Terminaison

Définitions

Définition

On dit qu'un algorithme termine si, pour toute entrée, il n'exécute qu'un nombre fini d'étapes avant de renvoyer un résultat.

Un programme sans appel récursif ni boucle conditionnelle termine. Dans le cas d'une boucle conditionnelle, il faut s'assurer de l'éventualité de la condition d'arrêt. Dans celui d'un programme récursif, qu'on arrive en un nombre fini d'étapes à un cas dit "de base".
Exemple : calcul d'une valeur maximale dans un tableau.

Définition

Un **variant de boucle** est une quantité entière positive dépendant des données en entrée de l'algorithme, et, de manière strictement décroissante, du nombre de passages déjà effectués dans la boucle.

Toute boucle admettant un variant n'est empruntée qu'un nombre fini de fois. un algorithme termine pour toute entrée pour laquelle chaque boucle de l'algorithme possède un variant. On peut procéder de même dans le cas d'appels récursifs.

Algorithme 1 : PGCD : Calcul du PGCD de a et b

Input : a,$b > 0$
Output : PGCD Le PGCD de a et b
Data : Paramètre x
$x \leftarrow a$
$y \leftarrow b$
while $y \neq 0$ **do**
  $z \leftarrow y$
  $y \leftarrow x[y]$
  $x \leftarrow z$
end
return max

Exercice : Prouver la terminaison et la correction de la fonction PGCD
Exercice (infaisable) : Prouver la terminaison et la correction de la fonction Syracuse

Correction d'un algorithme et invariant de boucle

Définition

Un algorithme qui, lorsqu'il s'arrête, fournit le résultat attendu est dit **partiellement correct**.
Un algorithme qui termine et fournit toujours le résultat attendu est dit **totalement correct**.

On peut vérifier la correction d'un algorithme en utilisant des **invariants de boucle**.

Définition

On appelle **invariant de boucle** toute assertion vérifiant les conditions suivantes :

- **Initialisation** L'assertion est vraie avant la première itération de la boucle
- **Conservation** si l'assertion est vraie avant une itération de la boucle, elle le reste avant l'itération suivante
- **Terminaison** lorsque la boucle est terminée, l'invariant fournit une propriété qui permet de prouver que l'algorithme renvoie la valeur attendue

En pratique, la recherche d'un invariant peut également servir à comprendre le fonctionnement d'un algorithme.

Exemple :

```
def mult(x,y):
    prod = 0
    for i in range(y):
        prod += x
    return prod
```

On souhaite montrer que l'algorithme renvoie bien le produit des entiers x et y. A l'entrée dans la boucle, la variable `prod` contient 0. On peut faire l'hypothèse qu'à l'entrée de la boucle k, elle contient $(k-1) \times x$: c'est l'invariant de boucle.

- à l'entrée de la boucle : le prédicat est vrai
- lorsqu'il est vrai à l'entrée de la boucle k, il l'est à l'entrée de la boucle suivante.
- lorsqu'on sort de la boucle, la variable contient $k \times x$, ce qui permet de conclure : l'algorithme effectue bien le calcul annoncé

Exemple 2 : Algorithme d'Euclide : l'invariant est l'égalité entre le PGCD des deux données en entrée et en sortie d'une itération de la boucle. En sortie de la boucle, la valeur de x est bien le PGCD recherché.

Tri par insertion :

Exercice :

1. Écrire une fonction qui échange le contenu de deux cases aux indices i et j d'un tableau "tab" ;
2. Écrire une fonction qui, dans un tableau "tab", cherche l'indice de la valeur minimale entre deux indices "i_min" et "i_max" ;
3. Écrire une fonction qui utilise les deux précédentes pour trier un tableau "tab" **en place** ;
4. Prouver la correction de l'algorithme obtenu.

Exercices sur machine

1. Créer une boucle while où l'invariant de boucle est : "à la k-ème boucle, x vaut $2k+1$"
2. Créer une fonction récursive `myst`, telle que l'appel de `myst` sur n a, dans son corps, un appel à `myst` pour $n+1$. Quel variant utiliser ?

Révisions

Travaux pratiques

Terminaison, correction

1. Donner une spécification puis prouver la terminaison et la correction de l'algorithme suivant :

```
def myst(n):
    bool = False
    i=0
    while (i<n or bool):
        bool=not(bool)
        i+=1
    return i
```

2. On considère des villes notées par un entier de 0 à $n-1$, reliées (parfois) par des routes unidirectionnelles. On représente le réseau par un tableau bidimensionnel `tab` tel que `tab[i][j]` vaut 1 s'il y a une route directe de la ville i vers la ville j, et 0 sinon. Il peut exister une route d'une ville vers elle-même.
   (a) Créer (sur le papier) ce tableau pour un petit entier n (~ 3) et un réseau `non total` (il doit exister au moins deux villes a et b telles que l'on ne peut aller directement de a vers b)
   (b) Écrire une fonction déterminant si deux villes a et b sont reliées par un chemin de longueur 1.
   (c) Écrire une fonction déterminant si toute ville est reliée par un chemin de longueur 1, dans un sens ou l'autre, à chacune des autres villes. *Penser à tester ces fonctions !*
   (d) Observer que tout chemin de longueur n de a vers b se décompose nécessairement en chemin de longueur $n-1$ de a vers une autre ville c, puis un chemin de longueur 1 de c vers b. En déduire l'expression d'un tableau représentant le nombre de chemin d'une ville vers une autre en fonction du tableau précédent.
   (e) Étant donné une configuration donnée sous forme de tableau et une ville a, écrire une fonction renvoyant la liste des villes non accessibles depuis a. En justifier la terminaison et la correction.
3. **Retravailler :** génération aléatoire de nombres, de tableaux avec conditions sur les éléments du tableau. Recherche dans un tableau bidimensionnel.

Correction

1. Terminaison
   Correction
2. On a caché derrière ce problème la notion de graphes et de matrices.
   (a) Par exemple, le graphe correspondant à `L = [[0,1,1],[1,0,1],[0,1,1]]`
   (b) Il faut regarder s'il existe une route de a vers b (`tab[a][b]` vaudrait alors 1) ou de b vers a (c'est `tab[b][a]` qui nous renseigne ici). Dès que l'un est vrai, la réponse est oui : on utilise donc l'opérateur `or`.

```
def sont_relies(tab,a,b):
    return (tab[a][b]==1 or tab[b][a]==1)
```

(c) **Pour chaque ville** `depart`, il faut regarder si **pour chaque ville** `arrivée`, **dans le cas où** a et b sont distinctes, **si** a et b sont reliées par une route (de taille 1). Il ne reste qu'à transcrire cet algorithme, en observant que dès qu'on a trouvé un couple a, b de villes sans routes entre elles-deux, on peut arrêter la recherche : la réponse est `False`. Si on n'a jamais eu besoin d'arrêter la recherche, c'est que tous les couples de villes distinctes sont reliées : la réponse est `True`.

```
def tous_relies(tab):
    n = len(tab)
    for i in range(n):
        for j in range(n):
            if (i!=j):
                if (not sont_relies(tab,a,b)):
                    return False
    return True
```

(d) Soient les villes i et j fixées, et supposons que A compte les chemins de longueur $n-1$ et B compte les chemins de longueur 1. Tout chemin de longueur n passe au bout de $n-1$ pas par une ville.
Or, le nombre de chemin de i à j passant par k au bout de $n-1$ pas vaut "nombre de chemins de taille $n-1$ de i à k multiplié par nombre de chemins de taille 1 de k à j". Et le nombre de chemins de i à j de longueur n est la somme de tous ces chemins de ville intermédiaire k : $\sum_{k=0}^{m} A[i][k] \times B[k][j]$. On calcule le tableau C, dont la composante en ligne i et colonne j vaut ce nombre de chemins. La fonction qui construit C à partir de A et de B s'appelle ici `prod`, elle comporte deux boucles imbriquées sur les lignes et les colonnes (pour observer chaque terme), et une boucle imbriquée à l'intérieur de ces boucles pour calculer la somme.

```
def prodMat(A,B):
    n = len(A)
    C = [[0 for j in range(n)] for i in range(n)] # initialisation du tableau $C$
    for i in range(n):
        for j in range(n):
            for k in range(n):
                C[i][j]+=A[i][k]*B[k][j]
    return C
```

Le nombre de chemins de a vers b de taille n est accessible dans B si $n = 1$. Sinon, il faut s'intéresser au tableau obtenu à partir de A et de B, où A est le tableau des nombres de chemins de taille $n-1$: on doit calculer récursivement ce tableau, par analogie avec le calcul de puissance des entiers : si $n = 1$, $x^n = x$, si $n > 1, x^n = x \times x^{n-1}$. De même, ici, si $n = 1$, on renvoie A, sinon, on renvoie $A^{n-1} \times A$. C'est dans la matrice A^n qu'on cherchera les chemins de taille n.

```
def puissMat(A,n):
    if n==1:
        return A
    else:
        return (prod (puissMat(A,n-1),A))
```

(e) Si on veut savoir si une ville est accessible depuis a, il suffit de savoir s'il existe un chemin (ou non) depuis a. Or, il y a m villes : s'il existe un chemin entre a et b, ou plusieurs, le plus petit d'entre eux est de taille au plus $m-1$ (sans quoi on trouverait une boucle dans le chemin : un tel chemin peut être raccourci en en enlevant

la boucle). On pourrait donc observer $A^k[a][b]$ pour tout k entre 0 et $m-1$ (A^0 est alors l'**identité**, qui n'a de 1 que sur sa diagonale est des 0 partout ailleurs, ce qui correspond à dire qu'il n'y a de chemin de longueur 0 qu'entre une ville et elle-même) : si l'un de ces termes est non nul, il existe bien un chemin.

Autre possibilité : on rajoute artificiellement des arêtes de chaque ville vers elle-même. Alors, A^k compte les chemins de longueur k dans le graphe où un tour sur soi-même compte pour une unité de longueur, c'est-à-dire tous les chemins de longueurs plus petite ou égale à k dans le graphe original. Il suffit alors d'observer $A^{m-1}[a][b]$.*On ne regarde ici que les villes accessibles depuis la ville "départ", donc dans un sens précis.*

```
# on ajoute un chemin de chaque ville dans elle-même s'il n'existe pas
def ajoutDiag(A):
    for i in range(len(A)):
        A[i][i]=1


def connecteesAuDepart(A,depart):
    m = len(A)
    ajoutDiag(A)
    B = puissMat(A,m-1)
    listeVilles = []
    for ville in range(m):
        if B[depart][ville]>0:
            listeVilles.append(ville)
    return listeVilles
```