Cours 1 : Complexité

1. Algorithmes, programmes et efficacité

1.1. **Définition et premières questions.** Même si vous ne vous êtes jamais posé de questions d'algorithmique, en tant qu'étudiants vous avez été confrontés à l'écriture de **programmes**. Un **programme** est une suite d'instructions directement destinées à être exécutées par un ordinateur. Ainsi, le programme dépend du **langage** dans lequel il est écrit et sera traduit en instructions machine (compilation) qui seront exécutées par le processeur (exécution).

La notion d'algorithme est plus générale et permet d'aborder les problèmes de façon théorique sans s'occuper de l'aspect matériel.

Définition 1.1. *Un* Algorithme *est un procédé formel bien défini qui prend un ensemble de données en entrée (*Input*) et retourne un ensemble de données résultat (*Output*).*

Exemple 1.2.
Une recette de cuisine :
Input : les ingrédients.
Procédé : les instructions de la recette.
Output : le plat terminé.

En particulier, un algorithme ne dépend pas du langage de programmation. Un programme est une implantation d'un algorithme donné.

Exemple 1.3.
Problème : Chercher un nombre dans un tableau.
Algorithme :

```
Input : Un tableau de nombres T[0] à T[N-1], un nombre a
Procédé :
    Pour i entier allant de 0 à N-1, faire :
        Si T[i] = a :
            Retourner Vrai
    Retourner Faux
Output : Vrai ou Faux
```

Implantation en C :

```
int find(int T[], int n, int a) {
    int i;
    for(i=0; i< n; i++) {
        if(T[i] == a) {
            return 1;
        }
    }
    return 0;
}
```

Implantation en Python :

```
def find(T, a):
    for b in T:
        if b == a:
            return True
    return False
```

Un algorithme répond à un *problème*. c'est-à-dire à une description formelle de l'input et output attendu. Sur un même problème, on peut définir plusieurs algorithmes. Autour des notions de problèmes et algorithmes, se posent de nombreuses questions :

— Pour un problème donné, existe-t-il un algorithme ? *décidabilité d'un problème*
— S'il existe plusieurs algorithmes répondant à un problème, lequel est le "meilleur" ?
— L'algorithme termine-t-il ?
— L'algorithme donne-t-il la bonne réponse ?
— Combien de temps "dure" mon algorithme ? *complexité en temps*
— Combien d'espace mémoire nécessite-t-il ? *complexité mémoire*

Ces questions ne sont pas simples !

Exemple 1.4 (Syracuse)**.**
L'algorithme suivant termine-t-il quel que soit n ?

```
Input : un entier n supérieur ou égal à 1
Procédé :
    Tant que n > 1, faire :
        Afficher n
        Si n est pair :
            n <- n/2
        Sinon :
            n <- n*3+1
Output : l'affichage produit
```

Exemple 1.5.
Soit le problème suivant, qu'on appelle HALT :

```
Input : Un programme P écrit en python et un input possible T.
Output : Vrai si P(T) termine et Faux sinon
```

Si l'algorithme existait, on pourrait alors appeler HALT(Syracuse, 56) où Syracuse est l'implantation python de l'algorithme donné dans l'exemple 1.4 et HALT répondrait Vrai.

Existe-t-il un algorithme pouvant résoudre ce problème ? *Problème de l'arrêt.*

1.2. **Efficacité d'un programme.**

Exemple 1.6.
Lequel de ces deux programmes est le plus efficace ?

```
long puissance(int a, int n) {
    int i;
    long r = 1;
    for(i = 0; i < n; i++) {
        r = r*a;
    }
    return r;
}
```

```
def puissance(a,n):
    d = n.bit_length()-1
    r = 1
    while d >= 0:
        r = r*r
        if 1 & (n >> d) == 1:
            r = r*a
        d -= 1
    return r
```

Les deux programmes utilisent des algorithmes différents, un algorithme est-il intrinsèquement meilleur que l'autre ?

Problèmes :

— langages différents,
— sur quelles données ?
— optimisation bas niveau : code, compilation...
— dépendances de l'environnement

Cependant, il est possible d'étudier les algorithmes eux mêmes en se basant sur un **modèle théorique** qui permet de guider les choix de l'implantation pratique.

2. Complexité

2.1. **Complexité en temps : analyse d'algorithme.** On décrit ici un modèle pour **estimer le temps de calcul** d'un programme implantant un algorithme donné. Le modèle que l'on utilisera pour l'analyse d'algorithme s'inspire du fonctionnement de type **random-access machine (RAM)**. On suppose que les instructions sont exécutées les unes à la suite des autres sans opérations concurrentes (on ne traite donc pas des questions de parallélisme). On supposera aussi que chaque instruction élémentaire de l'algorithme prend un temps constant : par exemple, une ligne "Afficher n" prend un certain temps c qui ne dépend pas de n. Les instructions élémentaires correspondent à toutes les opérations arithmétiques (additions, multiplications, comparaisons), à l'accès et modification des données en mémoire (affichage, affectation) ainsi qu'aux sauts conditionnels ou inconditionnels (boucles). **Attention**, ceci est un **choix du modèle** qui correspond à une simplification de la réalité. En particulier, lorsqu'on travaille sur des grands nombre, on pourra choisir d'autres modèles où la multiplication et l'addition ne sont plus considérées comme constantes.

Exemple 2.1 (Analyse de l'algorithme puissance)**.**
On reprend l'algorithme du programme calculant la puissance en C (Exemple 1.6) et on estime son temps de calcul à l'aide du modèle décrit.

```
Input : deux entiers a et n
Procédé :
1.      r <- 1                                  c1
2.      Pour i allant de 0 à n-1:               c2
3.          r <- r*a                            c3
4.      Retourner r                             c4
Output : a puissance n
```

On suppose que $c_1, \dots c_5$ sont les constantes qui correspondent à l'exécution des différentes lignes de code. Les lignes en dehors de la boucle seront exécutées une fois, celles de la boucle n fois. On obtient donc que le temps d'exécution selon notre modèle est donné par

$$c_1 + c_4 + n(c_2 + c_3). \tag{2.1}$$

Le résultat est de type $An + B$ où A et B sont des constantes, on dit que la complexité de l'algorithme est **linéaire en** n.

On remarque que dans cet exemple, la valeur de a n'influe pas sur la complexité de l'algorithme. De façon générale, il faudra définir ce qu'est la **taille** des données entrantes dont dépendra la complexité. Cela peut être la valeur du paramètre (comme dans le cas de l'exemple 2.1), le nombre de donnée (taille d'un tableau) ou parfois un ensemble de valeurs (nombre de sommets et nombre d'arêtes d'un graphe).

Questions : de quoi dépend la complexité de l'algorithme présenté dans l'exemple 1.3 ?

Problème : le nombre d'opérations ne dépend pas que d'une *taille* mais aussi de la valeur recherchée et des valeurs du tableau.

Pour répondre à ce problème, on pose des hypothèses sur les valeurs de l'input et on étudie les différentes complexités obtenues. On peut ainsi isoler trois cas différents :

- Les données sont dans la meilleure configuration possible : **complexité meilleur des cas**.
- Les données sont dans la pire configuration possible : **complexité dans le pire des cas** (complexité usuelle) ;
- Moyenne sur l'ensemble des configurations : **complexité en moyenne**. Problème : nécessite de définir quel est l'ensemble des configurations ou quelle est sa distribution de probabilité.

Exemple 2.2.
Problème : un tableau T d'entiers contient-il deux entiers égaux ?

```
Input : Un tableau d'entiers T[0] à T[n-1]
Procédé :
1.    Pour i allant de 1 à n-1 :                                  c1
2.        Pour j allant de 0 à i-1 :                              c2
3.            Si T[i] = T[j]                                      c3
4.                Retourner Vrai                                  c4
5.    Retourner Faux                                              c5
Output : Vrai si le tableau contient deux entiers égaux, Faux sinon
```

Complexité meilleur des cas : Les deux premiers entiers de T sont égaux, dans ce cas l'algorithme s'arrête dès le premier teste, les instructions de boucles ne sont exécutées qu'une seule fois. On obtient

$$c_1 + c_2 + c_3 + c_4, \tag{2.2}$$

c'est-à-dire une complexité en **temps constant**.

Complexité dans le pire des cas : T ne contient pas deux entiers égaux, on doit effectuer tous les tests. Pour chaque i, les instructions 2 et 3 sont exécutées i fois, on obtient donc

$$(n-1)c_1 + \sum_{i=1}^{n-1} i(c_2 + c_3) + c_4 = (n-1)c_1 + (c_2 + c_3)\frac{(n-1)n}{2} + c_4 \tag{2.3}$$

$$= \frac{(c_2 + c_3)}{2}n^2 + (c_1 - \frac{c_2}{2} - \frac{c_3}{2})n + c_4 - c_1. \tag{2.4}$$

C'est-à-dire une formule de la forme $An^2 + Bn + C$ avec A, B, C constantes. On dit que la complexité est **quadratique**.

Complexité en moyenne : nécessite de définir l'ensemble des données où sont choisis les valeurs de T puis de calculer la loi de probabilité associée à l'arrêt de l'algorithme. On suppose que le tableau est de taille n et que les valeurs sont prises uniformément et indépendamment dans l'ensemble des entiers de 1 à $k > n$. On calcule la probabilité $p(i)$

qu'un indice i tel que $1 \leq i < n$ soit le premier tel qu'il existe $0 \leq j < i$ avec $T[i] = T[j]$. On trouve

$$p(i) = \frac{k(k-1)\dots(k-i+1)}{k^i}(i-1). \tag{2.5}$$

Par ailleurs, la probabilité qu'on soit dans le pire des cas est donnée par

$$P = \frac{k(k-1)\dots(k-n+1)}{k^n}. \tag{2.6}$$

A partir de là, on peut calculer l'espérance du temps de calcul pour différentes valeurs de k et n. Lorsque $k = n$, la probabilité d'avoir deux valeurs égales dans le tableau est très forte et on trouve une complexité linéaire. Quand $k = n^2$ (donc on pioche dans un ensemble de nombre qui évolue quadratiquement en fonction de la taille), on retrouve une complexité quadratique.

Le calcul de la complexité en moyenne peut s'avérer très complexe même sur des algorithmes simples. On étudiera principalement la **complexité du pire des cas**. Cependant, les complexités en moyenne et du meilleur des cas sont intéressantes à connaître en pratique sur certains algorithmes qui s'avèrent efficaces sur des données réelles.

2.2. **Notation O.** On cherche à comprendre comment évolue le temps de calcul en fonction de taille n de l'input. Pour cette raison, les constantes liées à l'exécutions des commandes élémentaires ne sont pas très significatives dans un premier temps. On utilise la notation mathématique $O(f)$ qui représente la borne supérieure de l'évolution asymptotique de f.

Définition 2.3. *Soit $g(n)$ une fonction, l'ensemble $O(g)$ est l'ensemble des fonctions f telles qu'il existe deux constantes α et n_0 telles que*

$$f(n) \leq \alpha g(n), \tag{2.7}$$

pour tout $n \geq n_0$.

Exemple 2.4.
Soit un algorithme dont le temps d'exécution est donné par $f(n) = An + B$ où A et B sont des constantes. Dans ce cas, on a

$$f(n) \in O(n) \tag{2.8}$$

avec $\alpha = A + B$ et $n_0 = 1$.

De même, si le temps d'exécution est donné par $f(n) = An^2 + Bn + C$, on a $f \in O(n^2)$. De façon générale, si f est un polynôme de degré k, on a que $f \in O(n^k)$.

2.3. **Fonctions de référence.** Lorsque l'on pose la question "Donnez la complexité de cet algorithme", on cherche à trouver la fonction g la plus "rapide" telle que $f \in O(g)$. Voici les fonctions de référence auxquels on se rapportera le plus souvent.

Fonction	Désignation	Exemple d'algorithme
1	temps constant	lire une valeur indicée dans un tableau
$\log(n)$	logarithmique	Dichotomie
$\sqrt{(n)}$	racinaire	test simple de primalité
n	linéaire	parcours d'un tableau
$nlog(n)$	quasi-linéaire	tri rapide (en moyenne), tri par tas
n^a	polynomiale	tri bulle, tri insertion
a^n	exponentielle	Tour de Hanoi
$n!$	factorielle	problème du voyageur de commerce