Partie 1

Analyser un algorithme

1.1 Preuve de correction d'un algorithme récursif

Alan Turing (1912 – 1954) énonce ainsi son principe d'indéciablité de la terminaison d'un algorithme(1936) : *L'indecidabilite c'est l'impossibilité absolue et définitivement démontrée de résoudre par un procédé général de calcul un problème donné.*

La terminaison, ainsi que la correction ou la complexité doivent être prouvées par des arguments mathématiques.

- terminaison : recherche du convergent
- correction / validité : recherche de l'invariant de boucle pour démontrer sa (non) variation selon un argument de recurence
- puis finir en montrant que l'invariant de boucle ou bien le resultat obtenu repond bien à la specification de l'algorithme

1.2 Terminaison

1 Principe : Un programme ou une fonction *termine* si le nombre d'instructions à effectuer est fini. La méthode consiste à démontrer, par des arguments logiques, que le variant se rapproche de la condition d'arrêt. La réponse peut aussi s'appuyer sur un *tableau de suivi des variables.*

Remarque : "Normalement" les boucles for en Python itèrent un nombre fini de fois donc dans ce cas il est facile de prouver la terminaison puisqu'il y aura automatiquement un nombre fini d'itérations. Les principales sources de non terminaison sont les boucles while (tant-que) et les appels récursifs (notion au programme de terminale). C'est dans l'un de ces 2 cas (qui arrive en fait assez souvent) que la terminaison peut être difficile à prouver. Pour cette année, pour prouver la terminaison d'un algorithme il faudra montrer que les boucles "tant que" ne bouclent pas à l'infini, c'est-à-dire que leurs conditions deviennent fausses au bout d'un certain nombre d'itérations.

2. Retour sur la notion de variant de boucle. Pour prouver la terminaison d'un algorithme qui contient une boucle "tant que", on doit identifier un "variant de boucle".

Définition : Un variant de boucle est une expression (c'est souvent le simple contenu d'une variable) dont la valeur varie à chacune des itérations de la boucle et dont on peut démontrer que la valeur :

- est un entier positif tout au long de boucle,
- décroit strictement à chaque itération,
- et qui, si elle devient négative (ou nulle) assure qu'on sort de la boucle.

Si on arrive à trouver un variant de boucle alors on peut conclure que la boucle termine.

3. Exemple

```
i = 2
while i > 0:
    p = p * 2
    i = i - 1
```

La variable i joue le rôle d'un compteur, au début de l'algorithme i contient le nombre d'itérations à faire. Après chaque itération, on enlève 1 à i. La condition d'arrêt de la boucle teste si toutes les itérations ont été faites.

- i définit un variant de boucle car :
  - i est un entier strictement positif,
  - i décroit strictement (i = i - 1) à chaque itération,
  - si i devient inférieur ou égal à 0 (donc négatif) alors on sort de la boucle.

On a trouvé un variant de boucle donc il y a un nombre fini d'itérations et donc l'algorithme termine.

1.3 Correction

1. Principe : Un algorithme est correct (partiellement) s'il reproduit directement une formule, ou s'il fait bien ce pour quoi il a été conçu.
2. Invariant de boucle Pour prouver la correction (partielle) d'un algorithme, on utilise un invariant de boucle.

Définition : On appelle invariant de boucle une propriété P telle que :

- P est vérifiée avant l'entrée dans la boucle,
- si P est vérifiée avant une itération, alors elle reste vérifiée après cette itération, c'est à dire que P est préservé lors d'un passage dans la boucle au suivant et
- en particulier P reste vérifiée en sortie de boucle, quand on la quitte.

Dans la pratique, on procède alors en 3 étapes pour **prouver la correction (partielle)** d'un algorithme :

- 1ère étape : Initialisation. On prouve (en général c'est une simple vérification) que l'invariant est vrai avant le premier passage ;
- 2ème étape : Hérédité (ou conservation). On montre qu'il est préservé d'une itération à la suivante ;
- 3ème étape : Conclusion. On conclut qu'il est vrai lorsque l'on quitte la boucle.

Remarque : Ce type de démonstration est très proche du raisonnement par récurrence vu en mathématiques en terminale et l'invariant de boucle qu'on utilise en informatique correspond à "l'hypothèse de récurrence" mathématique.

3. Exemple : Calculer la somme des entiers naturels de 0 jusqu'à un entier n

```
def somme(n):
    s = 0
    for i in range(n + 1):
        s = s + i
    return s
```

L'algorithme n'est pas réduit à la simple application d'une formule ou bien à quelques instructions en nombre fini.

Terminaison : Il faut être sûr qu'il y a un nombre fini d'étapes. Cet algorithme a une boucle mais c'est une boucle for donc il y a un nombre fini d'itérations : n + 1 itérations, puisque i varie de 0 à n. Le nombre d'instructions étant fini, l'algorithme termine.

Correction : Il faut vérifier que s contient à la fin de l'exécution le résultat attendu, c'est à dire la somme $0 + 1 + 2 + 3 + ... + n$

Il faut vérifier les étapes :

- L'initialisation de s est correcte (s = 0 au début, la somme de 0 à 0 est bien égale à 0).
- Les modifications de s à chaque itération sont correctes : on ajoute i à chaque itération et i varie de 0 à n en ajoutant 1 à i à chaque étape donc à toute étape k (k étant inférieur à n), on aura calculé $0 + 1 + .. + k$
- enfin le nombre d'itérations est le bon (n+1 itérations ce qui correspond bien au nombre de nombres de 0 à n) donc le résultat sera bien celui attendu et donc la correction est vérifiée.

4. Exemple : maximum d'une liste

```
def maximum(L):  #L est une liste de n nombres.
    maxi = L[0]
    for i in range(1, len(L)):
        if L[i] > maxi :
            maxi = L[i]
    return maxi
```

On note la liste $L = [l_0 + ... + l_{n-1}]$

Pour prouver la correction de cette fonction, il faut trouver un invariant de boucle, donc il faut trouver une propriété qui est vraie avant d'exécuter le 1er tour de la boucle for et qui le reste.

Soit i un entier strictement positif, on note P(i) la propriété : $maxi = maximum([l_0, l_1, l_{i-1}])$. On va démontrer que cette propriété est un invariant de boucle.

- Initialisation :

On vérifie que P est vraie pour i = 1, c'est à dire que $maxi = maximum([l_0, l_{1-1}])$

Avant la 1ère itération (en entrant dans la boucle), $maxi = L[0]$ et donc on a bien $maxi = maximum([l_0])$

- Hérédité :

Supposons que P soit vraie au début de l'itération d'indice i, c'est à dire que $maxi = maximum([l_0, l_1, l_{i-1}])$ et montrons alors qu'elle sera conservée par itération de la boucle, c'est à dire que $maxi = maximum([l_0, l_1, l_{i-1},$

On a : $maxi = maximum([l_0, l_1, l_{i-1}])$

Si $l_i <= maxi$ alors $maxi$ ne change pas et donc $maxi = maximum([l_0, l_1, l_{i-1}]) = maximum([l_0, l_1, l_{i-1}, l_i])$ à la fin de l'itération d'indice i.

Si $l_i >= maxi$ alors $maxi = l_i$ et le maximum de $[l_0, l_1, l_{i-1}, l_i]$ est bien l_i puisque c'est la plus grande valeur donc $maxi = maximum([l_0, l_1, l_{i-1}]) = maximum([l_0, l_1, l_{i-1}, l_i])$

Donc dans les 2 cas, P est conservée par itération de la boucle.

- Conclusion : En particulier l'invariant sera toujours vrai après la dernière itération de la boucle, celle d'indice n−1. Avant cette itération, $maxi = maximum([l_0, l_1, l_{i-2}])$ et donc après cette itération, $maxi = maximum([l_0, l_1, l_{i-1}])$ est donc bien le maximum de tous les éléments de L.

On a donc bien prouvé la correction de cette fonction.

Partie 2

À retenir : Comportement d'un algorithme (ou d'un programme)

Donne-t-il un résultat ? boucle-t-il à l'infini ou génère-t-il une erreur ?

-> Terminaison de l'algorithme : Utiliser un variant de boucle.

Donne-t-il LE résultat attendu ?

-> Correction de l'algorithme : Trouver et utiliser un invariant de boucle.

Est-il performant ? Combien de temps met-il à s'exécuter ?

-> Étude de la complexité de l'algorithme (en temps).

Partie 3

Sources

bien clair sur nsirenoir : preuve de correction et de terminaison