Correction des algorithmes

Spé NSI - Lycée du parc

Année 2020-2021

Introduction

On poursuit ici l'étude théorique des algorithmes entreprise dans le chapitre traitant de la complexité.

On se pose maintenant la question de savoir si un algorithme donné répond bien au problème qu'il est censé traité dans sa *spécification*. Il se pose alors deux grandes questions :

- Se termine-t-il ? C'est la question de la *terminaison*.
- Résout-il bien le problème qu'il est censé traiter ? C'est la question de la *correction*[1].

Le but de ce chapitre est d'introduire les méthodologies qui permettent de traiter ces problèmes.

I Terminaison

Pour un algorithme ou une partie d'algorithme qui ne comporte pas de boucles ou seulement des boucles inconditionnelles, la question de la terminaison ne se pose, a priori, pas. Le cas des boucles conditionnelles est plus délicat : la condition est censée être vraie au départ (sinon c'est du code mort) et cette même condition doit finir par être fausse sinon les itérations ont lieu indéfiniment.

Exercice 1

Laquelle de ces deux boucles ne se termine pas ?

```
1 x = 0
2 while x >= 0:
3     x += 1
```

1

```
1 x = 10
2 while x >= 0:
3     x = x - 1
```

2

---

1. On parle parfois de correction partielle plutôt que de correction. La correction totale étant la conjonction de la terminaison et de la correction partielle.

Vocabulaire : On appelle **itération** d'une boucle **une** exécution des instructions qui figure dans le corps de la boucle.

Démonstration de la terminaison

Pour démontrer qu'une boucle conditionnelle (**while**) se termine, il suffit de déterminer une quantité exprimée à l'aide des variables de l'algorithme qui **reste positive et entière** tout au long de l'exécution de la boucle et **décroit strictement** à chaque itération. Comme il n'existe pas de suite infinie à valeurs dans $\mathbb{N}$ qui soit strictement décroissante cela prouve alors que le nombre d'itérations est fini.

On appelle souvent **variant** de la boucle une telle quantité [2].

Exercice 2 Montrer que l'algorithme de division euclidienne dans $\mathbb{N}$ se termine (avec $b > 0$) :

```
1 def division_euclidienne(a, b):
2     q = 0
3     r = a
4     while r >= b :
5         r = r - b
6         q = q + 1
7     return (q, r)
```

Exercice 3 Montrer que l'algorithme de recherche séquentielle dans une liste se termine :

```
1 def appartient(x, L):
2     n = len(L)
3     i = 0
4
5     while i < n and L[i] != x:
6         i += 1
7
8     return i != n
```

2. Certain auteurs utilisent aussi *convergent* ou *expression de terminaison*.

Exercice 4

Les deux fonctions ci-dessous calculent le produit de deux entiers positifs. Justifier qu'elles terminent à l'aide d'un variant.

```
def prod1(a, b):
    S = 0
    while b > 0:
        S = S + a
        b = b - 1
    return S
```

```
def prod2(a, b):
    S = 0
    while b > 0:
        if b%2 == 1:
            S = S + a
        b = b//2  # quotient / 2
        a = a + a
    return S
```

Entraînement 1

Les deux fonctions suivantes calculent le nombre de chiffres de l'écriture décimal de `n` pour $n > 0$. Justifier qu'elles terminent à l'aide d'un variant.

```
def nbChiffres1(n):
    k = 0
    while n > 0:
        n = n // 10
        k = k + 1
    return k
```

```
def nbChiffres2(n):
    p = 1
    k = 0
    while p < n + 1:
        p = p*10
        k = k + 1
    return k
```

II Correction

Pour montrer la correction d'un algorithme, les difficultés se posent dans les boucles (quel qu'en soit le type, conditionnelles ou inconditionnelles). On utilise un **invariant de boucle** c'est une propriété qui reste vraie tout au long des itérations et qui, à la fin, donne le résultat attendu.

Exercice 5 Montrer que la fonction `puissance` calcule bien x^n.

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

Exercice 6 Montrer que l'algorithme de division euclidienne dans $\mathbb{N}$ est correct (avec $b > 0$).

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

Exercice 7

Montrer que l'algorithme de recherche du maximum dans une liste est correct :

```
def maximum(L):
    n = len(L)
    maxi = L[0]
    for x in L[1:]:
        if x > maxi:
            maxi = x
    return maxi
```

Exercice 8

Montrer que l'algorithme de recherche séquentielle dans une liste est correct :

```
def appartient(L, x):
    n = len(L)
    i = 0
    while i < n and x != L[i]:
        i += 1
    return i < n
```

Exercice 9 Les deux fonctions ci-dessous calculent le produit de deux entiers positifs. Justifier leur correction à l'aide d'un invariant.

```
def prod1(a, b):
    S = 0
    while b > 0:
        S = S + a
        b = b - 1
    return S
```

```
def prod2(a, b):
    S = 0
    while b > 0:
        if b%2 == 1:
            S = S + a
        b = b//2  # quotient / 2
        a = a + a
    return S
```

Entraînement 2

Écrire une fonction qui calcule la somme des éléments d'un tableau et montrer sa correction avec un invariant.

Entraînement 3

On considère le tableau ci-contre où seul l'élément à l'intersection de la première ligne et de la troisième colonne est un signe « − », les autres éléments étant des signes « + ». On effectue des transformations en changeant tous les signes d'une ligne, d'une colonne ou d'une diagonale quelconque.
Est-il possible qu'après un certain nombre de telles opérations on se retrouve avec un tableau où tous les éléments sont des « + » ?

+	+	−	+
+	+	+	+
+	+	+	+
+	+	+	+

III Retour sur la recherche dichotomique

On rappelle l'algorithme de recherche dichotomique dans une liste triée :

```
def rech_dicho(L, x):
    """ La liste L est supposée triée.
    Renvoie un indice d'un élément de L de valeur x
    si x est présent dans L et None sinon """

    g, d = 0, len(L) -1

    while g <= d:
        m = (g + d) // 2
        if L[m] < x:
            g = m + 1
        elif L[m] > x:
            d = m - 1
        else:
            return m

    return None
```

3

Exercice 10

Faire fonctionner l'algorithme à la main pour :

1. `L=[1, 3, 5, 7, 10]` et `x` =7.
2. `L=[1, 3, 5, 7, 10, 17]` et `x` =2.

Exercice 11 Montrer la terminaison de l'algorithme de recherche dichotomique.

Exercice 12 Montrer la correction de l'algorithme de recherche dichotomique.

IV Les tris

Exercice 13

On rappelle le code du tri par selection[3].

Justifier sa terminaison et démontrer sa correction.

```
def tri_selection2(t):
    for i in range(len(t)):
        j_record = i
        for j in range(i + 1, len(t)):
            if t[j] < t[j_record]:
                j_record = j
        permute(t, i, j_record)
```

3. On dispose d'une fonction **permute(t, i, j)** qui permute les éléments d'indices i et j du tableau t.

Exercice 14

On rappelle le code du tri par insertion.

Justifier sa terminaison et démontrer sa correction.

```
def insere(t, i):
    """ t est un tableau et i est un indice de t. On suppose
    que t est trié jusqu'à la position i-1. La fonction positionne
    l'élément d'indice i à sa place (effet de bord) """

    for j in range(i, 0, -1):
        if t[j - 1] > t[j]:
            permute(t, j, j-1)
        else:
            return

def tri_insertion_en_place(t):
    for i in range(1, len(t)):
        insere(t, i)
```