Chapitre 3 : Analyse de Programmes

1er octobre 2023

On a vu qu'un algorithme correspond à une séquence d'instructions à exécuter pour répondre de façon automatique en temps fini à un problème, qui a une spécification donnée, c'est-à-dire des entrées possibles vérifiant certaines contraintes, et des sorties dont la valeur dépend de l'entrée.

Pour un algorithme donné, plusieurs questions peuvent se poser :

- Est-on sûr que l'algorithme termine ?
- Est-on sûr que l'algorithme est correct, c'est-à-dire, que la sortie calculée par la séquence d'instruction est bien celle attendue pour l'entrée donnée selon la spécification du problème ?
- Combien de temps et combien de ressources matérielles vont être nécessaires pour l'exécution de l'algorithme sur une entrée donnée ?

Nous allons voir dans ce chapitre des méthodes qui permettent, par des raisonnements, d'observer les propriétés des algorithmes, concernant leur **terminaison**, leur **correction** et leur **complexité**.

1 Terminaison et Correction

1.1 Terminaison

On a vu qu'il était possible, dans un algorithme, de se retrouver dans une situation où des calculs sont effectués à l'infini, par exemple lors de l'utilisation d'une boucle dont la condition ne cesse jamais d'être vérifiée.

Définition

On dit qu'un algorithme **termine** lorsque pour toute entrée de l'algorithme, le nombre d'étapes de calcul en fonction de l'entrée est fini.

Exemple. Le programme ci-dessous ne termine pas, car il utilise une boucle infinie :

```
#include <stdio.h>

int main(){

    int x = 1;
    while (x>0){
        printf("%d \n",x);
        x++;
    }
}
```

Exemple. Le programme ci-dessous termine, car la condition de la boucle cesse d'être vérifiée au bout de 20 itérations :

```
#include <stdio.h>

int main(){

    int x = 1;
    while (x<=20){
        printf("%d \n",x);
        x++;
    }
}
```

Remarque. Il est possible de ne pas savoir si un programme termine : la conjecture de Syracuse, qui affirme que toute itération de la suite de Syracuse à partir d'un entier donné finit par arriver sur le nombre 1, n'a jamais été démontrée. Par conséquent, si l'on écrit un algorithme qui calcule les termes de la suite de Syracuse et s'arrête lorsque l'un d'entre eux est égal à 1, on ne sait pas si cet algorithme termine pour toute entrée.

Parmi les outils de programmation qui ont été vus pour l'instant, le seul permettant d'effectuer un nombre d'opérations non-borné (et donc potentiellement infini) est la boucle. Pour prouver qu'un algorithme contenant des boucles termine, il faut donc prouver que les boucles présentes dans l'algorithme terminent. Pour cela, on utilise la technique du variant :

Définition

Soit un algorithme contenant une boucle. Un **variant** est une valeur s'exprimant en fonction des variables de l'algorithme qui :

- ne prend que des valeurs entières positives ou nulles lors d'une exécution de la boucle ;
- décroît strictement à chaque tour de boucle.

Théorème

Si une boucle admet un variant, alors cette boucle termine.

Cela peut servir d'outil de preuve pour la terminaison d'un algorithme. Si l'on reprend le second programme, on considère la quantité $y = 20 - x$:

- La condition `x <= 20` est équivalente au fait que le variant $y = 20 - x$ soit positif ou nul, et y est entier ;
- x est incrémenté à chaque itération de la boucle : x croît, donc y décroît.

Attention

Il faut bien que le variant soit entier pour assurer la terminaison : c'est parce qu'il est entier et décroissant que sa valeur baisse d'au moins une unité à chaque itération de la boucle, et qu'il y a au plus n itérations si la valeur initiale du variant est n.
Si le variant n'est pas entier, il peut décroître une infinité de fois sans devenir inférieur à 0 (exemple : suite des $\frac{1}{n}$).

Exemple. Algorithme d'exponentiation naïve :

```
int exp_naive(int a, int p)
    {
    int r = 1;
    for (int i = 0; i < p; i++){
        r = r*a;
    }
    return r;
    }
```

1.1.1 Problème de l'Arrêt

On sait que lorsqu'un programme écrit en C n'est pas syntaxiquement correct, le compilateur renvoie un message d'erreur : un programme est utilisé pour analyser l'écriture du programme initial et déterminer de façon systématique s'il est syntaxiquement correct ou non. On peut se demander si un tel programme pourrait exister pour déterminer de façon systématique, en prenant en entrée un programme écrit dans un langage donné, si son exécution va terminer :

Définition

Le **Problème de l'Arrêt** est un problème de décision dont la spécification est la suivante :

— **Entrée :** Code source d'un programme dans un langage donné

— **Sortie :** *Vrai* si le programme termine, *Faux* sinon.

Le théorème suivant répond par la négative à la question posée :

Théorème

Le problème de l'Arrêt est **indécidable** : il n'existe pas d'algorithme capable de déterminer, pour un code source de programme donné, si le programme termine ou non.

Cela signifie que les analyses de terminaison de programme doivent se faire au cas par cas, et lorsque cela est possible.

1.2 Correction

On a vu dans la sous-section précédente qu'il existe des méthodes pour déterminer, dans certains cas, si un algorithme termine. On peut également se demander, lorsque l'on a un algorithme, s'il répond bien à un problème donné ; autrement dit, si les sorties calculées par l'algorithme en fonction des entrées correspondent bien à la spécification du problème associé.

On a vu qu'un algorithme peut ne pas terminer. On peut définir plusieurs types de correction d'algorithme en fonction de leur terminaison :

Définition

On dit qu'un algorithme est **partiellement correct** s'il correspond bien à sa spécification pour les entrées sur lesquelles il termine.
On dit qu'un algorithme est **totalement correct** s'il est partiellement correct et s'il termine pour toute entrée.

Exemple. Un algorithme qui prend en entrée un entier naturel non-nul et calcule l'ensemble des successeurs dans la suite de Syracuse jusqu'à la première occurrence de 1 sera partiellement correct, car pour toute entrée où il termine il correspondra à la spécification du problème auquel il doit répondre, mais on ne sait pas s'il termine sur toute entrée.

De la même façon que l'on a des méthodes (variant) pour s'assurer de la terminaison de certains programmes, il existe des méthodes pour s'assurer de leur correction :

Définition

Soit un algorithme contenant une boucle. Un **invariant de boucle** est une propriété qui :

— est vraie avant le début de la boucle ;

— si elle est vraie au début d'un tour de boucle, est vraie également à la fin.

Les invariants de boucle permettent de s'assurer de la véracité de certaines proposition à la fin de l'exécution d'un programme :

Théorème

Soit un algorithme comprenant une boucle, et un invariant de cette boucle. Alors pour toute entrée pour laquelle l'algorithme termine, l'invariant est vérifié à la fin de l'exécution de la boucle.

Exemple. Dans l'algorithme d'exponentiation naïve, on a $r = a^i$ à chaque étape de calcul ; c'est un vinariant de la boucle `for`.

2 Complexité

On a vu que pour un problème donné, il pouvait exister plusieurs algorithmes (dont on peut prouver la correction et la terminaison). En pratique, lorsqu'ils sont implémentés, ces algorithmes peuvent avoir un comportement différent dans la façon dont ils occupent de l'espace en mémoire, où dans le temps qu'ils prennent pour s'exécuter.

On peut donc chercher à mesurer la **performance** de ces algorithmes, en mesurant :

— Leur **complexité temporelle**
— Leur **complexité spatiale**

2.1 Complexité Temporelle : Définition

La complexité temporelle (ou *coût*) est définie de la façon suivante :

Définition

Pour un programme et une entrée donnée, la **complexité temporelle** est le nombre d'opérations élémentaires réalisées pendant l'exécution du programme sur l'entrée.

Les opérations élémentaires (ou opérations atomiques) sont les opérations correspondant à une unique instruction assembleur. Les opérations considérées comme élémentaires sont les suivantes :

— Accès mémoire pour la lecture ou l'écriture d'une valeur de variable (ou de case de tableau)
— Opération d'addition, soustraction, multiplication, et opérations de comparaisons sur des valeurs numériques.

Pour une entrée e d'un programme, on peut noter $Op(e)$ ce nombre d'opérations. On veut en particulier étudier la façon dont évolue la complexité d'un algorithme en fonction de la taille de l'entrée. Pour ça, on peut définir la complexité dans la pire cas et dans le cas moyen :

Définition

Soit un programme, soit O_p la fonction associant à chaque entrée e du programme la complexité temporelle de celui-ci sur cette entrée. Soit $n \in \mathbb{N}$ un entier naturel et soit E_n l'ensemble des entrées de taille n.

— La **complexité dans le pire cas** en fonction de n est la complexité temporelle la plus élevée pour une entrée de taille n :

$$C(n) = \max_{e \in E_n} Op(e)$$

— La **complexité en moyenne** en fonction de n est la moyenne des complexités temporelle de toutes les entrées de taille n.

La définition de la taille d'une entrée pourra dépendre du problème étudié : longueur d'une chaîne de caractères, longueur d'une liste, nombre de sommets et d'arêtes dans un graphe...

On peut également avoir plusieurs entrées et calculer la complexité en fonction de la taille de chacune des entrées.

Remarque. On peut également (en remplaçant max par min) définir une complexité dans le meilleur cas.

Remarque. Pour la complexité en moyenne, elle peut être définie en fonction de la probabilité p_e de tomber sur chacune des entrées e : $C_m(n) = \sum_{e \in E_n} p_e \times Op(e)$.

On considérera le plus souvent une répartition uniforme discrète : $p_e = \frac{1}{Card(E_n)}$ pour toute entrée e, d'où $C_m(n) = \frac{1}{Card(E_n)} \times \sum_{e \in E_n} Op(e)$.

Remarque. La complexité en moyenne ne peut pas toujours s'exprimer, notamment lorsqu'il existe une infinité de cas possibles. On peut pour palier à ce problème projeter l'ensemble des entrées dans un ensemble fini, ou émettre d'autres hypothèses sur les entrées possibles du problème.

Exemple. On considère un algorithme, qui prend deux listes de longueurs N_1 et n_2 en entrées et vérifie l'existence d'un élément en commun entre les deux listes, à l'aide d'une boucle parcourant les éléments de la deuxième liste, à l'intérieur d'une boucle parcourant les éléments de la première liste.

Dans le pire cas, s'il n'y a pas d'élément commun ou s'il est à la toute fin des deux listes, on fait N_1 itérations de la boucle principale, et à chaque itération on fait N_2 itérations de la boucle intérieure, dans laquelle on fait une opération de comparaison à chaque fois : on a un total de $N_1 \times N_2$ opérations. Dans le meilleur cas, l'élément commun est au début de chaque liste et une seule opération est faite.

Le cas moyen dépend de l'ensemble dans lequel peuvent se situer les éléments de chaque liste, et de leur probabilité d'apparition ; s'il y a un très grand nombre d'éléments possibles par rapport à la taille des listes, il y a peu de chances d'avoir un élément commun et le pire cas aura une forte probabilité d'arriver.

2.2 Notations de Landau, Complexité Asymptotique

En pratique, on s'intéressera, plutôt qu'à des valeurs exactes, à des ordres de grandeur : en pratique, on aura un intérêt à utiliser un algorithme sur des données de grandes tailles, et on souhaite savoir comment se comporte la complexité temporelle de l'algorithme lorsque ces entrées sont de grande taille - ou, dans un vocabulaire mathématique, lorsque leur taille tend vers $+\infty$. On parle de **complexité asymptotique**.

Pour définir la complexité asymptotique, on aura besoin de définir rigoureusement les ordres de grandeur. Pour cela, on utilisera les notations de Landau :

Définition

Soient f et g deux fonctions de $\mathbb{N}$ dans $\mathbb{N}$.

— On dit que g est **majorée par** f à un facteur constant près s'il existe un facteur $k \in \mathbb{R}_+^*$ et un rang $n_0 \in \mathbb{N}$ tels que, pour tout entier $n \geq n_0$, on a $g(n) \leq kf(n)$. L'ensemble des fonctions majorées par f à un facteur constant près se note $O(f(n))$, et on note :

$$g(n) = \mathcal{O}(f(n)) \text{ ou } g(n) \in \mathcal{O}(f(n))$$

— On dit que g est **minorée par** f à un facteur constant près s'il existe un facteur $k \in \mathbb{R}_+^*$ et un rang $n_0 \in \mathbb{N}$ tels que, pour tout entier $n \geq n_0$, on a $kf(n) \leq g(n)$. L'ensemble des fonctions minorées par f à un facteur constant près se note $\Omega(f(n))$, et on note :

$$g(n) = \Omega(f(n)) \text{ ou } g(n) \in \Omega(f(n))$$

— On dit que g est **de même ordre de grandeur que** f si g est à la fois majorée par f à un facteur constant près et minorée par f à un autre facteur constant près. L'ensemble des fonctions de même ordre de grandeur que f se note $\Theta(f(n))$, et on note :

$$g(n) = \Theta(f(n)) \text{ ou } g(n) \in \Theta(f(n))$$

— On dit que g est **équivalente à** f si $\lim_{n\to+\infty} \frac{g(n)}{f(n)} = 1$ (en supposant que f ne s'annule pas). On note :

$$g(n) \sim f(n)$$

On s'intéressera **très principalement** à la notation O, pour majorer les complexités asymptotiques d'algorithmes.

Remarque. On a $g(n) \in \mathcal{O}(f(n)) \Leftrightarrow f(n) \in \Omega(g(n))$: g est majorée par f si et seulement si f est minorée par g.

Notations

La notation avec le symbole d'égalité, $g(n) = \mathcal{O}(f(n))$, est fréquemment utilisée. Elle peut s'interpréter comme "g est une fonction majorée par f" ; néanmoins, ce n'est pas une vraie relation d'égalité, car deux fonctions g_1 et g_2 différentes peuvents être toutes les deux majorées par f.
On peut également dire que $\mathcal{O}(f(n))$ désigne *l'ensemble* des fonctions majorées par f, et utiliser la notation $g(n) \in \mathcal{O}(f(n))$ (pareil pour $\Omega(f(n))$ et $\Theta(f(n))$). Néanmoins, la notation avec l'égalité est la notation conventionnelle en informatique et sera celle que l'on utilisera par la suite.

Exemple. Soit un algorithme dont la complexité dans le pire cas est $C(n) = 4n^3+3n^2+2n+1$. On a :

1. $C(n) = \mathcal{O}(n^3)$ (facteur $k = 5$)

2. $C(n) = \Omega(n^3)$ (facteur $k = 1$)
3. $C(n) = \Theta(n^3)$ (conséquence des deux affirmations précédentes)
4. $C(n) \sim 4n^3$

Exemple. Cf TD.

2.3 Règles de Calcul, Complexités Usuelles

2.3.1 Propriétés

On a les propriétés suivantes sur les relations $g(n) = \mathcal{O}(f(n))$, $g(n) = \Omega(f(n))$ et $g(n) = \Theta(f(n))$:

Proposition

La relation $g(n) = \mathcal{O}(f(n))$ est :

— **réflexive** : pour toute fonction f de $\mathbb{N}$ dans $\mathbb{N}$, $f(n) = \mathcal{O}(f(n))$

— **transitive** : pour toutes fonctions f, g et h de $\mathbb{N}$ dans $\mathbb{N}$, si $g(n) = \mathcal{O}(f(n))$ et $h(n) = \mathcal{O}(g(n))$ alors $h(n) = \mathcal{O}(f(n))$

Proposition

La relation $g(n) = \Omega(f(n))$ est réflexive et transitive.

Proposition

— La relation $g(n) = \Theta(f(n))$ est réflexive et transitive.

— La relation $g(n) = \Theta(f(n))$ est **symétrique** : pour toutes fonctions f et g de $\mathbb{N}$ dans $\mathbb{N}$, si $g(n) = \Theta(f(n))$ alors $f(n) = \Theta(g(n))$.

Preuve : Exercice (TD)

Proposition

— Si $C(n) = \Theta(f(n))$ alors $kC(n) = \Theta(f(n))$ pour $k > 0$.

— Si $C(n) = \Theta(f(n))$ et $D(n) = \Theta(f(n))$ alors $C(n) + D(n) = \Theta(f(n))$.

— Si $C(n) = \Theta(f(n))$ et $D(n) = \Theta(g(n))$ alors $C(n)D(n) = \Theta(f(n)g(n))$.

— Si $C(n) = \mathcal{O}(f(n))$ et $D(n) = \mathcal{O}(f(n))$ alors $C(n) + D(n) = \mathcal{O}(f(n))$.

— Si $C(n) = \mathcal{O}(f(n))$ et $D(n) = \mathcal{O}(g(n))$ alors $C(n)D(n) = \mathcal{O}(f(n)g(n))$.

— Si $C(n)= \mathcal{O}(D(n))$ alors $C(n) + D(n) = \Theta(D(n))$.

— Si $C(n) = \Theta(f(n))$ et $D(n) = \mathcal{O}(f(n))$ alors $C(n) + D(n) = \Theta(f(n))$.

Preuve : Exercice (TD)

2.3.2 Complexités Usuelles

Les règles précédentes nous indiquent que l'on peut facilement ramener des complexités temporelles à des ordres de grandeurs et majorants plus simples, en enlevant des facteurs multiplicatifs et des termes négligeables. Par exemple, si $C(n) = 3n^3 + \frac{1}{2}x + 1$, on pourra dire simplement que $C(n) = \mathcal{O}(n^3)$.

Cela permet de définir une familles de fonctions usuelles auxquelles on comparera les fonctions. Voici un ensemble de complexités rangées par ordre croissant que l'on trouvera de façon récurrente :

- $\mathcal{O}(1)$: complexité **constante**
- $\mathcal{O}(\log_a(n))$, $a > 1$: complexité **logarithmique**
- $\mathcal{O}(n)$: complexité **linéaire**
- $\mathcal{O}(n\log(n))$: complexité **linéarithmique**
- $\mathcal{O}(n^2)$: complexité **quadratique**
- $\mathcal{O}(n^3)$: complexité **cubique**
- $\mathcal{O}(n^k)$, $k \in \mathbb{N}^*$: complexité **polynomiale**
- $\mathcal{O}(a^n)$, $a > 1$: complexité **exponentielle**

Remarque. Pour a et a' deux réels strictement supérieurs à 1, on a $log_a(n) = \Theta(log'_a(n))$. Le choix du a dans la complexité logarithmique ou linéarithmique n'a donc pas d'importance. Les calculs feront le plus souvent intervenir des logarithmes en base 2 dans notre cas.

Proposition

Soient $k, k \in \mathbb{R}$ avec $1 < k < k'$, soient $a, a' \in \mathbb{R}$ avec $1 < a < a'$. On a :

$$\mathcal{O}(1) \subset \mathcal{O}(\log(n)) \subset \mathcal{O}(n) \subset \mathcal{O}(n\log(n)) \subset \mathcal{O}(n^k) \subset \mathcal{O}(n^{k'}) \subset \mathcal{O}(a^n) \subset \mathcal{O}(a'^n)$$

2.3.3 Ordres de Grandeurs en Pratique

Comme on l'a indiqué, les ordres de grandeurs, majoration et minoration servent à avoir une idée des différences de comportements entre plusieurs algorithmes utilisés pour un problème similaire, lorsqu'ils sont appliqués sur des entrées de tailles très différentes. Ainsi, la différence entre un coût exponentiel et un coût linéaire sera difficilement observable sur des entrées de petite taille. En revanche, elles seront facilement observables sur des entrées de très grande taille, on pourra obserber qu'un calcul qui se fait de façon quasi-immédiate dans un cas peut prendre plusieurs secondes (ou minutes, heures, jours, années...) avec un autre algorithme.

En informatique, on calcule la vitesse de calcul d'un système informatique en FLOPS (FLoating-point Operations Per Second) : on calcule le nombre moyen d'opérations élémentaires effectuées par seconde. Pour un calcul donné, connaissant le coût C (nombre d'opérations réalisées) et le nombre n_{FLOPS} d'opérations élémentaires par seconde, le temps de calcul t en secondes est donné par :

$$t = \frac{C}{n_{FLOPS}}$$

Sur un processeur actuel, la vitesse de calcul est de l'ordre de la centaine de GigaFLOPS (c'est-à-dire 10^{11} opérations par secondes). Voici, pour différentes valeurs de n et pour différentes complexités algorithmiques, le temps de calcul avec cette vitesse de calcul :

	10^2	10^3	10^4	10^5	10^6
$\log(n)$	0.7 ns	1 ns	1.3 ns	1.7 ns	2 ns
n	1 ns	10 ns	0.1 μs	1 μs	10 μs
$n\log(n)$	5 ns	0.7 μs	0.9 μs	12 μs	1 ms
n^2	0.1 μs	10 μs	1 ms	0.1s	10s
n^3	10 μs	10 ms	10s	2.8h	116 jours
2^n	$4 \cdot 10^{11}$ ans	10^{290} ans	...	...	...

TABLE 1 – Temps de calcul à 10^{11} FLOPS en fonction de la taille de l'entrée et de la complexité algorithmique

2.3.4 Sommes usuelles

On va voir qu'un des éléments dans le calcul de la complexité est le calcul du nombre d'opérations à l'intérieur d'une boucle ; lorsque le nombre d'opérations au sein de la $i$ème itération dépend du nombre i, on peut être amené à calculer des sommes spécifiques pour trouver le nombre total d'opérations sur l'ensemble des itérations de la boucle (et par conséquent son ordre de grandeur).

Voici une liste de sommes qui peuvent être fréquemment rencontrées dans le calcul :

1. $\displaystyle\sum_{0\leq k\leq n} k = \frac{n(n+1)}{2} = \Theta(n^2)$
2. $\displaystyle\sum_{0\leq k\leq n} k^p = \Theta(n^{p+1})$ pour $p \in \mathbb{N}^*$
3. $\displaystyle\sum_{0\leq k\leq n} 2^k = 2^{n+1} - 1 = \Theta(2^n)$
4. $\displaystyle\sum_{1\leq k\leq n} \frac{1}{k} = H_n = \Theta(\log(n))$
5. $\displaystyle\sum_{0\leq k\leq n} \frac{1}{2^k} = 2 - \frac{1}{2^n} = \Theta(1)$
6. $\displaystyle\sum_{1\leq k\leq n} \log(k) = \log(n!) = \Theta(n\log(n))$

2.4 Analyse de Complexité

Comme on l'a dit, on obtient le coût d'un algorithme en obtenant le coût de chacune de ses composantes. En particulier, on a dit que l'on chercherait essentiellement des majorants du coût total, que l'on obtiendra en majorant le coût de chaque composantes.

Voici la façon de calculer les coûts pour chacun des outils de programmation qui ont été vus pour l'instant :

— **Opérations élémentaires** (opérations sur les valeurs, numériques, comparaisons) : complexité constante $\mathcal{O}(1)$

— **Instructions conditionnelles** : on considère les complexités de chacun des blocs conditionnels et on majore par le maximum.

— **Fonctions** : coût de la fonction pour la valeur sur laquelle elle est appelée, chaque fois qu'elle est appelée sur une valeur donnée dans le programme.

— **Boucle** : la complexité de la boucle est égale à la somme des complexités de chaque itération. Si on a un majorant pour chaque itération, la complexité totale de la boucle est donc majorée par la somme des majorants.

Attention

Une façon de majorer la complexité d'une exécution de boucle est d'avoir un majorant global de toutes les itérations, indépendant de celles-cin et de le multiplier par le nombre de tours de boucle ; cependant, cela peut amener à avoir un majorant qui n'est pas optimal.

Plusieurs cas particuliers peuvent apparaître dans le calcul de complexité de boucles. On peut notamment trouver les cas suivants :

2.4.1 Boucle bornée simple

Lors de l'utilisation d'une boucle `for`, dans laquelle le coût de chaque itération est constant ou majoré par une constante indépendante du nombre n d'itérations, on peut majorer la complexité totale de l'exécution de la boucle par n.

On note que si on a une incrémentation ou une décrémentation de k unités au lieu d'une seule, on va avoir $\lfloor \frac{n}{k} \rfloor$ itérations de boucle au lieu de n, si l'on cherche à calculer la complexité exacte.

Exemple. Pour l'exponentiation naïve : un appel à la fonction fait p tours de boucles, où p est la puissance indiquée. On fait donc p multiplications au total.

2.4.2 Boucles While

Dans le cas des boucles `while`, on pourra chercher à borner le nombre d'itérations de la boucle si c'est possible, en fonction de l'évolution des variables intervenant dans la condition d'arrêt.

Exemple. On considère l'algorithme qui calcule les termes de la suite de Fibonacci jusqu'à un seuil n donné :

```
int fibo(int n)
    {
    int a = 0;
    int b = 1;
    while (b < n){
```

```
        b = a + b;
        a = b - a;
    }
    return a;
    }
```

on peut estimer la complexité algorithmique de l'algorithme à l'aide de raisonnements mathématiques non-triviaux.

2.4.3 Boucles Consécutives

Dans le cas de deux boucles consécutives, les complexités s'ajoutent. L'ordre de grandeur de l'exécution successive des deux boucles est le plus grand des deux ordres de grandeur de chaque exécution.

2.4.4 Boucles emboîtées

Lorsque l'on a une boucle à l'intérieur d'une autre boucle :

- Si on a n_1 itérations de la boucle externe, et si le nombre n_2 de tours de la boucle interne ne dépend pas du numéro d'itération $i < n_1$ de la boucle externe, on a $n_1 \times n_2$ itérations du bloc à l'intérieur de la boucle interne ;
- Sinon, si le nombre n_2 d'itérations de la boucle interne varie selon l'étape d'itération i, on calcule la valeur de $n_2(i)$ pour chaque i puis la somme $\sum_{1 \leq i \leq n_1} n_2(i)$.