CALCUL DE LA COMPLEXITÉ ALGORITHMIQUE

1. CONTEXTE

2.1 Enoncé

Ecrire une fonction qui renvoie 1 si un nombre appartient à une liste L ou 0 si non.

2.2 Solutions

Plusieurs solutions sont possibles pour répondre à cet exercice :

- La boucle **for**
- La boucle **while**
- Une fonction récursive
- Etc.

Exercice : écrire les pseudocodes (algorithmes) et les codes (programmes) correspondant aux différentes approches de solution.

2.3 Quelques questions

- Le code le plus courts : est-il meilleur ?
- Comment peut-on désigner le meilleur code parmi les solutions proposées ?

2.4 Définitions

La **théorie de la complexité** est un domaine des mathématiques, et plus précisément de l'informatique théorique, qui étudie formellement la *quantité de ressources* (temps et/ou espace mémoire) nécessaire pour résoudre un *problème algorithmique* au moyen de l'exécution d'un algorithme.

2.5 Objectifs des calculs de complexité

- Pouvoir prévoir le temps d'exécution d'un algorithme ;
- Pouvoir comparer deux algorithmes réalisant le même traitement.

Exemples :

- Si on lance le calcul de la factorielle de 100, combien de temps faudra-t-il attendre le résultat ?
- Quel algorithme de tri vaudrait-il mieux utiliser pour retrier un tableau où on vient de changer un élément ?

2.6 Complexité temporelle vs spatiale

L'efficacité d'un algorithme peut être évaluée en temps et en espace :

- Complexité en temps : évaluation du temps d'exécution de l'algorithme ;
- Complexité en espace : évaluation de l'espace mémoire occupé par l'exécution de l'algorithme.

Règle (non officielle) de l'espace-temps informatique : pour gagner du temps de calcul, on doit utiliser davantage d'espace mémoire.

On s'intéresse essentiellement à la complexité en temps (ce qui n'était pas forcément le cas quand les mémoires coutaient cher)

Exemple : échange de deux valeurs entières

```
// échange des valeurs de deux variables
entier x, y, z;
... // initialisation de x et y
z ← x;
x ← y;
y ← z;
```

```
// échange des valeurs de deux variables
entier x, y;
... // initialisation de x et y
x ← y-x;
y ← y-x;
x ← y+x;
```

- La première méthode utilise une variable supplémentaire et réalise 3 affectations ;
- La deuxième méthode n'utilise que les deux variables dont on veut échanger les valeurs, mais réalise 3 affectations et 3 opérations.

2. MÉTHODES

L'évaluation de la complexité peut se faire à plusieurs niveaux :

- Au niveau de l'exécution du programme expérimentalement ;
- Au niveau purement algorithmique, par l'analyse et le calcul.

2.1 Méthode expérimentale

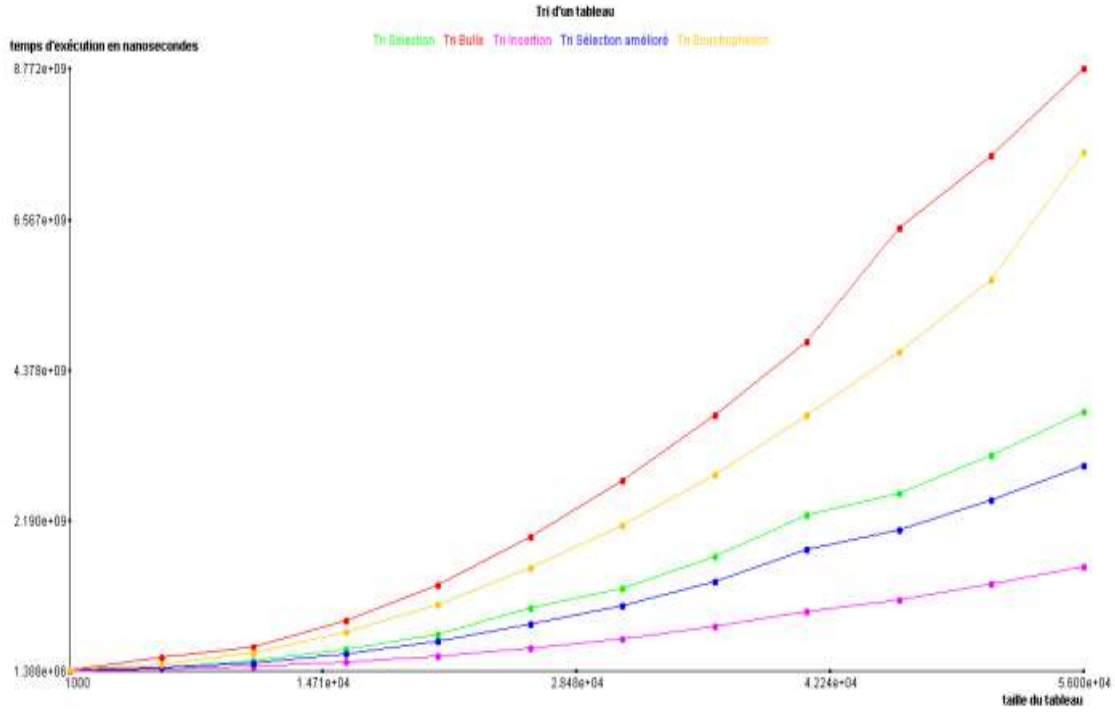
La solution la plus intuitive pour choisir la meilleure solution consiste à mesurer le temps d'exécution exact pour chaque algorithme.

https://www-geeksforgeeks-org.translate.goog/how-to-measure-time-taken-by-a-program-in-c/?_x_tr_sl=en&_x_tr_tl=fr&_x_tr_hl=fr&_x_tr_pto=rq

En C : pour calculer le temps d'un processus, nous pouvons utiliser la fonction clock(), disponible *dans time.h.* Nous pouvons appeler la fonction clock au début et à la fin du code pour lequel nous mesurons le temps, soustraire les valeurs, puis diviser par CLOCKS_PER_SEC (le nombre de tops d'horloge par seconde) pour obtenir le temps processeur, comme suit :

```
#include <time.h>
clock_t start, end;
double cpu_time_used;
start = clock();
... /* Faites le travail. */
end = clock();
cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

Il est possible d'évaluer de façon expérimentale le temps d'exécution des programmes.



Inconvénients :

- Implémentation de l'algorithme ;
- Résultats dépends de l'environnement ;
- Lancer plusieurs exécutions, qui peuvent prendre du temps.

⇒**Environnement de simulation**

- Processeur Intel Core i5-4210U CPU 1,70GHz x 4
- Mémoire vive 8Go
- OS : Linux Ubuntu 14.04 LTS 64bits

⇒**Résultats de l'expérience**

N	10^2	10^3	10^4	10^5	10^6	10^7
Tri fusion	0,3ms	0,9ms	5,5ms	20ms	215ms	2,3s
Tri insertion	1,8ms	67ms	5,7ms	11mn	19h	66j

2.2 Méthode théorique

Avantages :

- Comparer les algorithmes théoriquement ;
- Pas besoin d'implémentation.

Etapes de calcul de la complexité

1. Identifier le paramètre de complexité de l'algorithme ;
2. Calculer le coût de l'algorithme en fonction du paramètre de la complexité ;
3. Calculer la complexité asymptotique.

2.2.1 Identification du paramètre de complexité

Le paramètre de la complexité est ce qui va faire varier le temps d'exécution de l'algorithme.
Exemple 1 : le calcul de la factorielle.

```
FONCTION factorielle(nombre) : ENTIER
PARAMETRE
        (D) nombre : ENTIER
VARIABLES
        fact, cpt : ENTIER
DEBUT factorielle
        fact ← 1
        POUR cpt ← 2 À nombre FAIRE
                fact ← fact * cpt
        FPOUR
        RENVOYER fact
FIN factorielle
```

Le paramètre de complexité est la valeur de nombre : n = nombre.

Exemple 2 : multiplication des éléments d'un tableau par un nombre.

```
PROCEDURE multiplication(tab, nbElem, x)
PARAMETRES
        (D) nbElem : ENTIER
        (D) tab : TABLEAU[nbElem] d'ENTIER
        (D) x : ENTIER
VARIABLES
        cpt : ENTIER
DEBUT multiplication
        POUR cpt ← 1 À nbElem FAIRE
                tab[cpt] ← tab[cpt] * x
        FPOUR
FIN multiplication
```

Le paramètre de complexité est le nombre d'éléments (nbElem) : n = nbElem.

Exemple 3 : somme de la i ème ligne d'une matrice.

```
PROCEDURE sommeLigne(tab, nbLigne, nbColonne, iLigne)
PARAMETRES
        (D) nbLigne, nbColonne, iLigne : ENTIER
        (D) tab : TABLEAU[nbLigne][nbColonne] d'ENTIER
VARIABLES
        Somme, cpt : ENTIER
DEBUT sommeLigne
        somme ← 0
        POUR cpt ← 1 À nbColonne FAIRE
                somme ← somme + tab[iLigne][cpt]
        FPOUR
FIN sommeLigne
```

Le paramètre de complexité est le nombre de colonnes : n = nbColonne.

Exemple 4 : table de multiplication d'un entier.

```
PROCEDURE tabMult(n)
VARIABLES
        resultat, cpt : ENTIER
DEBUT tabMult
        POUR cpt ← 1 À 10 FAIRE
                resultat ← n * cpt
                AFFICHER(resultat)
        FPOUR
FIN tabMult
```

Pas de paramètre de complexité.

2.2.2 Calcul du coût d'un algorithme

Pour calculer la complexité temporelle d'un algorithme, on détermine d'abord son coût en nombre d'opérations de base.

Pour déterminer le coût d'un algorithme, on se fonde en général sur le *modèle de complexité* suivant :

1. Une affectation, une comparaison ou l'évaluation d'une opération arithmétique ayant en général un faible temps d'exécution, celui-ci sera considéré comme l'unité de mesure du coût d'un algorithme.
2. Le coût des instructions p et q en séquence est la somme des coûts de l'instruction p et de l'instruction q.
3. Le coût d'un test **if (b) p else q** est inférieur ou égal au maximum des coûts des instructions *p* et q, plus le temps d'évaluation de l'expression b.
4. Le coût d'une boucle **for** est égal au nombre d'éléments de l'itérable multiplié par le coût de l'instruction p si ce dernier ne dépend pas de la valeur de i. Quand le coût du corps de la boucle dépend de la valeur de i, le coût total de la boucle est la somme des coûts du corps de la boucle pour chaque valeur de i.
5. Le cas des boucles **while** est plus complexe à traiter puisque le nombre de répétitions n'est en général pas connu a priori. On peut majorer le nombre de répétitions de la boucle de la même façon qu'on démontre sa terminaison et ainsi majorer le coût de l'exécution de la boucle.

Exemples de calcul de coût d'un algorithme

Numéro	Bloc d'instructions/Programme	Coût
1.	x = a + b ;	2 = 1 addition + 1 affectation
2.	A = 0 ; B = 1 ; X = A + B ;	4 = 3 affectations + 1 addition
3.	A = X + Y + Z ;	3 = 2 additions + 1 affectation

4.	if (A==B){
	printf("\nÉgalité") ;
	}else{
	C = A – B ;
	printf("%d", C) ;
	}
		1 opération +
		1 opération
		3 opérations
		+ max(1, 3) = 3
		= 4

5.	s = 0 ;
	for (cpt=1 ; cpt<11 ; cpt++){
	printf(″\nX = ″) ;
	scanf(″%d″, &x) ;
	s = s + x ;
	}
		41

6.	for (i=1 ; i<11 ; i++)
	{
	printf(″\nTable de : %d″, i) ;
	for (j=1 ; j<11 ; j++)
	{
	printf(″%d x %d = %d″, i, j, i*j) ;
	}
	}
		110

7.	for (int i = 1; i < 5; ++i) {
	for (int j = 1; j <= i; ++j) {
	printf("X");
	}
	printf(″\n″);
	}
		1+2+3+4 = 10

8.	void table1(int n) {
	for (int i = 1; i <= 10; ++i) {
	printf(″%d\n″, i*n);
	}
	}
		20

9.	void table2(int n) {
	for (int i = 0; i < n; ++i) {
	printf(″%d\n″, i*i);
	}
	}
		2n

10.	int somme(int n) {
	int i = 1, s = 0;
	while (i <= n) {
	s = s + i;
	i = i + 1;
	}
	return s;
	}
		2 + 5n + 1

11.	void table3(int n) {
	for (int i = 0; i < n; i++) {
	for (int j = 0; j < n; j++) {
	printf("%d ", i * j);
	}
	printf("\n");
	}
	}
		$2n^2+n$

12.	void affiche(int n) {
	int i = 1;
	while (i <= n) {
	printf("%d\n", i);
	i = i + 2;
	}
	}
		1+4*(n/2) + 1

2.2.3 Nuances de la complexité temporelle

Lorsque, pour une valeur donnée du paramètre de la complexité, le temps d'exécution varie selon les données d'entrée, on peut distinguer :

- La complexité au pire : temps d'exécution maximum, dans le cas le plus défavorable (le pire des cas) ;
- La complexité au mieux : temps d'exécution minimum, dans le cas le plus favorable (le meilleur des cas) ; en pratique, cette complexité n'est pas très utile ;
- La complexité moyenne : temps d'exécution dans un cas médian, ou moyenne des temps d'exécution.

Le plus souvent, on utilise la complexité au pire, car on veut borner le temps d'exécution.

bool rechercherElement (int L[] , int taille, int x)
{
  int i=0;
  int trouve = 0;
  while (i < taille) {
  if (L[i]==x) {
  trouve = 1;
  }
  i = i + 1;
  }
  return trouve;
}
	La complexité au pire :
	4n+3
	La complexité au mieux :
	8
	La complexité au moyen :
	3n+3

2.2.4 La notation O (notation Landau)

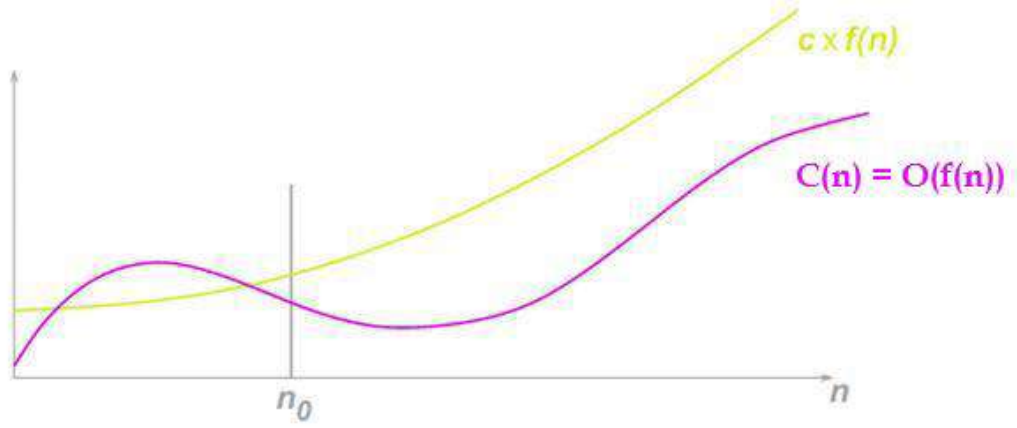
Pour écrire la complexité, on utilise la **notation Big O**, de la forme $O(f(n))$. Elle permet d'exprimer la complexité d'un algorithme en fonction de la taille de l'entrée, généralement notée n. Le $f(n)$ dans $O(f(n))$ représente ainsi une **fonction mathématique** qui décrit comment la complexité de l'algorithme croît avec n.

Lorsqu'on utilise la notation Big O, l'accent est mis sur le comportement des algorithmes à mesure que la taille de l'entrée devient très grande, donc pour de grandes valeurs de n.

Soit C(n) une fonction qui désigne le temps de calcul d'un algorithme A.

On dit que C(n) est en grand O de f(n) : C(n) = O(f (n)) si et seulement

si : ∃(n0; c) telle que C(n) <= c*f (n) pour tous n >= n0



La notation O(f (n)) est aussi appelée Notation de Landau : (Complexité asymptotique) (i.e.quand n → ∞)

Cette méthode écarte les éléments de moindre importance, comme les constantes et les termes mineurs, se concentrant sur le terme qui a le plus grand impact sur la croissance de la fonction.

Exemple 1 :

Si la complexité exacte d'un algorithme est $O(3n^2 + 5n + 7)$, on simplifie cela en $O(n^2)$, car pour de grandes valeurs de n, n^2 est le terme dominant.

Exemple 2 :

Si C(n) = 3n + 6 alors C(n) = O(n)

Démonstration :

En effet, pour n >= 2, on a 3n + 6 <= 9n ; la quantité 3n + 6 est donc bornée, à partir d'un certain rang, par le produit de n et d'une constante.

3 *2 + 6 <= 9 *2

3 *3 + 6 <= 9 *3

3 *4 + 6 <= 9 *4

3 *5 + 6 <= 9 *5

Exemple 3 :

Si $C(n) = n^2 + 3n$ alors $T(n) = O(n^2)$

Démonstration :

En effet, pour n >= 3, on a $n^2 + 3n <= 2n^2$; la quantité $n^2 + 3n$ est donc bornée, à partir d'un certain rang, par le produit de n^2 et d'une constante.

Simplification :

On calcule le temps d'exécution comme avant, mais on effectue les simplifications suivantes :

- On oublie les constantes multiplicatives (elles valent 1) ;
- On annule les constantes additives ;
- On ne retient que les termes dominants.

Soit un algorithme effectuant $C(n) = 4n^3-5n^2+ 2n + 3$ opérations :

=> On remplace les constantes multiplicatives par 1 : $1n^3-1n^2+ 1n + 3$

=> On annule les constantes additives : $n^3-n^2+ n + 0$

=> On garde le terme de plus haut degré : n^3

Donc : $C(n) = O(n^3)$.

Propriétés :

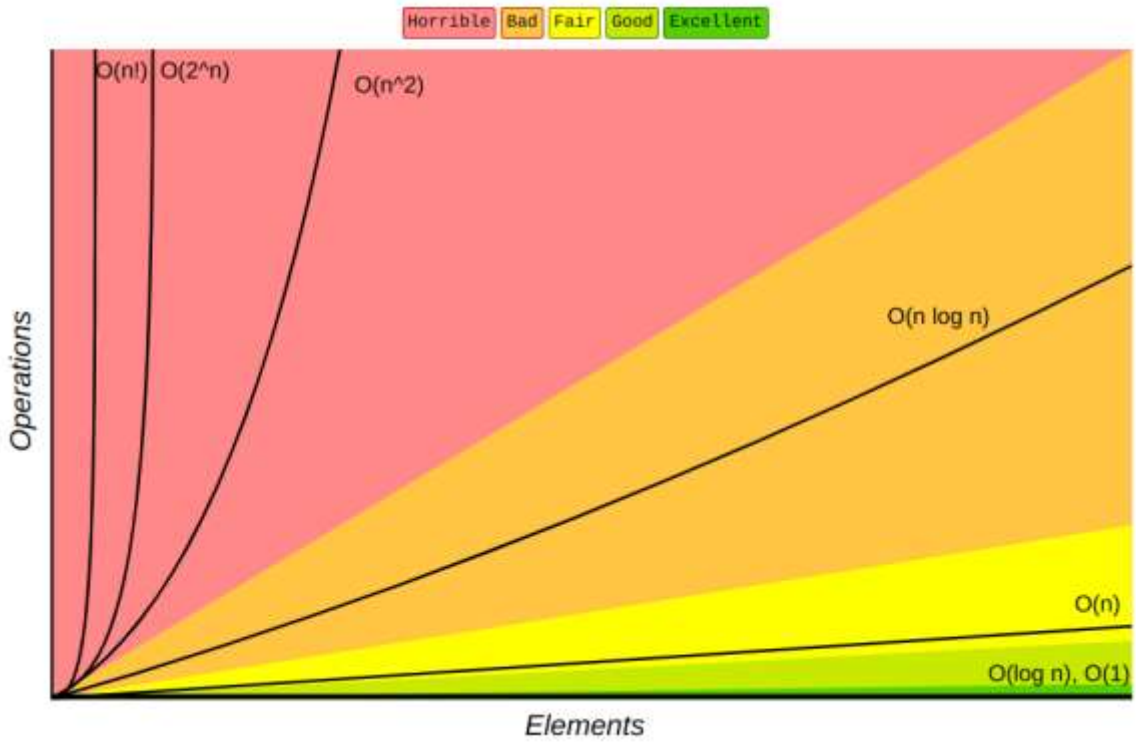
- c *O(f (n)) = O(f (n))
- O(f (n)) + O(g(n)) = O(f (n) + g(n))
- O(f (n)) * O(g(n)) = O(f (n) * g(n))

2.2.5 Classes de complexité

Maintenant que nous avons vu comment calculer des complexités, nous pouvons comparer et classer les algorithmes en fonction de leur efficacité. Connaître et comprendre les classes de complexité est important pour choisir le bon algorithme pour résoudre le problème. Voici les complexités les plus courantes :

- **Complexité constante $O(1)$** : Cette classe indique que l'algorithme exécute un nombre fixe d'opérations, ou s'exécute dans un temps constant, indépendamment de la taille de l'entrée. Un exemple typique est l'accès à un élément dans un tableau par son index.
- **Complexité logarithmique $O(\log(n))$** : Les algorithmes ayant une complexité logarithmique réduisent l'espace de problème à chaque itération. Par exemple, l'algorithme de recherche binaire divise par deux l'espace de recherche à chaque étape.
- **Complexité linéaire $O(n)$** : Cette catégorie indique que la quantité d'opérations réalisées ou le temps d'exécution augmentent de manière proportionnelle avec la taille des données d'entrée. La recherche linéaire est un exemple typique, nécessitant dans le pire des cas de parcourir la totalité de la liste pour trouver un élément.
- **Complexité quasi-linéaire $O(n \cdot \log(n))$** : On retrouve dans cette classe quelques algorithmes de tri populaires, comme le tri rapide et le tri fusion. En moyenne, ils sont plus performants pour que le tri par insertion.
- **Complexité quadratique $O(n^2)$** : Comme on l'a observé plus tôt, la complexité quadratique se manifeste dans les algorithmes où une boucle est imbriquée dans une autre, chaque boucle parcourant l'intégralité des données. La complexité quadratique fait partie de la **complexité polynomiale**, de la forme $O(n^a)$ où a est un nombre entier.
- **Complexité exponentielle $O(2^n)$** : Chaque élément supplémentaire dans les données d'entrée entraîne un doublement du nombre d'opérations et/ou du temps d'exécution. De manière générale, une complexité $O(a^n)$ implique que chaque nouvel élément augmente les opérations ou le temps d'exécution d'un facteur de a. Un exemple classique d'algorithme à complexité exponentielle est l'implémentation naïve de la fonction qui génère un nombre de Fibonacci.
- **Complexité factorielle $O(n!)$** : Avec les algorithmes à complexité factorielle, chaque nouvel élément multiplie le nombre total d'opérations par le nombre d'éléments déjà présents. L'arrangement de différentes villes dans un itinéraire de voyage, en explorant tous les itinéraires possibles, est un exemple typique.

Complexité	Nom courant	Description
$O(1)$	constante	Le temps d'exécution ne dépend pas des données traitées, ce qui est assez rare !
$O(log(n))$	logarithmique	augmentation très faible du temps d'exécution quand le paramètre croit.
$O(n)$	linéaire	augmentation linéraire du temps d'exécution quand le paramètre croit.
$O(nlog(n))$	quasi-linéaire	augmentation un peu supérieure à O(n)
$O(n^2)$	quadratique	quand le paramètre double, le temps d'exécution est multiplié par 4. Exemple : algorithmes avec deux boucles imbriquées.
$O(n^k)$	polynômiale	ici, n^k est le terme de plus haut degré d'un polynôme en n ; il n'est pas rare de voir des complexités en $O(n^3)$ ou $O(n^4)$.
$O(k^n)$	exponentielle	quand le paramètre double, le temps d'exécution est élevé à la puissance k avec $k > 1$.
$O(n!)$	factorielle	asymptotiquement équivalente à n^n



Courbe des nombres d'opérations en fonction du nombre d'éléments en entrée de l'algorithme. (Eric Dowell/Big-O Cheat Sheet)

Le graphique ci-dessus montre le comportement des différentes complexités en fonction du nombre d'éléments en entrée. Il met en évidence l'efficacité des algorithmes à complexité logarithmique ou linéaire pour de grandes entrées, tandis que les complexités quadratique, exponentielle et factorielle deviennent rapidement impraticables à mesure que le nombre d'éléments augmente.

Cependant, les algorithmes à complexité quadratique ou exponentielle, ne sont pas nécessairement à bannir à tout prix. Dans des cas où la taille des données est relativement petite ou lorsque la complexité du problème à résoudre l'exige, ces algorithmes peuvent fournir des solutions exactes et optimales. Choisir le bon algorithme dépend de la taille des données mais aussi d'autres nombreux facteurs.

2.2.6 Calcul de la complexité asymptotique

Ce qui nous intéresse n'est pas un temps précis d'exécution mais un ordre de grandeur de ce temps d'exécution en fonction de la taille des données. Cela se calcule en deux approximations :

- Considérer la complexité au pire des cas ;
- Considérer le calcule asymptotique de la complexité, s'intéresser juste aux grandes quantités des données.

Exercices :

Numéro	Code	Complexité

1.	//Calcul de la factorielle
	long factoriel(int n)
	{
	long f = 1;
	for (int i = 2; i <= n; ++i) {
	f = f * i;
	}
	return f;
	}
		Complexité linéaire : O(n)

2.	//Recherche du minimum dans un tableau
	int minimum(int L[], int size)
	{
	int min = L[0];
	for (int i = 1; i < size; i++) {
	if (L[i] < min) {
	min = L[i];
	}
	}
	return min;
	}
		Complexité linéaire : O(n)

3.	//Table de multiplication
	void table_multiplication (int n)
	{
	for (int i = 1; i <= 10; i++) {
	printf("%d x %d = %d\n", n, i, n * i);
	}
	}
		Complexité constante : O(1)

4.	//Déclarations
	int A[50][50]; // matrice donnée
	int B[50][50]; // matrice donnée
	int C[50][50]; // matrice résultat
	int N ; //Nombre de lignes de A
	int M ; //Nombre colonnes A = nb lignes de B
	int P ; //Nombre de colonnes B
	int I, J, K; /* indices courants */
	…
	//Calcul résultat
	for (I=0; I<N; I++){
	for (J=0; J<P; J++){
	C[I][J]=0;
	for (K=0; K<M; K++){
	C[I][J] += A[I][K]*B[K][J];
	}
	}
	}
		Complexité polynomiale : $O(n^3)$

5.	//Tri par bulle	
6.	//Tri par sélection	
7.	//Tri par insertion	