Fiche 8 - Complexité, correction, terminaison

I. Complexité

Les programmes informatiques sont parfois utilisés pour traiter un grand nombre de données ou des données de grandes tailles. Pour améliorer le traitement on peut envisager :
- d'augmenter la puissance de calcul de la machine
- de répartir le travail sur plusieurs machines
- d'élaborer un nouvel algorithme

La **complexité** est une **méthode théorique** permettant de comparer des algorithmes sans avoir besoin de les programmer et encore moins de les tester. Elle mesure l'efficacité d'un algorithme, c'est-à-dire, sa capacité à traiter plus d'informations avec une puissance de calcul fixée.

Cette mesure porte en général sur l'étude de deux paramètres :
- le temps d'exécution
- l'espace mémoire utilisé

La mesure peut être faite :
- dans le pire des cas
- dans le meilleur des cas

Par fois on introduit aussi une complexité *moyenne* mais en général, c'est la complexité du pire qui fait référence.

On exprime la complexité avec la notation $\mathcal{O}(\ )$. Voici les familles courantes de complexité :
- logarithmique : $\mathcal{O}(\ln(n))$
- linéaire : $\mathcal{O}(n)$
- polynômiale : $\mathcal{O}(n^k)$
- exponentielle : $\mathcal{O}(\alpha^n)$ avec $\alpha > 1$.

Exercice 1 Donner une estimation de la complexité, en terme d'opérations binaires, de la somme, du test d'égalité (ou la comparaison) et du produit de deux nombres $n_1, n_2 \in [\![0, N]\!]$ en fonction de N.

La complexité est exprimée en fonction de la taille des paramètres d'entrée. On peut dénombrer toutes les opérations (binaire) ou se limiter à une opération qui soit significative de l'algorithme et en général celle qui à la plus grande complexité.

Exemple : Soit $n \in \mathbb{N}^*$. On cherche tous les nombres de $[\![0, n]\!]$ qui s'écrivent comme la somme de deux carrés.

Algorithme 1
```
pour i variant de 0 à n faire
  pour j variant de 0 à n faire
    pour k variant de 0 à n faire
      si i = j^2 + k^2 alors afficher i finsi
    finpour
  finpour
finpour
```

On peut dénombrer :
- les additions : $(n+1)^3 \sim n^3$
- les multiplications : $2(n+1)^3 \sim 2n^3$
- les tests d'égalité : $(n+1)^3 \sim n^3$
- les affichages : moins de $(n+1)^3$

La complexité de cet algorithme est polynômiale : $\mathcal{O}(n^3)$.

Ici, la complexité résulte essentiellement du nombre d'*étapes* résultant des trois boucles imbriquées. On peut considérer la multiplication* comme l'opération de référence.

Algorithme 2
```
pour i variant de 0 à n faire
  pour j variant de 0 à √i faire
    si i − j^2 est un carré alors afficher i finsi
  finpour
finpour
```

L'amélioration consiste en une vérification judicieuse : inutile de tester tous les nombres lorsqu'un seul est possible !

On dénombre :
- les itérations (boucles) : $\sum_{i=0}^{n} \sqrt{i} \sim \frac{2}{3} n\sqrt{n}$
- les multiplications : $\sim \frac{2}{3} n\sqrt{n}$
- on note l'introduction d'un autre algorithme : calcul d'une racine carrée. Proposez un algorithme ? Une complexité ?
- il y a les $n+1$ bornes (calculs de racines carrées) de la boucle imbriquée
- il y a dans chaque boucle un test de carré parfait (calcul de racine carré)

Attention ! Introduire une nouvelle opération nous impose d'estimer le sur-coût calculatoire. La complexité du calcul d'une racine carrée d'un nombre inférieur à n compte moins de $\sqrt{n}$ multiplication : $\mathcal{O}(\sqrt{n})$.

La complexité de cet algorithme est : $\mathcal{O}(n^2)$.

Algorithme 3
```
pour j variant de 0 à √n faire
  pour k variant de j à √(n − j^2) faire
    afficher j^2 + k^2
  finpour
finpour
```

L'amélioration de cette approche réside dans le fait que l'on décrit directement l'ensemble solution au lieu de parcourir tous les nombres et de tester s'ils vérifient la propriété.

La complexité de cet algorithme est linéaire : $\mathcal{O}(n)$.

Remarque : Dans cet exemple, nous nous sommes intéressés uniquement à la complexité temporelle et non spatiale.

Exercice 2 Estimer la complexité *temporelle* des algorithmes suivants, en fonction $n \in \mathbb{N}^*$:

1.
```
Algo A
pour i variant de 0 à n faire
  pour j variant de 0 à i faire
    afficher i×j
  finpour
finpour
```

2.
```
Algo B
i ← 0
tant que i<n faire
  afficher i²
  i ← i+1
fintantque
```

3.
```
Algo C
i ← 0 ; j ← 0
tant que j<n faire
  i ← i+1
  j ← j+i
fintantque
afficher i
```

Exercice 3 Voici une fonction qui affiche tous les diviseurs d'un entier naturel n :

```python
def diviseurs(n):
    for i in range(1,n+1):
        if n%i==0:
            print(i)
```

1. Estimer la complexité de cette fonction.
2. Proposer une autre fonction de complexité moindre.

II. Correction - terminaison

➜ La **correction** de l'algorithme est la vérification que le résultat de sorti est conforme aux attentes.

On appelle **invariant de boucle** un propriété qui est vraie au début (et à la fin) de chaque boucle d'une structure répétitive. C'est un *outil théorique* permettant de prouver la correction qu'un algorithme.

Remarque : Établir un invariant de boucle s'apparente à faire une récurrence.

➜ La **terminaison** de l'algorithme est la vérification que l'algorithme se termine en temps fini.

On appelle **variant de boucle** une variable (ou une propriété) qui varie à chaque boucle d'une structure répétitive. Si le variant prend un nombre fini d'états, alors cela donne un preuve de la terminaison d'un algorithme.

Exemple : Recherche du reste et du quotient de la division euclidienne de a par b par soustractions successives, avec $a, b \in \mathbb{N}^*$.

Algorithme de la division euclidienne de a par b
```
q ← 0
r ← a
tant que r ≥ b faire
  r ← r-b
  q ← q+1
fintantque
afficher (q,r)
```

➢ Un invariant de boucle est a=qb+r.

Faire un récurrence pour établir l'invariant.
A l'issue de la boucle, il vient : $r \in [\![0, b-1]\!]$ et a=qb+r ce qui prouve bien que q et r sont respectivement le quotient et le reste de la division euclidienne de a par b.

➢ Un variant de boucle est r.

La suite des valeurs contenues dans la variable r est une suite strictement décroissante de $\mathbb{N}$ et de premier terme a. Ainsi, en au plus a étapes, la condition r<b est vérifie, c'est-à-dire l'algorithme s'arrête.

➜ La correction et la terminaison sont établies !

Exercice 4 L'algorithme suivant calcule $\sum_{k=0}^{n} \frac{x^k}{k!}$ avec x un nombre et $n \in \mathbb{N}$.

1. Compléter l'algorithme d'entrées x et n :

```
s ← ...
m ← ...    # numérateur
f ← ...    # dénominateur
pour k variant de 1 à n faire
  s ← s + m/f
  m ← ...
  f ← ...
fintantque
afficher s
```

2. Définir l'invariant pour $k \in [\![1, n]\!]$.
3. Justifier la correction de l'algorithme.
4. Justifier la terminaison de l'algorithme.
5. Bonus - compléter à nouveau l'algorithme en tenant compte de l'ordre des affectations :

```
s ← ...
m ← ...
f ← ...
pour k variant de 1 à n faire
  m ← ...
  f ← ...
  s ← ...
fintantque
afficher s
```