Ch 2 : complexité

I Notion de complexité d'un algorithme

I.1 Description du problème

Lorsqu'on exécute un programme sur un ensemble de donnée d'entrée, il va mettre un certain temps d'exécution, plus ou moins long, mais généralement croissant avec la taille de ces données.

Ce temps d'exécution peut être testé expérimentalement, par exemple en utilisant la fonction `time.time()` du package `time`. Cette approche a toutefois des défauts : elle est notamment dépendante du hardware et software utilisé. Par ailleurs, des données de même taille ne mènent pas nécessairement au même temps d'exécution.

On peut également partir d'une approche plus théorique : partir d'une description de haut-niveau de l'algorithme, et décompter le nombre d'opérations élémentaires effectuées selon la taille de la donnée d'entrée.

Il faut pour cela définir la taille de la donnée d'entrée : par exemple le nombre de chiffres n d'un entier, le nombre d'éléments n d'un tableau, le nombre de lignes n d'une matrice...

Il faut aussi définir ce qu'est une opération élémentaire, par exemple :

1. La somme de deux variables de type *int* ou *float*
2. L'affectation d'une valeur à une variable
3. La lecture d'une valeur en mémoire
4. Le test d'égalité de deux valeurs...

Il n'est pas facile de définir la liste de ces opérations, ni évident que toutes ces opérations ont le même temps d'exécution. C'est pourtant l'hypothèse qui sera généralement faite en pratique.

I.2 Un premier exemple

On donne ci-dessous un algorithme de recherche d'un maximum dans une tableau de nombres flottants T à l'aide de la méthode du candidat.

```
def maximum(tab):
        maxi = tab[0]
        for terme in tab[1:]:
                if terme > maxi:
                        maxi = terme
        return(maxi)
```

Décomptons le nombre d'opérations élémentaires (pour chaque ligne où une telle opération est effectuée) dans le tableau ci-dessous.

Ligne	Accès tableau	Aff. Entier	Aff. flottant	Comp. flottants
1				
2				
3				
4				
Total				

I.3 Rappels d'analyse asymptotique

On rappelle que si f et g sont des fonctions définies sur $\mathbb{N}$

1. $f = O(g)$ si et seulement si il existe un réel positif C et un entier n_0 tel que pour tout $n \geq n_0$, $|f(n)| \leq C|g(n)|$
2. $f = o(g)$ si et seulement si pour tout réel $\epsilon > 0$, il existe un entier n_0 tel que pour tout $n \geq n_0$, $|f(n)| \leq \epsilon|g(n)|$.
3. $f \sim g$ si et seulement si $|f - g| = o(f)$.

Dans le cas ou f et g sont non nulles à partir d'un certain rang, la dernière conditions équivaut à $\lim\limits_{+\infty} \frac{f}{g} = 1$.

Par exemple, si on note f le nombre d'opérations effectuées par l'algorithme de la section I.2, $f = O(n)$. On dit dans ce cas que le temps d'exécution est linéaire.

En règle générale, on se contentera de donner une estimation asymptotique de la complexité dans le pire des cas en dominant le nombre d'opération par une fonction usuelle. On donne ci-dessus les types de complexités les plus communes.

Constante	$O(1)$
Linéaire	$O(n)$
$n\log\ n$	$O(n\ln(n))$
Quadratique	$O(n^2)$
Cubique	$O(n^3)$
Exponentielle	$O(2^n)$
Factorielle	$O(n!)$

Face à un problème donné, il est essentiel de trouver un algorithme le plus performant possible, c'est-à-dire asymptotiquement dominé par une fonction à la croissance la plus lente possible.

II Comparaison d'algorithmes de tri

II.1 Un premier exemple : le tri bulle

On rappelle ici un des plus naïfs d'entre eux, l'algorithme de tri bulle, qui prend en entrée un tableau T.

```
def tri_bulle(T):
        for i in range(len(T)-1):
                for j in range(len(T)-1-i):
                        if T[j]>T[j+1]:
                                T[j],T[j+1]=T[j+1],T[j]
```

II.2 Un tri plus performant : le tri fusion

II.2.1 Fusion de deux tableaux triées

On rappelle que principe de l'algorithme de tri fusion est le suivant : prenons un tableau à trier T de taille n. Si $n = 0$ ou $n = 1$, le tableau est déjà trié. Sinon, on partage le tableau en deux sous-tableaux de tailles environ égales. On trie chaque sous-tableau en appelant récursivement l'algorithme, puis on fusionne les deux sous-tableaux obtenus. Il s'agit d'un algorithme basé sur l'approche *diviser-pour-régner*.

Avant de décrire l'algorithme complet, nous avons besoin d'un algorithme de fusion de deux tableaux. Cet algorithme prend en paramètres deux tableaux de nombres **déjà triées** T_1 et T_2 et retourne une tableau T contenant les éléments de T_1 et T_2 trié dans l'ordre.

L'idée de l'algorithme est la suivante : on part d'une tableau T vide. on compare les premiers éléments de T_1 et T_2. On ajoute à T le plus petit de ces deux éléments, on passe à l'élément suivant dans la tableau correspondant et on itère le processus jusqu'à avoir épuisé tous les éléments d'une des tableaux de départ. On termine en ajoutant tous les éléments restant du tableau dont les éléments n'ont pas été épuisés.

```
def Fusion(L1,L2):
    L=[]
    while L1 and L2:
        if L1[0]<L2[0]:
            L.append(L1[0])
            L1.pop(0)
        else:
            L.append(L2[0])
            L2.pop(0)
    if L1:
        L=L+L1
    else:
        L=L+L2
    return(L)
```

On peut remarque que si $n = len(T_1) + len(T_2)$, l'algorithme effectue au maximum n tests.

II.2.2 Le tri fusion

On peut maintenant écrire l'algorithme *TriFusion* de manière récursive.

```
def TriFusion(L):
        n=len(L)
        if n<2:
                return(L)
        else:
                m=n//2
                return(Fusion(TriFusion(L[:m]),TriFusion(L[m:])))
```

La figure 1 détaille l'ensemble des appels à l'algorithme *TriFusion* pour trier le tableau $T = [9, 5, 7, 3, 4, 1, 0, 2, 8, 6]$

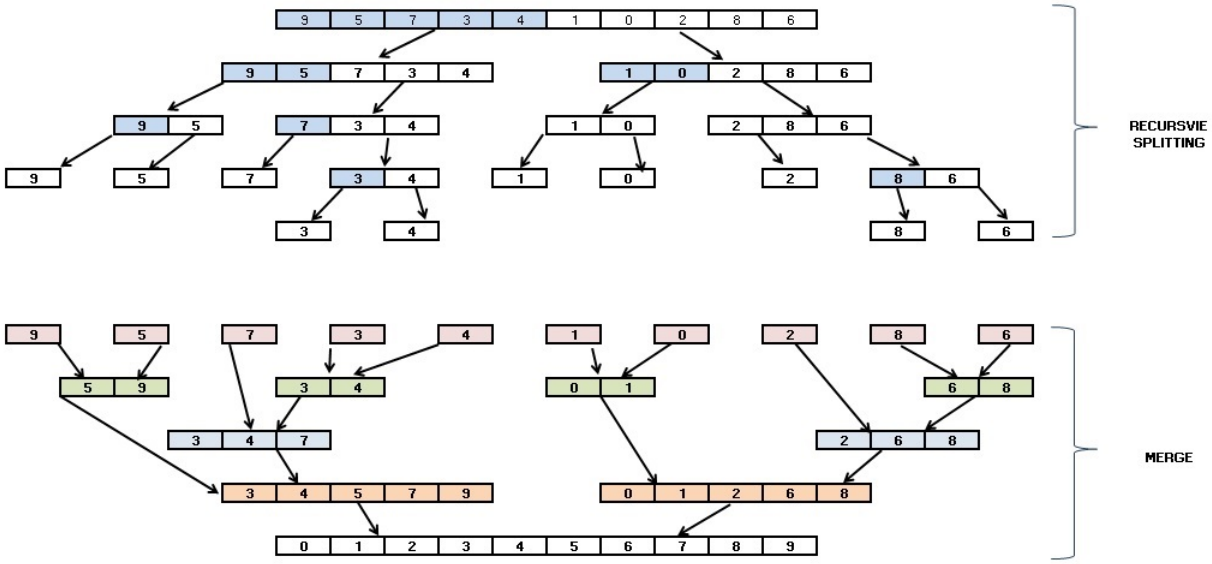

FIGURE 1 – Source : pythonandr.com