Complexité algorithmique

Badraddine AGHOUTANE
b.aghoutane@umi.ac.ma

Critère d'évaluation de complexité

L'évaluation de la **complexité d'un algorithme** se fait par l'analyse relative de deux **ressources de l'ordinateur** :

- Le **temps de calcul**.
- L'**espace mémoire**, utilisé par un programme, pour transformer les données du problème en un ensemble de résultats.

- L'**analyse de la complexité** consiste à mesurer ces deux grandeurs pour choisir l'**algorithme le mieux adapté** pour résoudre un problème :

  ➔ L'**algorithme le plus rapide et le moins gourmant en consommation mémoire**

Définition de la complexité algorithmique

- La **complexité d'un algorithme** est une mesure du **temps requis par l'algorithme** pour accomplir sa tâche, en fonction de la **taille de l'échantillon** (les données) à traiter.
- Souvent, comme dans le cas où un algorithme manipule un **tableau de n éléments** (*on dira un tableau de taille n*), la complexité sera notée en fonction de cette taille.
- Par exemple, un algorithme de **complexité O(2n+8)** prendra dans **le pire cas** 8 opérations, plus 2 opérations supplémentaires par élément du tableau.

Calculer la complexité d'un algorithme

- Pour calculer la **complexité d'un algorithme** donné, il convient tout d'abord de compter le nombre d'opérations impliquées par son exécution.
- Notation O :
  - La notation la plus utilisée pour noter la complexité d'un algorithme est la **notation O (Ordre de...)**, qui dénote un ordre de grandeur.
  - Par exemple, on dira d'un algorithme qu'il est **O(15)**, s'il nécessite au **maximum 15 opérations** (dans le pire cas) pour se compléter.

Catégorie de complexité d'un algorithme

Il existe plusieurs **catégories de complexité** à savoir:

- La complexité constante.
- La complexité logarithmique.
- La complexité linéaire.
- La complexité quadratique.
- La complexité cubique.
- La complexité exponentielle.
- La complexité factorielle.
- …

Algorithmes de complexité constante

Commençons par un exemple simple, soit une fonction prenant en paramètre le rayon d'une sphère, calculant le volume de cette sphère, et retournant cette valeur au sous-programme appelant. On aura le code suivant:

- On peut compter l'instruction notée **(0)** comme étant une opération (certains l'omettront)
- On peut la compter l'instruction notée **(1)** comme **cinq opérations ou comme une seule**
- L'instruction notée **(2)**, du retour de la valeur résultante de l'exécution de la fonction, est aussi une opération.

```
Fonction volume_sphere(rayon : Réel) : Réel
Constante PI = 3,14      // (0)
Variable volume : Réel
Début
        volume ← 4 / 3 * PI * rayon ^ 3; // (1)
        retourne(volume);   // (2)
FinFonction
```

➔ **L'algorithme sera donc O(2) ou O(8), tout dépendant de la manière de compter les opérations.**

Algorithmes de complexité constante

- Le plus important ici n'est pas la valeur exacte entre les parenthèses suivant le O (*O(2) ou O(8), ...*), mais le fait que cette valeur soit **constante**.
- Lorsqu'un algorithme est O(c) où « c » est une constante, on dit qu'il s'agit alors d'un **algorithme en temps constant.**
- Une **complexité constante** est la complexité algorithmique idéale, puisque **peu importe la taille de l'échantillon à traiter**, l'algorithme prendra toujours un nombre fixé à l'avance d'opérations pour réaliser sa tâche.
- Tous les **algorithmes en temps constant** font partie d'une classe nommée **O(1)**. En général, qu'un algorithme soit **O(3)**, **O(17)** ou **O(100000)**, on dira de lui qu'il est en fait O(1) puisque la différence de performance entre deux algorithmes en temps constant peut être satisfaite par un simple remplacement matériel (*utiliser un processeur plus rapide, par exemple*).

Algorithmes de complexité logarithmique

Les algorithmes de **complexité logarithmique** ont la propriété suivante:

- Pour chaque itération (boucle), l'algorithme va réduire (*de moitié par exemple*) la partie des données à traiter (*comme l'exemple d'une recherche dichotomique*).

```
Variable N : Entier
Fonction trouver_indice(tableau tab[N] : Entier, val : Entier) : Entier
Variable i, compteur : Entier
Début
                compteur = 0;                                   (1)
    TantQue (compteur < N && tab[compteur] != val) faire
                    compteur ← compteur +1;
     FinTantQue
                si (compteur < N) Alors                         (2)
                   indice = compteur;                           (3)
                Finsi
                retourne indice;                                (4)
FinFonction
```

Algorithmes de complexité logarithmique

On en compte ici quatre (4) opérations :

1. **« avant la boucle »** – l'initialisation de la variable compteur,
2. **« après la boucle »** la comparaison de compteur avec N,
3. **l'affectation** d'une valeur à la variable indice,
4. la **production** de la valeur de la fonction (retourne)

En plus, pour chaque itération de la **boucle Tantque**, on a *deux comparaisons et une incrémentation de compteur.*

```
Variable N : Entier
Fonction trouver_indice(tableau tab[N] : Entier, val : Entier) : Entier
Variable i, compteur : Entier
Début
                compteur = 0;    (1)
    TantQue (compteur < N && tab[compteur] != val) faire
                    compteur ← compteur +1;
     FinTantQue
                si (compteur < N) Alors                    (2)
                   indice = compteur;                      (3)
                Finsi
                retourne indice;                           (4)
FinFonction
```

Algorithmes de complexité logarithmique

- **Dans le cas moyen**, puisqu'en présumant que la fonction pourra être appelée de manière uniforme pour des éléments parfois près du début de tab[], et parfois près de la fin de tab[], il faudra en moyenne 3*(N/2)+4 opérations pour produire la solution.
- Cela signifie qu'en **moyenne**, cet algorithme nécessitera :
  - **19**, donc 3×(10/2)+4, opérations pour un tableau de **10 éléments**
  - **154**, donc 3×(100/2)+4, opérations pour un tableau de **100 éléments**
  - **1504**, donc 3×(1000/2)+4, opérations pour un tableau de **1000 éléments**

```
Variable N : Entier
Fonction trouver_indice(tableau tab[N] : Entier, val : Entier) : Entier
Variable i, compteur : Entier
Début
                compteur = 0;
                                                  (1)
    TantQue (compteur < N && tab[compteur] != val) faire
                    compteur ← compteur +1;
     FinTantQue
                si (compteur < N) Alors                    (2)
                   indice = compteur;                      (3)
                Finsi
                retourne indice;                           (4)
FinFonction
```

Algorithmes de complexité logarithmique

- **Dans le pire cas** (*si val se trouve à l'élément N-1 de tab[], ou si cette valeur n'est pas du tout présente dans tab[]*), il nous faut par contre **« 3*N+4 »** opérations pour produire la solution *(pour N itérations: 3*N, s'ajoutent les 4 opérations de base*).
- Notez toutefois que dans **le pire cas**, cet algorithme nécessitera :
  - **34**, donc 3×10+4, opérations pour un tableau de **10 éléments**
  - **304**, donc 3×100+4, opérations pour un tableau de **100 éléments**
  - **3004**, donc 3×1000+4, opérations pour un tableau de **1000 éléments**

```
Variable N : Entier
Fonction trouver_indice(tableau tab[N] : Entier, val : Entier) : Entier
Variable i, compteur : Entier
Début
                compteur = 0;                                   (1)
    TantQue (compteur < N && tab[compteur] != val) faire
                compteur ← compteur +1;
    FinTantQue
                si (compteur < N) Alors                         (2)
                   indice = compteur;                           (3)
                Finsi
                retourne indice;                                (4)
FinFonction
```

Algorithmes de complexité logarithmique

- **Dans le meilleur cas** (si val se trouve à l'élément 0 de tab[]), il nous faut seulement (2+4) opérations pour produire la solution, où le « 2 » constitue les deux conditions à évaluer au début de la boucle, et le « 4 » est le nombre d'opérations incontournables à faire.

```
Variable N : Entier
Fonction trouver_indice(tableau tab[N] : Entier, val : Entier) : Entier
Variable i, compteur : Entier
Début
                compteur = 0;                                   (1)
    TantQue (compteur < N && tab[compteur] != val) faire
                compteur ← compteur +1;
    FinTantQue
                si (compteur < N) Alors                         (2)
                   indice = compteur;                           (3)
                Finsi
                retourne indice;                                (4)
FinFonction
```

Algorithmes de complexité linéaire

- Par complexité linéaire, ou O(n), on dénotera des algorithmes pour lesquels le nombre d'étapes à effectuer changera en proportion directe de la taille de l'échantillon à traiter : si l'échantillon croît par un facteur de 100, la complexité sera étendue, elle augmentera aussi par un facteur de 100.
- Un algorithme qui fait la somme des éléments d'un tableau : **O(2 + n * 4)**

```
Fonction somme_elemts(tableau tab[N] : Entier) : Entier
Variable somme, i : Entier
Début
                    somme ← 0;
                    Pour i←0 à N-1 [par 1] faire
                        somme ← somme + tab[i];
                    FinPour
                    retourne somme;
FinFonction
```

Algorithmes de complexité quadratique

- On dira d'un algorithme de complexité $O(n^2)$ qu'il est de complexité quadratique.
- Notons qu'il existe plusieurs autres niveaux de complexité (par exemple, des complexités comme $O(n^3)$, $O(n^4)$, ou des complexités mettant en relation plusieurs variables comme $O(n+m^3)$.
- Il faut retenir ici est qu'il est possible de calculer la complexité d'un algorithme, et de comparer la complexité relative de deux algorithmes pour choisir le plus efficace.

- **Tri à bulle:**

```
Procédure permuter(a,b : Entier)
Variable temp : Entier
Début
                    temp ← a;
                    a ← b;
                    b ← temp;
FinProcédure
```

Algorithmes de complexité quadratique

L'algorithme de tri à bulles a une complexité en temps en $O(N^2)$ en pire cas, où N est la taille du tableau. Le pire cas correspond ici au cas où le tableau est 10 initialement trié par ordre décroissant : dans ce cas l'algorithme doit faire remonter chaque élément jusqu'à la i-ème place à chaque étape i, en exécutant à chaque fois un échange.
La complexité en temps indique ici que le temps pris par l'algorithme pour trier un tableau de taille N augmente quadratiquement en fonction de N lorsque N est très grand.

```
Procédure tri_bulle (tab[N] : Entier)
Variable i, j : Entier
Début
                    Pour i←0 à N-1 faire
                        Pour j←i+1; à N-1 faire
                            si(tab[i]>tab[j]) alors
                                permuter(tab[i],tab[j]);
                            finsi
                        FinPour
                    FinPour
FinProcédure
```