Université de Liège

INFO0946

Introduction à la Programmation

Un exercice dont vous êtes le Héros · l'Héroïne

Evaluation de la Complexité

Benoit Donnet Simon Liénardy Tasnim Safadi

20 octobre 2020

Préambule

Exercices

Dans ce « TP dont vous êtes le héros », nous vous proposons de suivre pas à pas la résolution d'un exercice sur l'évaluation de la complexité théorique.

Il est dangereux d'y aller seul [1] !

Partagez vos commentaires, questions, solutions alternatives sur le forum eCampus. N'hésitez jamais !

1. Référence vidéoludique bien connue des Héros.

4.1 Rappel sur la Complexité

La *complexité* s'intéresse à l'évaluation de l'efficacité d'une méthode de résolution de problème indépendamment de l'environnement (i.e., puissance CPU, quantité de RAM, version du compilateur, ...). Dit autrement, la complexité est l'étude formelle de la quantité de ressources nécessaire pour l'exécution d'un programme.
On envisage deux types de complexité :

complexité spatiale : utilisation de la mémoire que va nécessiter le programme ;

complexité temporelle : nombre d'opérations élémentaires effectuées par un programme.

Dans le cadre du cours INFO0946, nous nous intéressons à la complexité temporelle dans le pire des cas. Ceci nous permet d'avoir une borne supérieure sur l'efficacité de notre programme. D'autres formes d'étude de la complexité existent. Vous les aborderez dans le cours INFO2050.

Notations Pour évaluer la complexité d'un programme, nous allons devoir faire un inventaire des instructions qui le composent et représenter cet inventaire sous la forme d'une fonction mathématique que nous bornerons. Les notations que nous allons utiliser, pour cela, sont les suivantes :

n : la taille des données ;

$T(n)$: fonction mathématique reprenant le nombre d'opérations élémentaires du programme.

Alerte : Syndrome du Seigneur des Anneaux [a]

a. "Un Anneau pour les gouverner tous. Un Anneau pour les trouver. Un Anneau pour les amener tous, Et dans les ténèbres les lier." J. R. R. Tolkien. *The Lord of the Rings*. Ed. Allen & Unwin. 1954/1955.

Dans ce rappel, nous modélisons, arbitrairement, la taille des données par le symbole n. En pratique, ce symbole sera une des variables de votre code.
Ce n'est pas parce que, dans ce rappel (ou dans les Slides), nous utilisons le symbole n qu'il doit venir partout, quelque soit le code. Introduire le symbole n alors qu'il n'y a aucune variable `n` dans votre code est source d'erreur (et, dans 100% des cas, d'un 0 à la question lors d'une évaluation).

4.2 Enoncé

Soit le bout de code C suivant :

```
1  int i=0, j=0;
2  int n;
3  int cpt;
4
5  printf("Entrez une valeur pour n: ");
6  scanf("%d", &n);
7
8  for(i=0; i<n; i++){
9    cpt = 0;
10   for(j=n; j>1; j/=2)
11     cpt += i*j;
12 }
```

▷ **Exercice** Déterminez et démontrez **formellement** la complexité théorique du code ci-dessus. Pensez à justifier votre démonstration à l'aide des règles décrites dans le cours. N'hésitez pas à indiquer les numéros de lignes auxquelles vous vous référez.

4.2.1 Méthode de résolution

Pour résoudre ce problème, nous allons procéder pas à pas. Voici le programme :

1. Réécriture du code avec des boucles `while` (Sec. 4.3) ;
2. Découpe du code en régions (Sec. 4.4) ;
3. Découverte de la quantité d'instructions élémentaires de chaque région (Sec. 4.5) ;
4. Expression de la complexité du programme à l'aide de la notation de Landau (Sec. 4.6) ;

4.3 Réécriture du Code

Evaluer la complexité théorique d'un programme revient à déterminer le nombre d'instructions élémentaires de ce programme. Notre code contient des boucles `for` qui, par définition, expriment de manière succincte le comportement d'une boucle. Il est donc souhaitable, dès le départ, de réécrire explicitement toutes les instructions. Et, donc, transformer les boucles `for` en boucles `while`. C'est l'objectif de cette section.

Si vous voyez de quoi on parle, rendez-vous à la Section 4.3.4
Si vous ne maîtriser pas la boucle `for`, allez à la Section 4.3.1
Si vous ne maîtriser pas la boucle `while`, allez à la Section 4.3.2
Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 4.3.3

4.3.1 Rappel sur la Boucle `for`

L'instruction `for` est une structure itérative qui permet de spécifier explicitement les opérations à effectuer :

1. avant d'entrer dans la boucle pour la première fois ;
2. afin de décider s'il faut continuer (ou non) à itérer ;
3. entre deux itérations ;
4. dans le bloc d'instructions à répéter.

La syntaxe [2] de l'instruction `for` est la suivante :

```
for(instr1;expression;instr2){
  instr3;
}
```

L'instruction commence toujours par le mot-clé `for` (*pour*, en français). Dans un langage de programmation, un *mot-clé* est un mot réservé qui a une signification bien particulière dans le langage. Cela signifie donc que vous ne pouvez pas déclarer une variable (par exemple) dont l'identificateur est `for`. Cela provoquerait nécessairement une erreur de compilation car le compilateur ne pourra pas faire la différence entre une boucle `for` et une variable nommée `for`.

Le mot-clé `for` est suivi par des parenthèses entre lesquelles on place trois informations séparées par un `;`. Chacune de ces informations se présente sous la forme d'une expression. La différence entre les trois expressions se situe dans le flux d'exécution de la boucle (cfr. la sémantique de la boucle `for` ci-dessous). La première expression (instr_1) permet de construire une instruction exécutée une seule fois avant la boucle. La deuxième expression est évaluée de manière booléenne et indique si oui ou non on peut rentrer dans la boucle (i.e., exécuter le bloc d'instructions associé à la boucle). Enfin, la troisième expression, instr_2, construit une instruction qui sera exécutée après le bloc d'instructions associé à la boucle. Après les parenthèses, nous trouvons un bloc d'instructions (appelé le *Corps de la Boucle*), entouré d'accolades (elles ne sont pas obligatoires si le bloc contient une seule instruction). Les différentes instructions du bloc sont séparées par un `;`.

L'instruction `for` est comprise, par le compilateur, comme une seule instruction (quelque soit la quantité d'instructions du bloc). Il n'est néanmoins pas nécessaire de la terminer par un `;` (les accolades jouent ce rôle).

La sémantique [3] de l'instruction `for` est donnée par la Fig. 1. Dans cette figure, le bloc magenta correspond aux instructions avant la boucle. Le bloc couleur asperge correspond aux instructions après la boucle. Les flèches indiquent le sens du flux du programme, i.e., l'ordre d'exécution des différentes instructions et d'évaluation des expressions.

La Fig. 1 indique que lorsque le bloc magenta se termine, l'instr_1 est exécutée. Cette instruction ne sera exécuté qu'une et une seule fois. Typiquement, instr_1 sera l'initialisation du compteur de la boucle. Ensuite, on évalue l'expression de manière booléenne. Si c'est vrai ($\neq 0$), alors on entre dans la boucle et on exécute le Corps de la Boucle (instr_3). Une fois le Corps de la Boucle terminé, l'instr_2 est exécutée. Typiquement, il s'agit de l'incrémentation (ou décrémentation) du compteur de boucle. Ensuite, on réévalue l'expression. Quand l'évaluation devient fausse ($== 0$), alors on exécute les instructions qui suivent la boucle.

2. La syntaxe fait référence à l'ensemble des règles grammaticales d'une langue, i.e., comment les différents éléments doivent être assemblés pour former une phrase. Appliquée à la programmation (le C est un langage), c'est l'ensemble des règles permettant de construire des instructions correctes. La syntaxe des langages de programmation sera étudiée en détails dans le cours INFO0085.

3. La sémantique fait référence à l'éude du sens, de la signification des signes, dans le langage. Appliquée à la programmation, la sémantique consiste à donner une signification mathématique au programme. La sémantique permettra, entre autre, de faire des raisonnements formels – donc mathématiques – sur le programme. Ceci est intéressant pour les programmes requierant un niveau qualité très important (e.g., l'informatique dans l'aéronautique, dans les véhicules, ...). La sémantique des langages de programmation sera étudiée en détails dans le cours INFO0085.

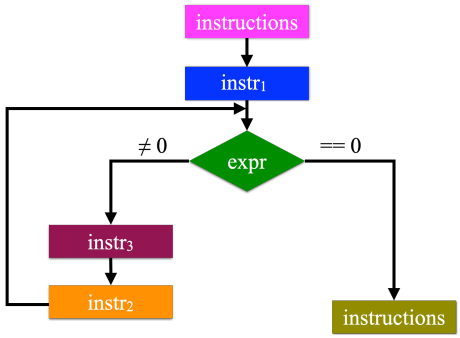


FIGURE 1 – Sémantique de l'instruction `for`. Un rectangle représente une suite d'instructions – éventuellement sous la forme d'un bloc (sauf pour le blocs bleu et orange qui représentent chacun une seule instruction). Un losange représente expression évaluée de manière booléenne (i.e., une condition).

Alerte : Vocabulaire

Le vocabulaire associé à une boucle est extrêmement important, dans le cadre de ce chapitre (bien entendu) mais, plus largement, dans le cadre des cours INFO0946 et INFO0947. Voici les éléments importants :

Gardien de Boucle. Il s'agit de l'expression qui est évaluée avant chaque entrée dans le Corps de la Boucle (losange vert sur la Fig. 2). Le résultat de l'évaluation de l'expression est de type booléen. Si l'évaluation est vraie ($! = 0$), alors le Corps de la Boucle est exécuté. Si l'évaluation est fausse ($== 0$), alors on sort de la boucle et on exécute l'instruction (ou la suite d'instructions) après la boucle (rectangle de couleur asperge sur la Fig. 2). Le Gardien de Boucle peut être une expression simple ou une expression plus complexe construite à l'aide des opérateurs booléens.

Critère d'Arrêt. Il s'agit de la condition qui est rencontrée (i.e., vraie) quand on sort de la boucle. Il y a donc un lien extrêmement fort entre Gardien de Boucle et Critère d'Arrêt : le Critère d'Arrêt est **la négation** du Gardien de Boucle. Le Critère d'Arrêt est un concept important. Toute boucle doit, un jour, rencontrer son Critère d'Arrêt. Sinon, la boucle ne se termine pas. On parle alors de *boucle infinie*. C'est l'une des erreurs les plus fréquentes et les plus mortelles.

Corps de la Boucle. Il s'agit de l'instruction ou du bloc d'instruction qui est exécuté chaque fois que le Gardien de Boucle est vrai (instr_3 et instr_2 sur la Fig. 2). Ce bloc est donc répété autant de fois que nécessaire, jusqu'à ce que le Critère d'Arrêt soit atteint.

Suite de l'Exercice

À vous ! Transformez les boucles `for` en boucles `while` et passez à la Sec. 4.3.4.
Si vous êtes mal à l'aise avec la boucle `while`, jetez un oeil à la Sec. 4.3.2
Si vous ne voyez pas comment procédez, reportez-vous à l'indice à la Sec. 4.3.3

4.3.2 Rappel sur la Boucle `while`

L'instruction **`while`** est une structure itérative qui permet :

1. de répéter une instruction (ou un bloc d'instructions) ;
2. tant qu'une condition (sous la forme d'une *expression booléenne*) est satisfaite ;
3. la condition est évaluée **avant** chaque itération.

La syntaxe[4] de l'instruction **`while`** est la suivante :

```
while(expression){
  instructions;
}
```

L'instruction commence toujours par le mot-clé **`while`** (*tant que*, en français). Dans un langage de programmation, un mot-clé est un mot réservé qui a une signification bien particulière dans le langage. Cela signifie donc que vous ne pouvez pas déclarer une variable (par exemple) dont l'identificateur est **`while`**. Cela provoquerait nécessairement une erreur de compilation.

Le mot-clé **`while`** est suivi par une expression entre paranthèses (elles sont obligatoires). Cette expression est évaluée de manière booléenne et indique si oui ou non on peut rentrer dans la boucle (i.e., exécuter le bloc d'instructions associé à la boucle). Après les parenthèses, nous trouvons un bloc d'instructions, entouré d'accolades (elles ne sont pas obligatoires si le bloc contient une seule instruction). Les différentes instructions du bloc sont séparées par un `;`.

L'instruction **`while`** est comprise, par le compilateur, comme une seule instruction (quelque soit la quantité d'instructions du bloc). Il n'est néanmoins pas nécessaire de la terminer par un `;` (les accolades jouent ce rôle).

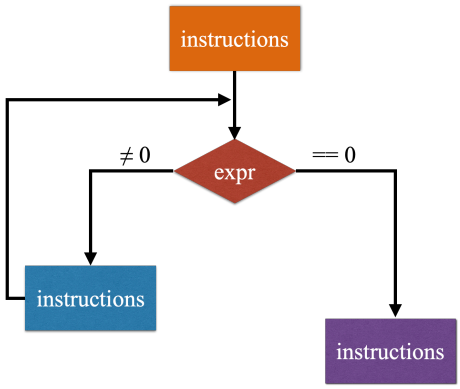


FIGURE 2 – Sémantique de l'instruction **`while`**. Un rectangle représente une suite d'instructions – éventuellement sous la forme d'un bloc. Un losange représente expression évaluée de manière booléenne (i.e., une condition).

La sémantique[5] de l'instruction **`while`** est donnée par la Fig. 2. Dans cette figure, l'instruction **`while`** reprend l'expression et le bloc d'instructions. Le bloc orange correspond aux instructions avant la boucle. Le

4. La syntaxe fait référence à l'ensemble des règles grammaticales d'une langue, i.e., comment les différents éléments doivent être assemblés pour former une phrase. Appliquée à la programmation (le C est un langage), c'est l'ensemble des règles permettant de construire des instructions correctes. La syntaxe des langages de programmation sera étudiée en détails dans le cours INFO0085.

5. La sémantique fait référence à l'éude du sens, de la signification des signes, dans le langage. Appliquée à la programmation, la sémantique consiste à donner une signification mathématique au programme. La sémantique permettra, entre autre, de faire des raisonnements formels – donc mathématiques – sur le programme. Ceci est intéressant pour les programmes requierant un

bloc mauve correspond aux instructions après la boucle. Les flèches indiquent le sens du flux du programme, i.e., l'ordre d'exécution des différentes instructions et d'évaluation des expressions.

La Fig. 2 indique que lorsque le bloc orange se termine, on évalue d'abord l'expression. L'instruction à exécuter ensuite va dépendre du résulat (booléen) de cette évaluation. Si c'est vrai ($\neq 0$), alors on entre dans la boucle et on exécute le corps de la boucle. Après la dernière instruction du boucle, on réévalue l'expression. Quand l'évaluation devient fausse ($== 0$), alors on exécute les instructions qui suivent la boucle.

Alerte : Vocabulaire

Le vocabulaire associé à une boucle est extrêmement important, dans le cadre de ce chapitre (bien entendu) mais, plus largement, dans le cadre des cours INFO0946 et INFO0947. Voici les éléments importants :

Gardien de Boucle. Il s'agit de l'expression qui est évaluée avant chaque entrée dans le Corps de la Boucle (losange rouge sur la Fig. 2). Le résultat de l'évaluation de l'expression est de type booléen. Si l'évaluation est vraie ($\neq 0$), alors le Corps de la Boucle est exécuté. Si l'évaluation est fausse ($== 0$), alors on sort de la boucle et on exécute l'instruction (ou la suite d'instructions) après la boucle (rectangle mauve sur la Fig. 2). Le Gardien de Boucle peut être une expression simple ou une expression plus complexe construite à l'aide des opérateurs booléens.

Critère d'Arrêt. Il s'agit de la condition qui est rencontrée (i.e., vraie) quand on sort de la boucle. Il y a donc un lien extrêmement fort entre Gardien de Boucle et Critère d'Arrêt : le Critère d'Arrêt est **la négation** du Gardien de Boucle. Le Critère d'Arrêt est un concept important. Toute boucle doit, un jour, rencontrer son Critère d'Arrêt. Sinon, la boucle ne se termine pas. On parle alors de *boucle infinie*. C'est l'une des erreurs les plus fréquentes et les plus mortelles.

Corps de la Boucle. Il s'agit de l'instruction ou du bloc d'instruction qui est exécuté chaque fois que le Gardien de Boucle est vrai (rectangle bleu sur la Fig. 2). Ce bloc est donc répété autant de fois que nécessaire, jusqu'à ce que le Critère d'Arrêt soit atteint.

Suite de l'Exercice

À vous ! Transformez les boucles `for` en boucles `while` et passez à la Sec. 4.3.4.
Si vous ne voyez pas comment procédez, reportez-vous à l'indice à la Sec. 4.3.3

niveau qualité très important (e.g., l'informatique dans l'aéronautique, dans les véhicules, ...). La sémantique des langages de programmation sera étudiée en détails dans le cours INFO0085.

4.3.3 Indice

Il faut repartir de la sémantique de la boucle `while` et de la sémantique de la boucle `for`. Il faut ensuite essayer de faire un "mapping" entre les deux sémantiques. Normalement, la transcription devrait être assez évidente.

Suite de l'Exercice
À vous ! Transformez la boucle `for` en boucle `while` et passez à la Sec. 4.3.4.

4.3.4 Mise en Commun de la Réécriture du Code

Le code initial comporte deux boucles `for`. Les boucles `for` et `while` fonctionnent de la même manière ; seul le positionnement des opérations change pour passer de l'une à l'autre. Repositionnons donc les différentes opérations sans les modifier, en traitant chaque boucle séparément.

Boucle Interne

Commençons par la boucle `for` interne. Soit

```
10 for(j=n; j>1; j/=2)
11   cpt += i*j;
```

La première instruction à effectuer avant d'entrer dans la boucle est j=n. Celle-ci sert à initialiser le compteur, soit `j`. Cette opération doit donc être placée juste avant la boucle `while`, de façon à ce qu'elle soit exécutée avant la toute première évaluation du Gardien de Boucle.

```
10 j = n;
11 while(...){
12   ...
13 }
```

On retrouve ensuite le Gardien de Boucle, soit l'expression j>1. Celui-ci doit être évalué à chaque tentative d'entrée dans la boucle. Dans une boucle `while`, il est placé dans les parenthèses juste après le mot-clé `while`.

```
10 j = n;
11 while(j>1){
12   ...
13 }
```

Après l'évaluation du Gardien de Boucle et si ce dernier est vrai, on exécute le Corps de la Boucle. Soit l'instruction cpt += i*j;. Il suffit donc de recopier cette instruction dans le corps de la boucle `while`.

```
10 j = n;
11 while(j>1){
12   cpt += i*j;
13 }
```

Enfin, dans une boucle `for`, l'instruction j/=2 est exécutée juste avant une nouvelle évaluation du Gardien de Boucle. Cette instruction sert à décrémenter le compteur de la boucle (soit `j`) et doit être effectuée à chaque tour de boucle. Elle sera donc placée à la fin du Corps de la Boucle du `while`.

```
10 j = n;
11 while(j>1){
12   cpt += i*j;
13   j/=2;
14 }
```

Boucle Externe

Passons maintenant à la boucle `for` externe. Soit

```
8  for(i=0; i<n; i++)
9    cpt = 0;
10   //Boucle interne
```

La première instruction à effectuer avant d'entrer dans la boucle est i=0. Celle-ci sert à initialiser le compteur, soit `i`. Cette opération doit donc être placée juste avant la boucle `while`, de façon à ce qu'elle soit exécutée avant la toute première évaluation du Gardien de Boucle.

```
i = 0;
while(...){
   ...
}
```

On retrouve ensuite le Gardien de Boucle, soit l'expression i<n. Celui-ci doit être évalué à chaque tentative d'entrée dans la boucle. Dans une boucle `while`, il est placé dans les parenthèses juste après le mot-clé `while`.

```
i = 0;
while(i<n){
   ...
}
```

Après l'évaluation du Gardien de Boucle et si ce dernier est vrai, on exécute le Corps de la Boucle. Soit l'instruction cpt =0 ;, suivie par la boucle interne. Il suffit donc de recopier cette instruction dans le corps de la boucle `while`.

```
i = 0;
while(i < n){
   cpt = 0;
   //Boucle interne
}
```

Enfin, dans une boucle `for`, l'instruction i++ est exécutée juste avant une nouvelle évaluation du Gardien de Boucle. Cette instruction sert à incrémenter le compteur de la boucle (soit `i`) et doit être effectuée à chaque tour de boucle. Elle sera donc placée à la fin du Corps de la Boucle du `while`.

```
i = 0;
while(i < n){
   cpt = 0;
   //Boucle interne
   i++;
}
```

Transcription Finale

Au final, le code complet devient le suivant

```
int i=0, j=0;
int n;
int cpt;

printf("Entrez une valeur pour n: ");
scanf("%d", &n);

i = 0;
while(i < n){
   cpt = 0;

   j = n;
   while(j>1){
      cpt += i*j;
      j/=2;
   }//fin while -- j

   i++;
}//fin while -- i
```

4.4 Découpe du Code en Région

Une fois le code réécrit, il faut le découper en diverses régions, avant de pouvoir déterminer la quantité d'instructions élémentaires. Ces régions nous permettront de travailler par Sp, chaque Sp étant une région particulière.

Si vous voyez de quoi on parle, rendez-vous à la Section 4.4.2

Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 4.4.1

4.4.1 Indice

Pour évaluer la complexité théorique d'un programme, il faut faire l'inventaire des instructions exécutées par celui-ci.

Le plus simple est de découper le programme en différents blocs : les déclarations, la Zone 1, une boucle et son Corps de la Boucle, une instruction conditionnelle ou encore la Zone 3.

Suite de l'Exercice

À vous ! Identifiez les différentes régions et passez à la Sec. 4.4.2.

4.4.2 Mise en Commun de la Découpe du Code

Première Approche

Commençons la découpe du code en régions par une première approche. Soit la découpe suivante :

```
1  int i=0, j=0;
2  int n;
3  int cpt;
4
5  printf("Entrez une valeur pour n: ");
6  scanf("%d", &n);
7
8  i = 0;
9
10 while(i < n){
11   cpt = 0;
12   j = n;
13
14   while(j>1){
15     cpt += i*j;
16     j/=2;
17   }//fin while -- j
18
19   i++;
20 }//fin while -- i
```

$T(A)$: lignes 1 à 8

$T(B)$: lignes 10 à 20

- La zone $T(A)$ reprend les lignes de code 1→8. C'est le code avant les boucles, qui sert aux diverses initialisations. Inutile ici de prendre chaque instruction séparément. On peut agréger, ces instructions, à l'aide d'une seule fonction $T(\cdot)$. L'idée n'est pas d'avoir une granularité ultra fine mais bien un ordre de grandeur.
- La zone $T(B)$ reprend les lignes de code 10→20. C'est le code correspondant aux deux boucles. Cette zone est beaucoup trop large. Afin de bien appliquer les règles pour l'évaluation des instructions élémentaires, il faut décomposer les boucles en leurs contenus.

Subdivision des Boucles

La zone $T(B)$ peut être décomposée de la façon suivante :

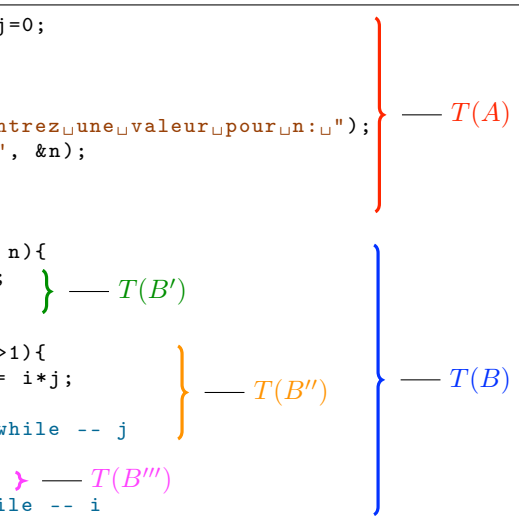
```
1  int i=0, j=0;
2  int n;
3  int cpt;
4
5  printf("Entrez une valeur pour n: ");
6  scanf("%d", &n);
7
8  i = 0;
9
10 while(i < n){
11   cpt = 0;
12   j = n;
13
14   while(j>1){
15     cpt += i*j;
16     j/=2;
17   }//fin while -- j
18
19   i++;
20 }//fin while -- i
```

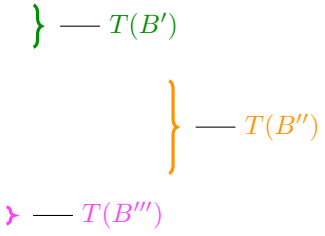
- la zone $T(B')$ reprend les lignes de code 11→12. Soit le début du Corps de la Boucle de la boucle externe qui correspond, aussi, à la phase d'initialisation de la boucle interne.
- la zone $T(B'')$ reprend les lignes de code 14→17. Soit la boucle interne. Il ne sera pas nécessaire de la décomposer plus que cela.
- la zone $T(B''')$ reprend la dernière ligne de code du Corps de la Boucle.

Découpe Finale

Au final, la découpe est la suivante :

```
1  int i=0, j=0;
2  int n;
3  int cpt;
4
5  printf("Entrez une valeur pour n: ");
6  scanf("%d", &n);
7
8  i = 0;
9
10 while(i < n){
11   cpt = 0;
12   j = n;
13
14   while(j>1){
15     cpt += i*j;
16     j/=2;
17   }//fin while -- j
18
19   i++;
20 }//fin while -- i
```





4.5 Analyse des Instructions de Chaque Région

On peut, maintenant, se pencher sur l'évaluation de la quantité d'instructions élémentaires pour chaque région.

Si vous voyez de quoi on parle, rendez-vous à la Section 4.5.3
Si les règles pour l'évaluation des instructions élémentaires vous posent problème, allez à la Section 4.5.1
Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 4.5.2

4.5.1 Rappel sur les Règles d'Evaluation des Instructions Élémentaires

Pour évaluer la quantité d'instructions élémentaires d'un programme, il faut construire une fonction $T(\cdot)$ qui les représent. La construction d'une telle fonction peut se faire en suivant six règles :

Règle 1 : instruction de base. Une instruction de base comprend l'écriture sur la sortie standard, la lecture sur l'entrée standard, la déclaration d'une variable et l'accès au contenu d'une variable. On considère, par défaut, que le temps nécessaire à l'exécution d'une instruction de base est constant. Dès lors, la fonction $T(\cdot)$ prend la forme $T(1)$;

Règle 2 : séquence d'instructions. Pour chaque instruction, on construit la fonction $T(\cdot)$ correspondante. Il suffit ensuite de prendre la somme des différentes fonctions. Par exemple :

```
1  instruction1; //T1(·)
2  instruction2; //T2(·)
3  instruction3; //T3(·)
```
} —— $T(\cdot) = T_1(\cdot) + T_2(\cdot) + T_3(\cdot)$

Attention, si la séquence d'instructions n'est composée que d'instructions élémentaires (voir Règle 1), alors la fonction $T(\cdot)$ correspondant à la séquence peut être simplifiée en $T(1)$. Il ne s'agit pas, ici, d'être le plus précis possible dans l'élaboration de la fonction $T(\cdot)$ mais bien d'avoir un ordre de grandeur sur le comportement global.

Règle 3 : instruction conditionnelle. La fonction $T(\cdot)$ pour une instruction conditionnelle est la fonction $T(\cdot)$ maximum de chaque branche. Soit :

```
1  if(expression)
2    Bloc1 //T1(·)
3  else
4    Bloc2 //T2(·)
```
} —— $T(\cdot) = max\left(T_1(\cdot), T_2(\cdot)\right)$

Le comportement de chacune des branches ($T_1(\cdot)$ et $T_2(\cdot)$) se construit en appliquant les six règles décrites dans ce rappel.

Règle 4 : instruction `switch`. Le comportement est assez proche de celui de la Règle 3. En effet, la fonction $T(\cdot)$ pour une instruction `switch` est le maximum de chaque `case`. Soit :

```
1  switch(variable){
2    case x1:
3      Traitement1; //T1(·)
4      break;
5    case x2:
6      Traitement2; //T2(·)
7      break;
8    ...
9    case xi:
10     Traitementi; //Ti(·)
11     break;
12   ...
13   case xk:
14     Traitementk; //Tk(·)
15     break;
16   default:
17     Traitement; //Tk+1(·)
18     break;
19 }
```
} —— $T(\cdot) = max\left(T_1(\cdot), T_2(\cdot), \ldots, T_i(\cdot), \ldots, T_k(\cdot), T_{k+1}(\cdot)\right)$

Comme pour Règle 3, le comportement de chacune des branches ($T_1(\cdot)$, ..., $T_{k+1}(\cdot)$) se construit en appliquant les six règles décrites dans ce rappel.

Règle 5 : instruction itérative. La fonction $T(\cdot)$ pour une instruction itérative est liée au nombre de tours de la boucle. Il s'agit, simplement, de la fonction $T(\cdot)$ du Corps de la Boucle multiplée par le nombre de tours. Soit (si on considère k tours de boucle) :

```
1
2  while(expression)
3    Bloc; //T_i(·)
4
```

$\}$ —— $T(\cdot) = \sum_{i=0}^{k-1} T_i(\cdot)$

Comme pour Règle 3, le comportement du Corps de la Boucle ($T_i(\cdot)$) se construit en appliquant les six règles décrites dans ce rappel.

Règle 6 : le programme complet. Cette dernière règle s'applique à l'ensemble du programme et fait écho à la Règle 2. A savoir que la fonction $T(\cdot)$ pour tout le programme correspond à la somme des fonctions $T(\cdot)$ des différentes parties du programme.

Suite de l'Exercice

À vous ! Analysez chacune des régions et passez à la Sec. 4.5.3.
Si vous ne voyez pas comment procédez, reportez-vous à l'indice à la Sec. 4.5.2

4.5.2 Indice

Commencez par identifier le paramètre utilisé pour la fonction T. Ce doit être une variable de votre programme (typiquement, celle qui ne varie pas et qui donne les dimensions de votre problème – pensez donc à inspecter les Gardiens de Boucles pour l'identifier). N'introduisez pas des variables qui n'existent pas dans votre code.

Repartez de la découpe en régions et appliquez les règles pour la construction de la fonction $T(\cdot)$. Appliquez ces règles de manière scrupuleuse, en conservant une approche formelle (i.e., mathématique) tout au long de votre raisonnement.

Suite de l'Exercice

À vous ! Analysez chacune des régions et passez à la Sec. 4.5.3.

4.5.3 Mise en Commun de l'Analyse

Pour rappel, la découpe en régions que nous allons utiliser est la suivante :

```
1  int i=0, j=0;
2  int n;
3  int cpt;
4
5  printf("Entrez une valeur pour n: ");
6  scanf("%d", &n);
7
8  i = 0;
9
10 while(i < n){
11   cpt = 0;
12   j = n;
13
14   while(j>1){
15     cpt += i*j;
16     j/=2;
17   }//fin while -- j
18
19   i++;
20 }//fin while -- i
```

— $T(A)$ (lignes 1–8)

— $T(B')$ (lignes 11–12)

— $T(B'')$ (lignes 14–17)

— $T(B''')$ (ligne 19)

— $T(B)$ (lignes 10–20)

Commençons par identifier la variable de la fonction T. Cela peut se faire en inspectant les Gardiens de Boucles. Soit `i<n` (boucle externe) et `j>1` (boucle interne). Or, il apparaît que `j` est initialisé à `n` (ligne 12). On voit que les variables `i` et `j` varient en fonction de `n`, qui a lui une valeur fixée au clavier (ligne 6). C'est donc bien `n` qui servira de paramètre à la fonction $T(\cdot)$.

Vision Globale

Par application des Règle 2 et Règle 6, on a le schéma global suivant :

$$T(n) = T(A) + T(B).$$

Premier Raffinement

$T(A)$ regroupe uniquement des instructions simples (déclaration de variables, simples affectations, affichage à l'écran, lecture au clavier). On sait, par application de la Règle 1, que dans ce cas la quantité d'instructions élémentaires est constante. On a donc $T(A) = 1$.

$T(B)$ reprend les deux boucles. On sait, par application de la Règle 5 que, dans ce cas, la quantité d'instructions élémentaires est la fonction $T(\cdot)$ du Corps de la Boucle multiplé par le nombre de tours. Ici, le Gardien de Boucle externe est `i<n`. Sachant que `i` est initialisé à 0 (ligne 1), la boucle effectuera `n` tours. Le Corps de la Boucle est, lui, composé des fonctions $T(B')$, $T(B'')$ et $T(B''')$. Soit :

$$\begin{aligned} T(n) &= T(A) + T(B) \\ &= 1 + \sum_{i=0}^{n-1} \left(T(B') + T(B'') + T(B''') \right). \end{aligned}$$

On peut appliquer, de nouveau, la Règle 1 sur $T(B')$ et $T(B''')$. Ce qui nous donne :

$$T(n) = 1 + \sum_{i=0}^{n-1} \left(1 + T(B'') + 1 \right).$$

A ce stade, il nous faut encore déterminer la fonction $T(\cdot)$ pour T(B") (i.e., la boucle interne)

Quantification de T(B")

Intéressons nous à la boucle interne. Soit :

```
14 while(j>1){
15   cpt += i*j;
16   j/=2;
17 }//fin while -- j
```

Comme indiqué dans la Règle 5, il faut commencer par déterminer le nombre de tours de la boucle. Ici, c'est un peu plus complexe car la variable d'itération, `j`, n'évolue pas de manière linéaire. Il faut donc "exécuter à la main" la boucle et voir l'évolution de `j` en cours d'exécution. Soit :

évaluation du Gardien de Boucle 0 [6] : $j = j_0$ [7]. Soit $j = \frac{j_0}{2^0}$

évaluation du Gardien de Boucle 1. Soit $j = \frac{j_0}{2}$. Dit autrement, $j = \left(\frac{j_0}{2^0}\right)/2$.

évaluation du Gardien de Boucle 2. Soit $j = \frac{j_0}{2^2}$. Dit autrement, $j = \left(\frac{j_0}{2^1}\right)/2$.

...

évaluation du Gardien de Boucle k. Soit $j = \frac{j_0}{2^k}$. Dit autrement, $j = \left(\frac{j_0}{2^{k-1}}\right)/2$.

Sur base de ce raisonnement,on peut trouver le nombre de tours de la boucle. Ainsi, nous savons que le Gardien de Boucle est `j>1` et que `j` est initialisé à `n` (dit autrement, j_0 vaut `n`). On peut donc reformuler le Gardien de Boucle en fonction du raisonnement ci-dessus. Soit $\frac{\texttt{n}}{2^k} > 1$. Il faut donc essayer d'estimer la valeur de `k` en fonction des variables du code. Soit

$$\left.\begin{aligned} \frac{n}{2^k} &> 1 \\ n &> 2^k \end{aligned}\right\} \text{—— Multiplication par } 2^k \text{ de chaque côté}$$
$$\left.\begin{aligned} \log_2 n &> k. \end{aligned}\right\} \text{—— Pour redescendre l'exposant } k\text{, logarithme en base 2 de chaque côté}$$

Pour la fonction $T(\cdot)$ associée à T(B"), il vient donc

$$\begin{aligned} T(B'') &= \sum_{i=0}^{\log_2 n} 1 \\ &= \log_2 n. \end{aligned}$$

On peut mettre 1 dans la somme car le Corps de la Boucle de la boucle est exclusivement composé d'instructions simples (cfr. Règle 1).

Formulation Finale

Au final, on obtient donc :

$$\left.\begin{aligned} T(n) &= 1 + \sum_{i=0}^{n-1} (1 + \log_2 n + 1) \end{aligned}\right\} \text{—— Dépliage de la notation } \sum$$
$$\left.\begin{aligned} &= 1 + 2 \times n \times \log_2 n \\ &= 2 \times n \times \log_2 n + 1. \end{aligned}\right\} \text{—— Réordonnancement des monomes}$$

6. La toute première évaluation du Gardien de Boucle, avant d'entrer pour la toute première fois dans le Corps de la Boucle.
7. j_0 se rapporte ici à la valeur initiale de `j`, avant toute modification da la boucle

4.6 Complexité Théorique

La dernière étape consiste à borner la fonction $T(\cdot)$ à l'aide la notation de landau.

Si vous voyez de quoi on parle, rendez-vous à la Section 4.6.3

Si vous ne maîtrisez pas la notation de Landau, regardez la Section 4.6.1

Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 4.6.2

4.6.1 Rappel sur la Notation de Landau

L'idée derrière l'évaluation de la complexité, dans le pire des cas, est de borner (supérieurement) la fonction $T(\cdot)$. Pour cela, nous utilisons la notation de *Landau*[8] (O ou "Big O").

La relation O est une relation de *domination* d'une suite par une autre. La lettre O est utilisée parce que le rythme de croissance d'une suite est aussi appelé son *ordre*.
Il s'agit d'une notation asymptotique dont la formulation mathématique est la suivante :

$$\text{Soient } f \text{ et } g, \text{ deux fonctions } \mathbb{N} \to \mathbb{R}^+$$
$$f \in O(g) \iff \exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+ : \forall n > n_0 : f(n) \leq c \times g(n)$$

La signification de cette formule est la suivante : $f(n)$ est de l'ordre de $O(g(n))$ s'il existe un seuil à partir duquel la fonction $f()$ est toujours dominée par la fonction $g()$, à une constante multiplicative fixée près. Ceci est illustré à la Fig. 3.

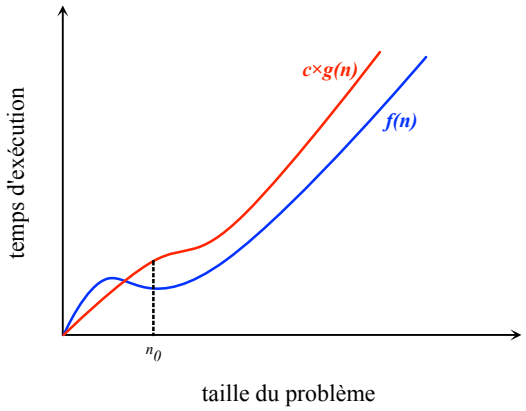


FIGURE 3 – Illustration de la définition de la notation de Landau.

Le symbole n utilisé dans les fonctions $f()$ et $g()$ représente la taille des données du programme. Il est donc "fonction" du problème et des identificateurs de variables que vous utilisez. Il faut ici être cohérent entre le symbole utilisé dans la notation de Landau et le symbole utilisé dans votre fonction $T(\cdot)$.

Notation	**Type de Complexité**	**Exemple**
$O(1)$	complexité constante	accès à une variable
$O(\log n)$	complexité logarithmique	recherche dichotomique
$O(n)$	complexité linéaire	triangulation de Delaunay
$O(n \times \log n)$	complexité linéarithmique	tri rapide
$O(n^2)$	complexité quadratique	parcours tableau 2D
$O(n^3)$	complexité cubique	parcours tableau 3D
$O(e^n)$	complexité exponentielle	décomposition en facteurs premiers
$O(n!)$	complexité factorielle	problème du voyageur de commerce
$O(2^{2^n})$	complexité doublement exponentielle	arithmétique de Presburger

TABLE 1 – Les complexités "classiques".

8. Edmund Landau (1877 – 1938) était un mathématicien allemand, qui est resté célèbre principalement pour la diffusion et l'usage de notations qui portent son nom (O, o, ω, Ω) bien que ces notations aient été inventées par d'autres.

Complexité et Ordres de Grandeur Classiques Le Tableau 1 reprend les grandes familles de complexité (il en existe bien d'autres, le Tableau ne fait que lister celles qui sont bien connues), tandis que la Fig. 4 illustre graphiquement le comportement de ces fonctions.

Une complexité au-delà de la complexité quadratique (i.e., $O(n^2)$) est, la plupart du temps, jugée peu efficace.

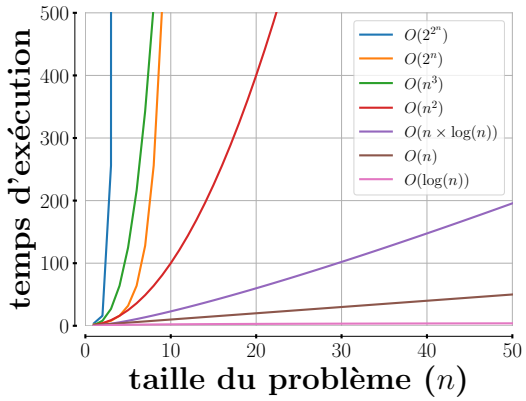


FIGURE 4 – Illustration des ordres de grandeur classiques.

La triangulation de Delaunay d'un ensemble de n points est l'unique triangulation telle qu'un cercle passant par les trois points d'un triangle ne contienne aucun autre point.

La décomposition en produit de facteurs premiers consiste à chercher à écrire un entier naturel non nul sous forme d'un produit de nombres premiers. Son utilisation principale, en informatique, se trouve en cryptographie asymétrique.

Le problème du voyageur de commerce est un problème d'optimisation qui, étant donné une liste de villes, et des distances entre toutes les paires de villes, détermine un plus court chemin qui visite chaque ville une et une seule fois et qui se termine dans la ville de départ (voir MATH0499).

L'arithmétique de Presburger est une arithmétique du premier ordre sans la multiplication (uniquement l'addition, le zéro et l'opérateur successeur).

Suite de l'Exercice

À vous ! Déterminez la complexité théorique et passez à la Sec. 4.6.3.
Si vous séchez, reportez-vous à l'indice à la Sec. 4.6.2

4.6.2 Indice

Repartez de la fonction $T(\cdot)$ et essayer de la borner à l'aide d'une notation de Landau bien connue. Pour ce faire, il faut que la fonction $T(\cdot)$ et la notation de Landau correspondante aient la même forme générale (et donc, suivent le même comportement général).

Suite de l'Exercice

À vous ! Déterminez la complexité théorique et passez à la Sec. 4.6.3.

4.6.3 Mise en Commun de la Complexité Théorique

La fonction $T(\cdot)$ est la suivante :

$$T(n) = 2 \times n \times \log_2 n + 1.$$

On voit bien que l'ordre de grandeur le plus important est $n \times \log_2 n$. La notation de Landau correspondante est $O(n \times log)$.

Nous sommes donc dans une situation de *complexité linéarithmique*.

Alerte : Microscopisme

Il est inutile, dans la notation $O()$ d'indiquer la base du logarithme. Que ce soit un logarithme en base 2 ($\log_2$), 3 ($\log_3$) ou même 100 ($\log_{100}$).
C'est la forme générale de la fonction logarithmique et sa vitesse de croissance qui importe.