# Preuves en informatique

Cours 03

## 1 Correction et terminaison

**Définition 1**

La spécification d'un algorithme consiste à préciser les entrées et les sorties attendues d'un algorithme.

Une spécification-jouet d'un algorithme simple :

**Entrée :** un entier naturel positif $n \in \mathbb{N}$

**Sortie :** un diviseur pair de $n$ s'il existe, $-1$ sinon.

**Définition 2**

Soit $A$ un algorithme.

— la correction partielle de $A$ signifie que lorsque $A$ termine, alors il renvoie ce qui est attendu de lui ;

— la terminaison de $A$ signifie que $A$ termine **sur toute entrée** ;

— la correction de $A$ signifie la correction partielle et la terminaison de $A$.

Reprenons notre spécification précédente :

```
def ppdp(n):
    d = 2
    while n%d != 0:
        d += 2
    return d
```

```
def ppdp(n):
    return 2
```

La fonction de droite termine, mais n'est pas partiellement correcte ; la fonction de gauche est partiellement correcte, mais ne termine pas sur toute entrée.

On parle aussi de **précondition** (qu'a-t-on en entrée ?) et de **postcondition** (qu'exige-t-on de la sortie ?). Il s'agit d'une interprétation contractuelle de l'informatique : si l'entrée vérifie la précondition, alors la sortie doit vérifier la postcondition.

### 1.1 Preuves de terminaison

Pour prouver la terminaison d'un algorithme, il existe deux types d'arguments :

— l'algorithme est exclusivement composé de boucles finies (= boucles « pour ») ;

— un argument par variant.

Techniquement, en Python, il est possible de créer des boucles `for` infinies ; il s'agit moins d'une difficulté théorique que d'un défaut du langage Python.

Pour construire un argument par variant, il faut exhiber une grandeur qui dépend des variables de l'algorithme, qui est entière, qui est positive, et qui décroît strictement.

Techniquement, on pourrait généraliser les variants aux ordres bien fondés (voir l'hydre de Kirby-Paris).

## 1.2 Preuves de correction partielle

Avant de commencer une preuve de correction partielle, on commence par établir la spécification souhaitée de l'algorithme, afin de savoir dans quelle direction aller.

Si votre programme ne fait aucune itération (pas de boucle, pas de récursivité, ...), on analysera le programme à la main.

Dès qu'il y a une boucle ou de la récursivité, la principale technique d'analyse est la méthode de l'invariant. Un invariant est une propriété mathématique, dépendante des variables du programme, qui :

— est vraie avant le début de l'itération ;

— est préservée à la fin d'une itération.

Ici, notre spécification est que l'entrée `N` est un entier positif, et que la sortie est $\sum_{k=0}^{\texttt{N}} k^2$.

Ici, on choisit comme propriété : « $\texttt{s} = \sum_{k=0}^{\texttt{i}} k^2$ ».

```python
def somme_carres(N):
    s = 0
    for i in range(N+1):
        s += i**2
    return s
```

On cherche à montrer que notre propriété est bien un invariant.

L'initialisation de notre propriété est un peu délicate : que vaut `i` avant le début de la boucle ? Trois solutions plus ou moins convaincantes s'offrent à vous.

— considérer que `i` n'existe pas, donc la somme est est une somme vide ;

— considérer que `i = -1`, donc la somme est est une somme vide ;

— réécrire le programme avec une boucle `while`, et potentiellement changer d'invariant.

Montrons maintenant la préservation notre propriété : supposons, à la fin d'une itération, que $\texttt{s} = \sum_{k=0}^{\texttt{i}} k^2$. Montrons, à la fin de l'itération suivante, qu'on a toujours $\texttt{s} = \sum_{k=0}^{\texttt{i}} k^2$. Pour ce faire, on va distinguer nos variables entre l'ancienne itération et la nouvelle itération : $\texttt{s}_a$ et $\texttt{i}_a$ désignent les valeurs des variables dans l'itération précédente, tandis que $\texttt{s}_n$ et $\texttt{i}_n$ désignent les valeurs des variables dans l'itération actuelle. En particulier, on a $\texttt{i}_n = \texttt{i}_a + 1$. On remarque ensuite que :

$$\begin{aligned}\texttt{s}_n &= \texttt{s}_a + \texttt{i}_n^2 \\ &= \sum_{k=0}^{\texttt{i}_a} k^2 + (\texttt{i}_a + 1)^2 \\ &= \sum_{k=0}^{\texttt{i}_a+1} k^2 = \sum_{k=0}^{\texttt{i}_n} k^2\end{aligned}$$

Donc on a bien $\texttt{s}_n = \sum_{k=0}^{\texttt{i}_n} k^2$ ; la propriété est bien préservée par l'itération de la boucle. Donc la propriété est bien un invariant.

En conclusion, à la fin du dernier passage de boucle, on a $\texttt{s} = \sum_{k=0}^{N} k^2$ ; donc notre programme vérifie bien la spécification souhaitée.

# 2 Complexité(s)

**Définition 3 : complexités temporelles**

Soit $A$ un algorithme. La complexité temporelle de $A$ dans le pire cas est la fonction $C_A : \mathbb{N} \to \mathbb{N}$ qui, à un entier $n$, associe le plus grand nombre d'opérations élémentaires nécessaire pour que $A$ traite une entrée de taille $n$.

La complexité moyenne est la moyenne des nombres d'opérations pour toutes les entrées d'une même taille.

Deux notions sont laissées volontairement floues dans ce contexte, « opérations élémentaires » et « taille » : ces notions dépendent du contexte. Par exemple, en général, on traite les opérations arithmétiques comme étant élémentaires ; mais dans le cadre de questions arithmétiques (comment encoder l'addition en machine, par exemple), ces opérations ne sont plus élémentaires. De même, la taille dépend de ce qu'on traite. On considère par exemple généralement que les entiers sont de taille constante, mais lorsqu'on s'intéresse aux problèmes arithmétiques, ce n'est plus le cas : un entier $n$ est de taille $\log(n)$.

**Définition 4 : complexité spatiale**

Soit $A$ un algorithme. La complexité spatiale de $A$ dans le pire cas est la fonction $\tilde{C}_A : \mathbb{N} \to \mathbb{N}$ qui, à un entier $n$, associe la taille mémoire de travail maximale nécessaire pour que $A$ traite une entrée de taille $n$.

Considérons les deux exemples suivants. Dans `exemple`, l'entrée est une matrice, donc l'entrée est potentiellement de taille quadratique : pourtant, on ne fait que renvoyer 1. On n'a pas eu besoin d'accéder et surtout **de stocker** temporairement les valeurs dans `matrice` pour travailler : on n'a pas besoin de mémoire pour exécuter la fonction.

```
def exemple(matrice):
    return 1
```

Dans le second exemple, il peut y avoir deux interprétations. Selon la première, on a besoin d'avoir la place pour faire les additions, et la place de stocker la matrice : on obtient une complexité en $O(|m|)$. Selon la seconde, les cases mémoires modifiées ne sont pas des cases de travail, car elles sont fournies par l'utilisateur : donc seule l'addition compte, ce qui donne une complexité en $O(1)$.

```
def exemple2(m):
    for i in range(len(m)):
        for j in range(len(m[0])):
            m[i][j] += 1
    return m
```

## 2.1 Établir la complexité temporelle : cas usuel des boucles « pour »

Lorsqu'on traite des boucles « pour » bornées, il est généralement suffisant de connaître le nombre d'opérations par passage de boucle, et le nombre total de passages de boucle.

Déterminons, hors de tout contexte, la complexité du programme suivant.

```
9   for i in range(n):
10      for r in range(b+1):
11          if obj[i][0]<=r:
12              T[i+1][r]=max(T[i][r],T[i][r-obj[i][0]]+obj[i][1])
13          else:
14              T[i+1][r]=T[i][r]
```

Observons les variables utilisées :

- `obj` semble être une matrice ;
- `T` aussi ;
- `n` et `b` sont des entiers.

On peut alors faire l'analyse suivante :
« Les opérations faites entre la ligne 11 et la ligne 14 sont en temps constant (affectations, calcul d'un max, opérations arithmétiques). Donc la boucle débutant ligne 10 fait $O(b)$ passages de boucle, chacun en $O(1)$ ; donc cette

boucle est en $O(\texttt{b})$. Donc la boucle débutant ligne 9 fait $O(\texttt{n})$ passages de boucle, chacun en $O(\texttt{b})$ ; au total, notre programme s'exécute en au plus $O(\texttt{n} \times \texttt{b})$ opérations élémentaires. »

## 2.2 Complexité temporelle par variant

Une autre technique consiste à attacher notre complexité à l'étude d'un variant de boucle.

De nouveau, cherchons à examiner la complexité de ce morceau de code sans comprendre ce qu'il fait précisément.

```
6       k=n-1
7       r=b
8       while r>0 and k>=0 and T[k+1][r]!=0:
9           while k>0 and T[k+1][r]==T[k][r]:
10              k-=1
11          S[k]=1
12          r=r-obj[k][0]
13          k-=1
```

Ici, on observe que `k` est un variant pour les deux boucles : `k` décroît strictement à chaque passage de des deux boucles. On peut alors faire l'analyse suivante :
« On observe qu'à chaque passage de la boucle débutant ligne 9 et chaque passage de la boucle débutant ligne 8, `k` décroît strictement. Les conditions de sortie de ces boucles font qu'on en sort si `k` est négatif (strictement ou non selon la boucle) ; donc on peut décroître `k` au plus `n-1` fois.

Soit $p$ le nombre de passages de boucles de la boucle 8, et $q_i$ le nombre de passages de la boucle 9 lors de la $i$-ème itération de la boucle 8. Chaque étape comptée par $q_i$ correspond à des décrémentations de `k`. Notre programme effectue donc au plus :

$$\begin{aligned}\sum_{i=1}^{p}(q_i + O(1)) &= \sum_{i=1}^{p} q_i + O(p) \\ &= O(n) + O(n) = O(n) \text{ opérations élémentaires dans le pire cas ».}\end{aligned}$$

## 2.3 Complexité par équation de récurrence

Ce cas s'applique principalement au cas des fonctions récursives. Il s'agit d'établir une (in)équation de récurrence à partir du schéma de récursion utilisé par le programme. La difficulté consiste principalement à résoudre ladite (in)équation.

Reprenons l'équation de récurrence de la complexité du tri fusion.

$$C(n) = 2C\left(\frac{n}{2}\right) + n$$

On remarque, en remplaçant l'occurrence de $C$ à droite par lui-même, qu'on obtient les égalités successives suivantes :

$$\begin{aligned}C(n) &= 2C\left(\frac{n}{2}\right) + n \\ &= 2\left(2C\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n = 4C\left(\frac{n}{4}\right) + 2n \\ &= 8C\left(\frac{n}{8}\right) + 3n\end{aligned}$$

De manière générale, on montre par récurrence que $C(n) = 2^k C\left(\frac{n}{2^k}\right) + kn$. En prenant $k \simeq \log(n)$, on obtient alors que $C(n) = n\log(n) + nC(1) = O(n\log(n))$.