Notions de base (Cours)

S6 - Algorithmique (1)

1. Introduction

L'**algorithmique** est la science qui étudie les **algorithmes**. Un algorithme est une suite d'instructions permettant de résoudre un problème.

Pour résoudre un problème, il existe souvent plusieurs algorithmes possibles.

Les objectifs de cette partie du cours sont d'apprendre à :

- prouver qu'un algorithme donné se termine en un temps fini ;
- prouver qu'un algorithme donné réalise bien ce pour quoi il a été écrit ;
- comparer deux algorithmes différents répondant au même problème.

2. Terminaison d'un algorithme

> Définition
>
> Prouver la **terminaison** d'un algorithme, c'est prouver que l'algorithme se termine dans tous les cas. C'est notamment très important lorsque l'algorithme comporte des boucles conditionnelles.

Prenons comme exemple l'algorithme suivant qui calcule la puissance entière d'un nombre :

```
def puissance(x: float, n: int) -> float:
    """retourne x^n"""
    p = 1
    compteur = 0
    while compteur < n:
        compteur = compteur + 1
        p = p * x
    return p
```

Comment justifier que cet algorithme se termine dans tous les cas ? Cela revient à montrer que la boucle conditionnelle `while compteur < n` se termine après un nombre fini d'itérations. Pour cela, on utilise un **variant de boucle** : c'est une valeur qui évolue à chaque itération de la boucle et qui permet de prouver que celle-ci se termine.

Dans notre exemple, on peut choisir comme variant de boucle la valeur de la variable *compteur*. En effet, cette variable est initialisée à 0 et est incrémentée de 1 à chaque itération. Après n itérations, la condition de sortie de boucle sera donc vraie et la boucle se terminera.

La **terminaison** de l'algorithme est donc démontrée.

3. Correction d'un algorithme

Définition

Prouver la **correction** d'un algorithme, c'est prouver que l'algorithme réalise bien ce pour quoi il a été écrit.

Considérons à nouveau l'algorithme de calcul de puissance entière. Comment prouver que cet algorithme calcule bien la puissance entière d'un nombre ?

On utilise pour cela un **invariant de boucle** : c'est une propriété qui est vraie avant et après chaque itération de la boucle, et qui doit permettre de prouver que l'algorithme réalise bien ce pour quoi il a été écrit.

Dans notre exemple de calcul de puissance entière, on peut choisir comme invariant de boucle la propriété suivante :

$$p = x^{compteur}$$

Pour prouver q'une propriété est un invariant de boucle…

Il faut démontrer :

1. **Initialisation** : la propriété est vraie avant le premier passage dans la boucle
2. **Conservation** : si la propriété est vraie avant une itération, alors elle sera aussi vraie après cette itération.
3. **Conclusion** : une fois la boucle terminée, la propriété est vraie.

Cette méthode de raisonnement est appelée **raisonnement par récurrence** et est très utilisée en mathématiques (au programme en spécialité mathématiques de terminale).

Dans notre exemple, on a :

- **Initialisation** : La propriété est vraie avec les valeurs initiales des variables car $x^0 = 1$ et $p = 1$.
- **Conservation** : Si nous avons $p = x^{compteur}$ avant une itération, alors nous avons $x^{compteur+1} = x^{compteur} \times x = p \times x$. le passage dans la boucle augmente *compteur* de 1 et remplace p par $p \times x$. Après l'itération, la propriété $p = x^{compteur}$ est donc encore vraie.
- **Conclusion** : En sortie de boucle, on a donc $p = x^{compteur}$. Or on a aussi l'égalité $compteur = n$ qui a provoqué la sortie de boucle. Finalement, nous avons donc $p = x^n$, ce qui prouve que l'algorithme effectue bien l'opération attendue.

4. Complexité

La durée d'exécution d'un programme traduisant un algorithme donné va dépendre des performances de la machine sur laquelle le programme est exécuté, mais aussi du **nombre d'instructions élémentaires** mobilisées lors de son exécution. Une partie de ce temps d'exécution provient donc de la façon dont l'algorithme est écrit et non de la façon dont il est programmé.

On parle de **complexité temporelle** d'un algorithme (et non d'un programme) pour mesurer l'efficacité **intrinsèque** de l'algorithme. Dans la pratique, il s'agit de compter le nombre d'opérations élémentaires (affectations, comparaisons, calculs arithmétiques, …) effectuées par l'algorithme.

La complexité en temps d'un algorithme dépend :

- de la taille des données passées en paramètres : plus ces données seront volumineuses, plus il faudra d'opérations élémentaires pour les traiter. On notera n le nombre de données à traiter.

- de la donnée en elle-même, de la façon dont sont réparties les différentes valeurs qui la constituent. Par exemple, si on effectue une recherche séquentielle d'un élément dans une liste non triée, on parcourt un par un les éléments jusqu'à trouver, ou pas, celui recherché. Ce parcours peut s'arrêter dès le début si le premier élément est "le bon". Mais on peut également être amené à parcourir la liste en entier si l'élément cherché est en dernière position, ou même n'y figure pas.

Cette remarque conduit à préciser la définition de la complexité en temps. On peut en effet distinguer deux formes de complexité en temps :

- **la complexité dans le meilleur des cas** : c'est la situation la plus favorable, par exemple : recherche d'un élément situé à la première position d'une liste ;
- **la complexité dans le pire des cas** : c'est la situation la plus défavorable, par exemple : recherche d'un élément dans une liste alors qu'il n'y figure pas.

On calculera le plus souvent la complexité dans le pire des cas, car elle est la plus pertinente. Il vaut mieux en effet toujours envisager le pire.

Ordres de grandeurs

Pour comparer des algorithmes, il n'est pas nécessaire de calculer la valeur exacte de la complexité, mais seulement un **ordre de grandeur asymptotique**, noté en mathématiques $\mathcal{O}$ (notation "grand O"). La définition rigoureuse de cette notation n'est pas au programme de NSI. Il faut cependant en avoir une idée intuitive : dire que la complexité d'un algorithme est en $\mathcal{O}(n^2)$, par exemple, signifie que cette complexité croît, lorsque n devient grand, de la même façon que la fonction carré. Plus précisément, elle est majorée par une fonction du type $c \times n^2$, où c est un réel positif.

Les classes de complexité à connaître en première, de la meilleure à la pire :

$\mathcal{O}$	Type de complexité	Exemple
$\mathcal{O}(1)$	constante	Accès à une cellule de tableau
$\mathcal{O}(n)$	linéaire	Recherche du maximum dans un tableau non trié
$\mathcal{O}(n^2)$	quadratique	Parcours d'un tableau à deux dimensions
$\mathcal{O}(n^3)$	cubique	Parcours d'un tableau à trois dimensions

Exemple

Reprenons l'algorithme du calcul de la puissance d'un nombre.

```
def puissance(x: float, n: int) -> float:
    """retourne x^n"""
    p = 1
    compteur = 0
    while compteur < n:
        compteur = compteur + 1
        p = p * x
    return p
```

Nous comptons la complexité en termes d'opérations arithmétiques : additions et multiplications. À chaque passage dans la boucle, nous avons deux opérations et la boucle est parcourue n fois .

Nous avons donc au total une complexité de $2n$ opérations arithmétiques, donc une complexité en $\mathcal{O}(n)$, linéaire.

Visualisation graphique du temps d'exécution

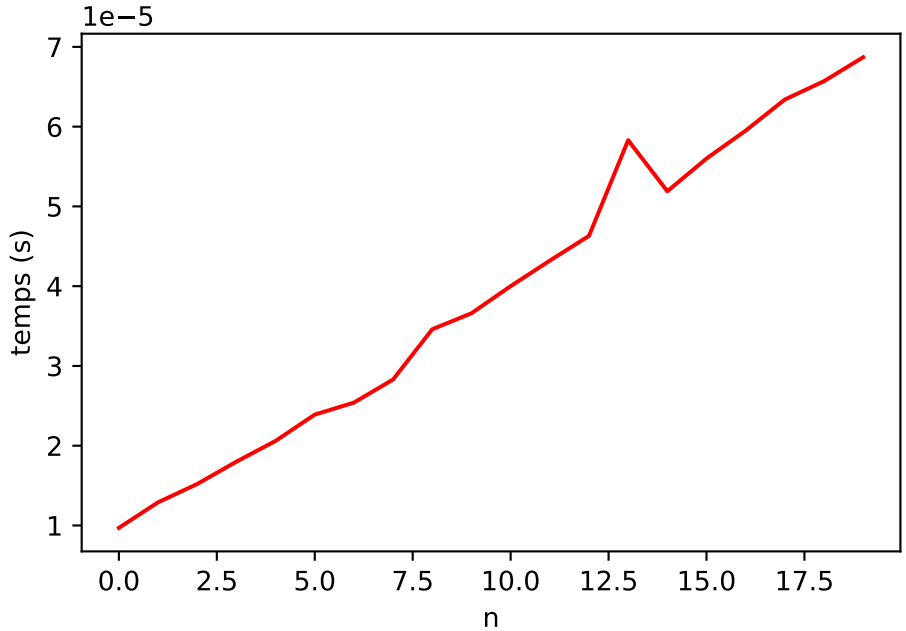
En utilisant le module `timeit` comme expliqué dans cet article, on peut visualiser graphiquement le temps d'exécution de l'algorithme en fonction de la taille des données.

```
import matplotlib.pyplot as plt
import timeit

def puissance(x: float, n: int) -> float:
    """retourne x^n"""
    p = 1
    compteur = 0
    while compteur < n:
        compteur = compteur + 1
        p = p * x
    return p

abscisses = [k for k in range(0, 20, 1)]
ordonnees = []
for n in abscisses:
    ordonnees.append(timeit.timeit('puissance(2, n)', number=100, globals=globals()))

fig, ax = plt.subplots()
ax.set_xlabel('n')
ax.set_ylabel('temps (s)')
plt.plot(abscisses, ordonnees, 'r')
plt.show()
```



La courbe obtenue est proche d'une droite, ce qui est cohérent avec la complexité linéaire de

l'algorithme..