienne de a par b.

```
def division_euclidienne(a, b):
    r = a
    q = 0
    while r >= b:
        r = r - b
        q = q + 1
    return q, r
```

L'entier naturel r décroît strictement à chaque itération puisque b est strictement positif. Cela prouve la terminaison de l'algorithme.

Démontrer la terminaison d'un algorithme est souvent assez simple, mais pas toujours. Par exemple, on ne sait pas démontrer que la fonction suivante (cf TP 2) termine toujours :

```
def syracuse(n):
    while n > 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3*n + 1
        print(n)
```

Correction

On dit que la correction d'un algorithme est **partielle** s'il renvoie un résultat correct lorsqu'il s'arrête. Elle est **totale** si elle est partielle et que l'algorithme termine.

Pour démontrer qu'une boucle `while` ou `for` fait ce qu'on attend d'elle, on utilisera généralement un **invariant de boucle**, c'est-à-dire une propriété qui est vraie avant la boucle et qui, si elle est vraie avant une itération, reste vraie après celle-ci. On peut alors en déduire qu'elle est vraie à la fin de la boucle (c'est la même idée que le raisonnement par récurrence en mathématiques).

Reprenons par exemple la fonction `division_euclidienne` précédente. On veut démontrer (par définition de la division euclidienne) que le couple (q, r) qu'elle renvoie vérifie

$$\begin{cases} a = bq + r \\ 0 \leqslant r < b \end{cases}.$$

Pour cela, on va montrer que la propriété $a = bq + r$ est un invariant de boucle.

Avant la boucle, on a $q = 0$ et $r = a$, donc on a bien $bq + r = a$.

Supposons que l'on ait $a = bq + r$ au début d'une itération. Soient q' et r' les valeurs de q et r à la fin de l'itération : on a $q' = q + 1$ et $r' = r - b$. Alors on a bien $bq' + r' = b(q + 1) + r - b = bq + r = a$.

La propriété $a = bq + r$ est donc bien un invariant de boucle. Par conséquent, elle est vraie à la fin de la boucle. Mais on a alors $0 \leqslant r < b$. Ainsi les q et r renvoyés sont bien respectivement le quotient et le reste dans la division euclidienne de a par b.

Dans la suite on étudie (terminaison, correction, complexité) deux algorithmes classiques où il n'est pas évident que le résultat final soit correct.

Exponentiation rapide

Soient x un réel et n un entier naturel. Pour calculer x^n on peut simplement écrire :

```
def puissance_naive(x, n):
    p = 1
    for i in range(n):
        p = p * x
    return p
```

La terminaison et la correction de cette fonction sont évidentes. Elle effectue n multiplications et $n + 1$ affectations donc sa complexité est $O(n)$ dans tous les cas (en fait sa complexité réelle est difficile à mesurer car p peut prendre des valeurs très grandes, faisant intervenir ainsi des entiers multi-précision qui sont plus longs à manipuler).

Testons-la avec une grande valeur de n :

```
from time import time
t = time()
puissance_naive(2, 1000000)
print(time()-t)
```

Le programme affiche `34.667269468307495`.

Considérons maintenant la fonction suivante :

```
def puissance_rapide(x, n):
    p = 1
    while n > 0:
        if n % 2 == 1:
            n = n - 1
            p = p * x
        else:
            n = n // 2
            x = x * x
    return p
```

En la testant avec différentes valeurs de x et de n, on peut penser qu'elle renvoie toujours bien x^n.

Comparons sa rapidité avec la fonction `puissance_naive` pour une grande valeur de n :

```
from time import time
t = time()
puissance_rapide(2, 1000000)
print(time()-t)
```

Le programme affiche `0.016097068786621094`. La fonction est manifestement très rapide, mais il n'est pas évident qu'elle renvoie bien x^n. Démontrons-le.

Terminaison : Elle vient du fait que l'entier naturel n décroît strictement à chaque itération.

Correction : On va montrer que la propriété $px^n = X^N$, où X et N sont les valeurs initiales de x et n, est un invariant de boucle.

Elle est évidemment vraie avant la boucle puisque $p = 1$ au départ.

Supposons qu'elle est vraie au début d'une itération. Notons x', n' et p' les valeurs de x, n et p à la fin de l'itération.

Si n est pair, alors $x' = x^2$, $n' = n/2$ et $p' = p$ donc $p'x'^{n'} = p(x^2)^{n/2} = px^n = X^N$. Si n est impair, alors $x' = x$, $n' = n - 1$ et $p' = px$ donc $p'x'^{n'} = pxx^{n-1} = px^n = X^N$.

La propriété $px^n = X^N$ est donc bien un invariant de boucle. Par conséquent, elle est vraie à la fin de la boucle. Mais on a alors $n = 0$, donc $px^n = p$, d'où $p = X^N$: la fonction renvoie bien le résultat attendu.

Complexité : À chaque itération on fait le même nombre d'opérations élémentaires. Au moins une fois sur deux, n est pair et on le divise par 2. Ainsi après $2p$ itérations n est divisé au moins par 2^p. La boucle s'arrête quand $n < 1$. Or :

$$\frac{n}{2^p} < 1 \Leftrightarrow n < 2^p \Leftrightarrow \log_2(n) < p,$$

où $\log_2(n) = \frac{\ln n}{\ln 2}$. Par conséquent on fait au plus $2\log_2(n)$ itérations : l'algorithme est en $O(\ln n)$ (en réalité il faudrait tenir compte des entiers multi-précision).

Recherche dichotomique dans une liste triée

Pour rechercher si un élément appartient ou non à une liste donnée, on peut parcourir cette liste jusqu'à ce qu'on trouve l'élément cherché. Dans le pire des cas (si l'élément n'est pas dans la liste), l'algorithme est en $O(n)$ où n est la longueur de la liste.

Si la liste est triée, il existe une méthode beaucoup plus efficace : la recherche dichotomique. On commence par considérer l'élément situé au milieu de la liste. Si l'élément cherché est plus petit que cet élément, on recommence avec la partie gauche de la liste, et sinon on recommence avec la partie droite de la liste. Et on continue. En Python :

```
def recherche_dichotomique(x, L):
    a, b = 0, len(L) - 1   # a, b : indices entre lesquels on cherche x
    while a <= b:
        m = (a+b) // 2     # milieu de [|a,b|]
        if L[m] == x:
            return True    # on a trouvé x
        if x < L[m]:
            b = m - 1      # on cherche à gauche de m
        else:
            a = m + 1      # on cherche à droite de m
    return False
```

On pourra tester cette fonction avec diverses valeurs de `x` et de `L`.

Terminaison : L'entier naturel `b - a` décroît strictement à chaque itération. La fonction s'arrête soit quand il devient strictement négatif, soit quand on a trouvé `x`.

Correction : Si `x` n'est pas dans la liste, on n'aura jamais `L[m] == x` donc la fonction renverra `False`. Supposons que `x` est dans la liste. On va montrer que `x` est toujours compris entre `L[a]` et `L[b]`. Montrons que cette propriété est un invariant de boucle.

Avant la boucle c'est vrai car `a = 0` et `b = len(L) - 1`.

Supposons que `L[a] <= x <= L[b]` au début d'une itération. Alors :

- Si `L[m] == x` la fonction renvoie `True` et s'arrête.
- Si `x < L[m]` alors `x <= L[m-1]`, donc en remplaçant `b` par `m - 1` on a toujours `L[a] <= x <= L[b]`.
- Si `L[m] < x` alors `L[m+1] <= x`, donc en remplaçant `a` par `m + 1` on a toujours `L[a] <= x <= L[b]`.

La propriété `L[a] <= x <= L[b]` est donc bien un invariant de boucle. Au pire la boucle continue jusqu'à ce qu'on ait `a = b`, mais alors on a `L[a] <= x <= L[a]` donc `x = L[a]` et `m = a` donc la fonction renvoie bien `True`.

Complexité : On considère uniquement le nombre de comparaisons. Dans le meilleur des cas, `x` est situé au milieu de la liste et on le trouve à la première itération. On ne fait donc qu'une comparaison. Dans le pire des cas, `x` n'est pas dans la liste. Comme à chaque itération la taille de l'intervalle `[|a, b|]` est au moins divisée par 2, alors à la p^e itération elle est inférieure à $\frac{n}{2^p}$. Or $\frac{n}{2^p} < 1 \Leftrightarrow p > \log_2(n)$, donc l'algorithme est en $O(\ln n)$.