INFO2009 - Introduction à l'informatique

Complexité

Bertrand Alexis

Université de Liège

2025

Complexité

La complexité théorique en temps et en espace permet de caractériser le coût d'un algorithme en fonction de la taille de ses données.
Elle permet d'expliquer pourquoi et comment le temps d'exécution ou la mémoire utilisée varient lorsque la valeur des variables non-fixes change.
L'objectif n'est pas de mesurer exactement le temps ou la mémoire réels, mais de comprendre la manière dont ils varient asymptotiquement.

Notation

Pour exprimer la complexité, nous utilisons la notation de *Landau* (notation *Big-o*) qui regarde la croissance tout en ignorant les constantes et les termes de plus petit ordre.. Réaliser $1 \times n$ instructions ou $2 \times n$ instructions ne change rien. Nous approximerons donc chaque fois les constantes à 1. Si nous avons une complexité $N^2 + N$, on ne tiendra compte que du plus grand, donc N^2 dans ce cas.

Complexité en temps itérative

Pour calculer la complexité en temps, on doit regarder la complexité pour chaque instruction. Si jamais nous entrons dans une boucle, on doit regarder le nombre maximum qu'on va itérer dans cette boucle. Toutes les instructions à l'intérieur seront donc multipliées par le nombre d'itérations maximum. Enfin, si on utilise d'autres fonctions, on devra également prendre la complexité en temps de cette fonction en compte. On finit par approximer la somme des complexités.

Exemples

```
for (int i = 0; i < n; i++) {
    ...
    for (int j = 0; j < i; j++) {
        ...
    }
}
```

Cette suite d'instruction aurait une complexité théorique en temps de $\mathcal{O}(n^2)$ car la première boucle aura au maximum n itérations et la deuxième aura au maximum i itérations. Comme i prendra dans le pire des cas la valeur $n-1$ (approximé par n), la deuxième boucle aura un maximum de n itérations. De plus, comme la deuxième boucle est dans la première, la complexité vaut donc $n \times n$.

Exemples

```
for (int i = 0; i < n; i++) {
    ...
}
for (int j = 0; j < n; j++) {
    ...
}
```

Cette suite d'instruction aurait une complexité théorique en temps de $\mathcal{O}(n)$ car la deuxième boucle n'est pas dans la première. La complexité vaut donc $n + n = 2n$ ce qu'on approxime par n.

Complexité en espace itérative

Pour la complexité en espace, on va regarder le nombre de variables qu'on va créer. Une variable "normale" compte pour 1 et un tableau a une complexité en fonction de sa taille. Si jamais on a une boucle on va multipler le nombre de créations de variables par le nombre d'itérations. Enfin, si on appelle une fonction on doit prendre en compte la complexité en espace de cette dernière. On finit par approximer la somme des complexités en espace.
N.B. On ne considère que les varaibles créées et espaces mémoires alloués dans la fonction. Si on a des variables ou des tableaux en argument de la fonction, ceux-ci ne sont pas pris e considération pour la complexité en espace.

Exemples

```
void f (int *t, int taille) {
    ...
}
```

Cette fonction, si on part de l'hypothèse qu'aucune autre variables n'a été créée à l'intérieur, a une complexité théorique en espace de $\mathcal{O}(1)$.

```
t = malloc(n * sizeof(int));
...
```

Cette suite d'instruction a une complexité en espace de $\mathcal{O}(n)$ car nous venons de créer un tableau de taille n.

Complexité en temps récursive

La facon de faire est similaire à celle des fonctions itératives sauf qu'on doit multiplier la complexité en temps de la fonction (sans compter son appel à elle-même) par le nombre d'appels à elle-même, ce qui correspond au nombre de contextes mis sur la pile simultanément. Enfin, on finit par approximer le résultat.

Complexité en espace récursive

C'est de nouveau la même chose. On multiplie la complexité en espace de la fonction par le nombre de contextes mis sur la pile simultanément. Enfin, on finit par approximer le résultat.