Preuve et complexité

La nature des choses veut que plus un programme a de données à traiter, plus il prend du temps pour le faire. Cependant, certains algorithmes se comportent mieux que d'autres lorsque le nombre des données à traiter augmente.

Le nombre des opérations élémentaires effectuées par un algorithme est une fonction directe du nombre de données à traiter. La complexité d'un algorithme est donc directement reliée à cette fonction : plus elle croît rapidement avec le nombre de données à traiter, plus la complexité de l'algorithme est grande.

En réalité, la fonction exacte donnant le nombre d'opérations élémentaires effectuées par un algorithme n'est pas toujours facile à calculer. Cependant, il existe toujours une fonction plus simple qui dispose du même comportement que la fonction du nombre d'opérations de l'algorithme quand le nombre de données à traiter augmente. **Cette « forme simplifiée » n'est en fait rien d'autre que la partie croissant le plus vite avec le nombre de données, car lorsque celui-ci tend vers l'infini, c'est elle qui devient prédominante.** Cela signifie que si l'on trace le graphe de la fonction, sa forme finit par ressembler à celle de sa forme simplifiée lorsque le nombre de données à traiter devient grand.

La formulation complète de la fonction du nombre d'opérations réalisées par un algorithme n'importe donc pas tant que cela, ce qui est intéressant, c'est sa forme simplifiée. En effet, non seulement elle est plus simple (à exprimer, à manipuler et bien évidemment à retenir), mais en plus elle caractérise correctement le comportement de l'algorithme sur les grands nombres.

Un algorithme (O,V,S) est formé d'un ensemble fini O d'opérations liées par une structure de contrôle S et dont les opérandes portent sur un ensemble fini V de variables. Un algorithme résout le problème P si pour tout énoncé I de P (stocké dans le sous-ensemble des variables d'entrée de V), d'une part la suite des opérations exécutée est finie (condition de terminaison) et si d'autre part, lors de la terminaison le sous-ensemble des variables de sortie de V contient le résultat associé à l'énoncé I (condition de validité).

Prouver un algorithme, c'est démontrer mathématiquement que la propriété précédente (terminaison et validité) est satisfaite.
Une mesure de complexité d'un algorithme est une estimation de son temps de calcul, de sa mémoire utilisée ou de toute autre unité significative.
On s'intéresse le plus souvent à la recherche d'une estimation par excès à une constante positive multiplicative près du cas le plus défavorable sur le sous-ensemble des énoncés de taille fixée n (complexité dite «pire-cas»). On obtient ainsi le taux asymptotique de croissance de la mesure indépendamment de la machine sur laquelle l'algorithme est exécuté et l'on peut alors comparer les complexités pire-cas de plusieurs algorithmes résolvant un même problème.

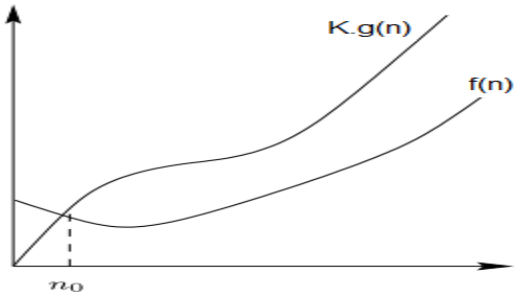
Afin de définir la notion de mesure de complexité, nous introduisons les trois notations de Landau relatives aux ensembles de fonctions **O(g),** Ω(g) et θ(g), lorsque g est une fonction de N dans N.

Définition : borne supérieure asymptotique

O(g(n))= { f: N →N | ∃ k > 0 et $n_0 \geq 0$ tels que ∀ n ≥ n_0, 0 ≤ f(n) ≤ k.g(n)}

Si une fonction *f(n) ∈ O(g(n))*, on dit que g(n) est une borne supérieure asymptotique pour f(n).

On note *f(n)=O(g(n))*

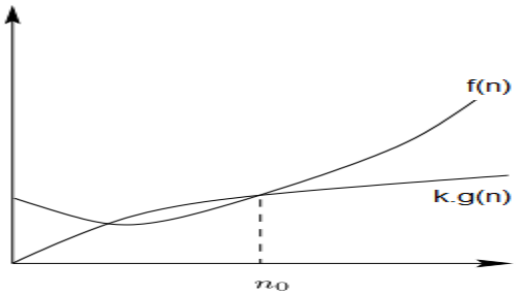


Définition : borne inférieure asymptotique

Ω (g(n))= { f: N →N | ∃ k > 0 et $n_0 \geq 0$ tels que ∀ n ≥ n_0, 0 ≤ k.g(n) ≤ f(n)}

Si une fonction *f(n) ∈ Ω (g(n))*, on dit que g(n) est une borne inférieure asymptotique pour f(n).

On note *f(n)= Ω (g(n))*

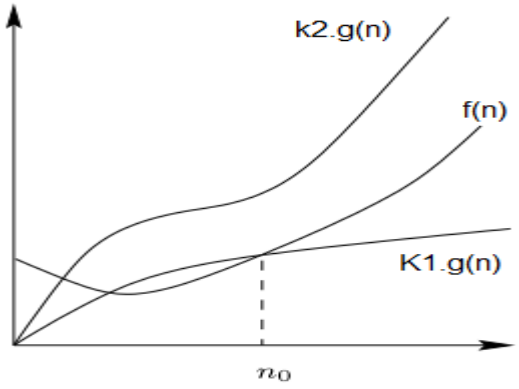


Définition : borne asymptotique

Θ (g(n))= { f: N →N | ∃ k_1 > 0, k_2>0 et $n_0 \geq 0$ tels que ∀ n ≥ n_0, 0 ≤ k_1.g(n) ≤ f(n) ≤ k2.g(n)}

Si une fonction *f(n) ∈ Θ (g(n))*, on dit que g(n) est une borne asymptotique pour f(n).

On note *f(n)= Θ (g(n))*



Interprétation pratique de la complexité

Toutes ces notions peuvent vous paraître assez abstraites, mais il est important de bien comprendre ce qu'elles signifient. Il est donc peut-être nécessaire de donner quelques exemples de complexité parmi celles que l'on rencontre le plus couramment.

Tout d'abord, une complexité de *1* pour un algorithme signifie tout simplement que son coût d'exécution est constant, quel que soit le nombre de données à traiter. Notez bien ici que l'on parle de coût d'exécution et non de durée. Le coût est ici le nombre d'opérations élémentaires effectuées par cet algorithme. Les algorithmes de complexité *1* sont évidemment les plus intéressants, mais ils sont hélas assez rares ou tout simplement triviaux.

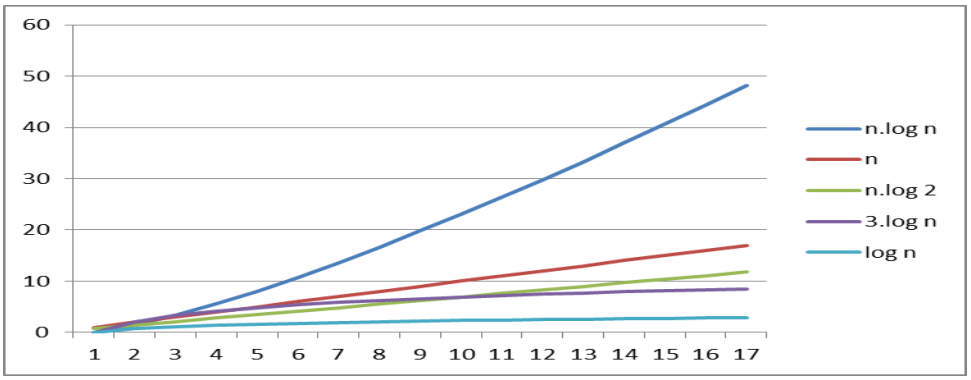
Généralement, les algorithmes ont une complexité de *n*, leur coût d'exécution est donc proportionnel au nombre de données à traiter. C'est encore une limite acceptable, et généralement acceptée comme une conséquence « logique » de l'augmentation du nombre de données à traiter. Certains algorithmes sont en revanche nettement moins performants et ont une complexité en n^2, soit le carré du nombre des éléments à traiter. Cette fois, cela signifie que leur coût d'exécution a tendance à croître très rapidement lorsqu'il y a de plus en plus de données. Par exemple, si l'on double le nombre de données, le coût d'exécution de l'algorithme ne double pas, mais quadruple. Et si l'on triple le nombre de données, ce coût devient neuf fois plus grand. Ne croyez pas pour autant que les algorithmes de ce type soient rares ou mauvais. On ne peut pas toujours, hélas, faire autrement...

Il existe même des algorithmes encore plus coûteux, qui utilisent des exposants bien supérieurs à 2. Inversement, certains algorithmes extrêmement astucieux permettent de réduire les complexités *n* ou n^2 en *ln(n)* ou *n.ln(n)*, ils sont donc nettement plus efficaces.

Note : La fonction *ln(n)* est la fonction logarithmique, qui est la fonction inverse de l'exponentielle, bien connue pour sa croissance démesurée. La fonction logarithme évolue beaucoup moins vite que son argument, en l'occurrence *n* dans notre cas, et a donc tendance à « écraser » le coût des algorithmes qui l'ont pour complexité.

Enfin, pour terminer ces quelques notions de complexité algorithmique, sachez que l'on peut évaluer la difficulté d'un problème à partir de la complexité du meilleur algorithme qui permet de le résoudre. Par exemple, il a été démontré que le tri d'un ensemble de *n* éléments ne peut pas se faire en mieux que *n×ln(n)* opérations (et on sait le faire, ce qui est sans doute le plus intéressant de l'affaire).

Malheureusement, il n'est pas toujours facile de déterminer la complexité d'un problème. Il existe même toute une classe de problèmes extrêmement difficiles à résoudre pour lesquels on ne sait même pas si leur solution optimale est polynomiale ou non. En fait, on ne sait les résoudre qu'avec des algorithmes de complexité exponentielle (si vous ne savez pas ce que cela signifie, en un mot, cela veut dire que c'est une véritable catastrophe). Cependant, cela ne veut pas forcément dire qu'on ne peut pas faire mieux, mais tout simplement qu'on n'a pas pu trouver une meilleure solution, ni même prouver qu'il y en avait une ! Toutefois, tous ces problèmes sont liés et, si on trouve une solution polynomiale pour l'un d'entre eux, on saura résoudre aussi facilement tous ses petits camarades. Ces problèmes appartiennent tous à la classe des problèmes dits « NP-complets ».



Exercices :

1)

- Démontrer que $10^{-5}n^3$ est une borne supérieure asymptotique pour n^2.
- Démontrer que n^4 est une borne supérieure asymptotique pour $25n^4-19n^3+13n^2$.
- Démontrer que 2^n est une borne supérieure asymptotique pour 2^{n+100}.

2) Sachant que si $f1(n) \leq f2(n)$ alors $O(f1(n)) \subset O(f2(n))$, donner les relations d'inclusion entre les ensembles suivants :
   $O(n.\log n)$, $O(2^n)$, $O(\log n)$, $O(1)$, $O(n^2)$, $O(n^3)$ et $O(n)$.
   (log = log népérien)

3) Montrer que si $f(n) \in O(g(n))$, alors $g(n) \in \Omega(f(n))$

4) Montrer que si $f(n) \in O(g(n))$, alors $f(n)+g(n) \in O(g(n))$

5) Montrer que $f(n)+g(n) \in \Theta(\max(f(n),g(n))$

Définition : raisonnement par récurrence

En mathématiques, le raisonnement par récurrence est une forme de raisonnement visant à démontrer une propriété portant sur tous les entiers naturels. Il consiste à démontrer les points suivants:

- La propriété est satisfaite pour l'entier 0, ou par toute autre valeur entière d'initialisation.
- Si cette propriété est satisfaite par un certain nombre entier naturel n, alors elle est satisfaite par son successeur n+1.

Une fois cela établi, on en conclut que cette propriété est vraie pour tous les nombres entiers naturels.

Application – somme des éléments d'un tableau

Soit un tableau tab de n entiers (n≥1). L'algorithme itératif calculant la somme des éléments de tab est le suivant :

```
Somme : entier
Somme ← 0
Pour i ← 1 à n faire
        Somme ← Somme+tab[i]
Fin pour
```

1) Dans un 1er temps, on veut prouver cet algorithme

Pour prouver cet algorithme, il faut 2 choses : la terminaison et la validité de l'algorithme.
La terminaison de l'algorithme est triviale puisque la boucle est exécutée n fois et que le corps de la boucle n'est constitué que d'une somme et d'une affectation, opérations qui s'exécutent en un temps fini.

Pour démontrer la validité, nous considérons la propriété suivante : à la fin de l'itération i, la variable *Somme* contient la somme des i premiers éléments du tableau. Notons $Somme_i$ la valeur de la variable *Somme* à l'itération i et $Somme_0$ =0 sa valeur initiale.
On effectue un raisonnement par récurrence sur i. La propriété est vraie pour i=1 puisqu'à l'issue de la 1ere itération *Somme* (initialisée à 0) contient la valeur tab[1].

$Somme_1 = somme_0 + tab[1] = tab[1]$

Maintenant si on suppose que la propriété est vraie pour une itération i :

$$Somme_i = \sum_{k=1}^{i} tab[k]$$

A la fin de l'itération i+1, *Somme* contiendra la valeur qu'elle contenait à la fin de l'itération i, donc la somme des i 1er éléments, plus tab(i+1] qui lui a été ajoutée à l'itération i+1.

Somme sera donc bien la somme des i+1 1er éléments de tab.

$$Somme_{i+1}=Somme_i + tab[i+1] = \sum_{k=1}^{i} tab[k]+tab[i+1] = \sum_{k=1}^{i+1} tab[k]$$

Vraie initialement, la propriété reste vraie à chaque nouvelle itération. Cette propriété est donc un invariant de l'algorithme, on parle d'**invariant de boucle**. L'algorithme se terminant à la fin de l'itération n, il aura donc bien calculé la somme des n éléments du tableau tab.

2) Détermination de la complexité de l'algorithme.

On s'intéresse au nombre a(n) d'additions effectuées par l'algorithme. De façon évidente, a(n)=n. On en déduit que l'algorithme est en O(n). L'inégalité a(n) ≤ n suffisait pour aboutir à cette conclusion. On peut aussi déduire de l'inégalité a(n) ≥ n que l'algorithme est en Ω(n) et donc globalement en Θ(n).

L'algorithme récursif permettant de faire la somme des éléments d'un tableau est donné ci-dessous.

```
fonction sommeRecursif(tab:entier; n:entier)
si n ≤ 0
        alors retourner 0
sinon
        retourner (sommeRecursif(tab,n-1)+tab[n])
fsi
```

1) Preuve

On démontre d'abord la terminaison par récurrence sur i. L'appel de la fonction *sommeRecursif* avec le paramètre 0 se termine immédiatement. Si on suppose que l'appel de la fonction avec un paramètre n se termine, l'appel de cette même fonction avec le paramètre n+1 appelle la fonction *sommeRecursif* avec le paramètre n qui par hypothèse se termine, ajoute à son résultat la valeur de tab[n] et se termine.

(Attention dans l'implémentation en langage C ci-dessous, les tableaux en C commence à l'indice 0 et non pas 1).

$Somme_0=0$

$Somme_n=somme_{n-1}+tab[n]$ pour n > 0

Dès l'instant ou la fonction se termine – ce que nous avons prouvé- elle calcule bien la somme des éléments de tab.

2) Complexité.

On appelle a(n) le nombre d'addition effectuées par la fonction appelée avec un paramètre n. a(n) est solution de l'équation de récurrence :

a(0)=0 et
a(n)=a(n-1)+1 si n > 0

La solution est a(n)=n. On en déduit à nouveau que l'algorithme est à la fois en O(n) et en Ω(n) et donc globalement en Θ(n).

Ci-dessous, une implémentation en C des 2 algorithmes précédents, on constate qu'ils ont la même complexité.

```
#include <stdio.h>
#include <stdlib.h>

int tab[5]={3,2,4,1,5};
int sommeRecursif(int tab[],int n);
int sommeIteratif(int tab[],int n);
int compt=1;
int main()
{
   int n=4;
    printf("\n\nalgo iteratif\n");     int res1=sommeIteratif(tab,n);      printf("res1=%d\n",res1);
    compt=1;
   printf("\n\nalgo recursif\n");     int res2=sommeRecursif(tab,n);     printf("res2=%d\n",res2);
   return 0;
}

int sommeRecursif(int tab[],int n) // appel recursif
{
if (n<0) // en C un tableau commence à 0
      return 0;
else
   {
   printf("compt=%d ",compt++);      printf("n=%d tab[n]=%d\n" ,n,tab[n]);
   return (sommeRecursif(tab,n-1)+tab[n]);
   }
}

int sommeIteratif(int tab[],int n) // appel recursif
{
int i;   int somme=0;
for(i=0;i<=n;i++)
{
   printf("compt=%d ",compt++);
   printf("n=%d tab[n]=%d\n" ,i,tab[i]);
   somme=somme+tab[i];
}
return somme;
}
```

```
algo iteratif
compt=1 n=0 tab[n]=3
compt=2 n=1 tab[n]=2
compt=3 n=2 tab[n]=4
compt=4 n=3 tab[n]=1
compt=5 n=4 tab[n]=5
res1=15


algo recursif
compt=1 n=4 tab[n]=5
compt=2 n=3 tab[n]=1
compt=3 n=2 tab[n]=4
compt=4 n=1 tab[n]=2
compt=5 n=0 tab[n]=3
res2=15
```

Terminaison par la méthode des compteurs de Knuth

En 1968, Knuth a suggéré d'introduire de nouvelles variables entières, appelées compteurs, qui sont augmentées de 1 à chaque passage dans le corps de l'itération. Si l'on peut montrer que la croissance des compteurs est bornée alors, non seulement on aura prouvé que l'algorithme se termine, mais on aura aussi une évaluation du nombre de fois que l'algorithme a exécuté le corps de l'itération.

Exemple : calcul du PGCD

```
Fonction PGCD-Knuth(a:Entier;b:Entier):Entier
Variablex,y:Entier
Variablei,j,n:Entier
Début
 i<-0
j<-0
n<-0
x<-a
y<-b
 TantQue(x<>y)Faire
        Si(x>y)Alors
                x<-x-y
                i<-i+1
        Sinon
                y<-y-x
                j<-j+1
        FinSi
        n<-n+1
FinTantQue
Retourner(x)
Fin
```

Démontrer que le nombre maximal n d'itérations dans le corps de la boucle est a+b-2 (pire cas).
$0 \leq n \leq a+b-2$

Implémentation en C

```
#include <stdio.h>
#include <stdlib.h>
int pgcd(int a,int b);
int main()
{
   int a,b;
   scanf("%d",&a);        scanf("%d",&b);
   printf("a=%d ",a);     printf("b=%d\n",b);
   int res=pgcd(a,b);
   printf("\npgcd(a,b)=%d\n",res);
```

```
   return 0;
}

int pgcd(int a,int b)
{
int i=0,j=0,n=0,x=a,y=b;
while(x!=y)
   {
      if (x>y)
      {        x=x-y;        i=i+1;       }
      else
      {        y=y-x;        j=j+1;      }
   n=n+1;
   }
printf("i=%d j=%d n=%d\n",i,j,n)    ;
 return x;
}
```

```
24
16
a=24 b=16
i=1 j=1 n=2

pgcd(a,b)=8



32768
1
a=32768 b=1
i=32767 j=0 n=32767

pgcd(a,b)=1
```

Complexité en fonction de 2 paramètres

Parmi les choix proposés ci-dessous, déterminer la complexité de chacun des algorithmes suivants (par rapport au nombre d'itération effectuées), où m et n sont 2 entiers positifs. Justifier !

O(min(m,n))
O(max(m,n))
O(m-n)
O(m+n)
O(m*n)
O(m/n)

Algo A

```
i ← 1  j ← 1
tant que ( i ≤ m) et (j ≤n) faire
        i ← i+1
        j ← j+1
fin tant que
```

Algo B

```
i ← 1  j ← 1
tant que ( i ≤ m) ou (j ≤n) faire
        i ← i+1
        j ← j+1
fin tant que
```

Algo C

```
i ← 1  j ← 1
tant que  j ≤ n faire
        si i ≤ m alors
                i ← i+1
        sinon
                j ← j+1
        fsi
fin tant que
```

Algo C

```
i ← 1  j ← 1
tant que  j ≤ n faire
        si i ≤ m alors
                i ← i+1
        sinon
                j ← j+1   i ← 1
        fsi
fin tant que
```

Exercices complémentaires : Fonctions F et G mystères

1) On considère une fonction récursive F d'un paramètre entier n suivante :

```
Fonction f(n:entier):entier
Si n=0 alors
        Retourner 2
Sinon
        Retourner f(n-1)*f(n-1)
Fsi
```

- Que calcule cette fonction ? Le prouver
- Déterminer la complexité de la fonction f.
- Comment améliorer cette complexité ?

2) On considère la fonction itérative G suivante :

```
Fonction g(n:entier):entier
R:entier
R ← 2
Pour i←1 à n faire
        R ← R*R
Fin pour
Retourner R
```

- Que calcule cette fonction ? Le prouver.
- Déterminer la complexité de la fonction g

3)

On considère 2 matrices carrés A et B d'ordre n. Le produit de A par B est une matrice carrée C définie par $Ci,j=\sum_{K=1}^{n} A_{i,k} \times B_{k,j}$

- Donner un algorithme calculant le produit de 2 matrices carrées de taille n.
- Coder l'algorithme pour 2 matrices carrées de taille 3 représentées sous la forme d'un tableau à 2 dimensions.
- Calculer la complexité de cet algorithme.
- Doit-on préciser dans quels cas (pire, meilleur, moyen) cette complexité est obtenue ?

4) Modifier l'algorithme précédent lorsque la matrice A est de dimension (m,n) et la matrice B de dimension (n,p).
- Quelle est alors la complexité de cet algorithme ?

5) Démontrer que $(n^2/2)-3n = \Theta(n^2)$

Etude de cas – algorithme du tri par insertion

		Coût	fois
1	Algo tri-insertion		
2	**données**		
3	A : *tableau de* n *entiers*		
4	$garde, i, j$: *entiers*		
5	**début**		
6	*lire*(A)	c_1	1
7	$j \leftarrow 2$	c_2	1
8	**tant que** $j \leqslant n$ **faire**	c_3	n
9	$garde \leftarrow A[j]$	c_4	$n-1$
10	$i \leftarrow j-1$	c_5	$n-1$
11	**tant que** $(i > 0)$ *et* $(A[i] > garde)$ **faire**	c_6	$\sum_{j=2}^{n} t_j$
12	$A[i+1] \leftarrow A[i]$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
13	$i \leftarrow i-1$	c_8	$\sum_{j=2}^{n}(t_j - 1)$
14	**fintq**		
15	$A[i+1] \leftarrow garde$	c_9	$n-1$
16	$j \leftarrow j+1$	c_{10}	$n-1$
17	**fintq**		
18	**fin**		

t_j correspond au nombre de fois où le test *(i>0) et (A[i] > garde)* est effectué. Lorsque ce test est faux, les lignes 12 et 13 ne sont pas exécutées, d'où t_j-1 pour les coûts c_7 et c_8.
c_4, c_5,c_9 et c_{10} ont un coût de n diminué de 1 car lorsque le test *j<=n* est faux, les lignes 9, 10, 15,16 ne sont pas exécutées.

On appelle T(n) le cout total de l'algorithme. On ajoute tous les coûts de c_1 à c_{10} et après rassemblement des termes constants, des termes en n, des termes en t_j et des termes en t_j-1, on obtient :

$$\begin{aligned} T(n) &= c_1 + c_2 - c_4 - c_5 - c_9 - c_{10} + (c_3 + c_4 + c_5 + c_9 + c_{10})n + c_6 \sum_{j=2}^{n} t_j + \\ &\quad (c_7 + c_8) \sum_{j=2}^{n} (t_j - 1) \\ &= \alpha + \beta n + \gamma \sum_{j=2}^{n} t_j + \delta \sum_{j=2}^{n} (t_j - 1) \end{aligned}$$

Contrairement à certains algorithmes, comme l'algorithme "somme" vu précédemment, il n'est pas possible de déterminer T(n) en fonction de n, t_j étant inconnu. En effet, pour un j fixé, il est impossible de savoir exactement combien de fois est exécuté le test (lignes 11, 12, 13), car ça dépend de l'ordre initial des éléments dans le tableau et de la position dans le tableau de l'élément entrain d'être traité.
Il va donc falloir "encadrer" T(n).

1) Démontrer que le cas favorable est : $T_{\min}(n) = \alpha - \gamma + (\beta + \gamma)n.$
2) Démontrer que le cas défavorable est : $T_{\max}(n) = \alpha - \gamma + (\beta + \gamma/2 - \delta/2)n + \frac{\gamma+\delta}{2} n^2$
3) Démontrer que $T_{max}(n)$ est en $O(n^2)$.
4) Démontrer que $T_{max}(n)$ est aussi en $\Theta(n^2)$.
5) On ne s'intéresse maintenant qu'au nombre de boucles a(n), dont le corps est exécuté.

- Déterminer les pire et meilleur cas pour a(n).

- Implémenter en langage C l'algorithme en ajoutant des compteurs de Knuth qui valident les meilleur et pire cas pour a(n).

6) Déterminer la complexité de cet algorithme

```
x,y,n:entier
X ← 0
Y ← 0
Pour k allant de 1 à n faire
        x ←x+1
        pour  j allant de 1 à n faire
                y ← y+1
                j ← j+1
        fin pour
k ← k*2
fin pour
```

Implémentation en langage C

```
int x = 0 ,y=0, j, k, n;
n=10; // par exemple
for (k=1; k<=n; k=k*2)
        {
        x++;
        for (j=1; j<=n; j++)
                y++;
        }
```