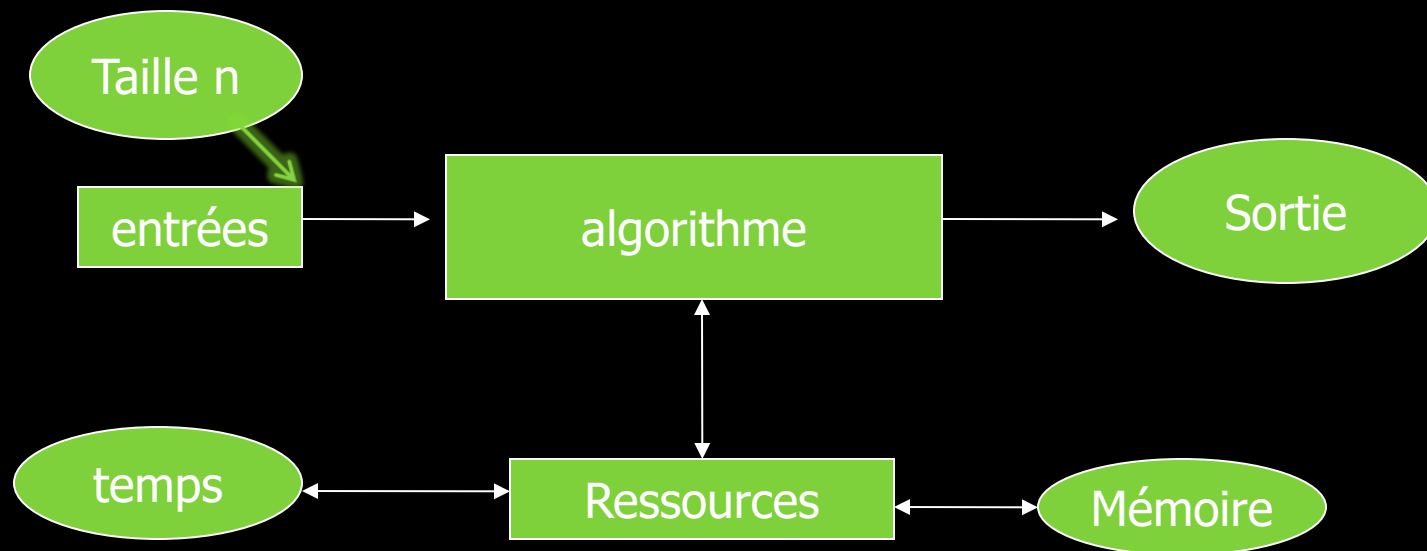


COMPLEXITE DES ALGORITHMES

- Introduction aux algorithmes
- Complexité des algorithmes
- Interprétation pratique de la complexité

**ALGORITHME= ENSEMBLE FINI D'ACTIONS
POUR ABOUTIR À UN RÉSULTAT D'UN
PROBLÈME DONNÉ.**



- Un algorithme est un ensemble d'instructions permettant de transformer un ensemble de données en un ensemble de résultats, en un nombre fini étapes.
- Pour atteindre cet objectif, un algorithme utilise deux ressources d'une machine: le temps et l'espace mémoire.



Caractéristiques


- Une sortie garantie (doit fournir un résultat)
- Simple à comprendre,
- Simple à mettre en œuvre.
- Une efficacité (exécution rapide).

REM: si la taille des données à traiter est petite,
la première caractéristique est la plus
importante, sinon la seconde.

- la complexité **temporelle** d'un algorithme est le temps mis par ce dernier pour transformer les données du problème considéré en un ensemble de résultats.
- la complexité **spatiale** d'un algorithme est l'espace utilisé par ce dernier pour transformer les données du problème considéré en un ensemble de résultats.



Temps d'exécution dépend de(s) :

- Données du problème nécessaire pour l'exécution.
 - l'efficacité de l'algorithme.
 - Rapidité des instructions offertes par le microprocesseur.
 - compilateur(qualité du code).
 - Qualité de la programmation.
- 




Temps d'exécution =

nombre d'instructions exécutées

X nombre moyen de cycles/instruction


X durée d'un cycle d'horloge

*Conclusion: un algorithme exécuté il y'a 20 ans,
n'a pas la même durée d'exécution qu'avec les
machines actuelles.(évolution technologique)*





Complexité

- Définition
 - Pourquoi la complexité?
 - Comment la calculer?
 - Différents types de complexité.
- 




La complexité



- La complexité vise à savoir si la solution à un problème peut être donnée:
 - très efficacement
 - efficacement
 - inatteignable en pratique et/ou en théorie.

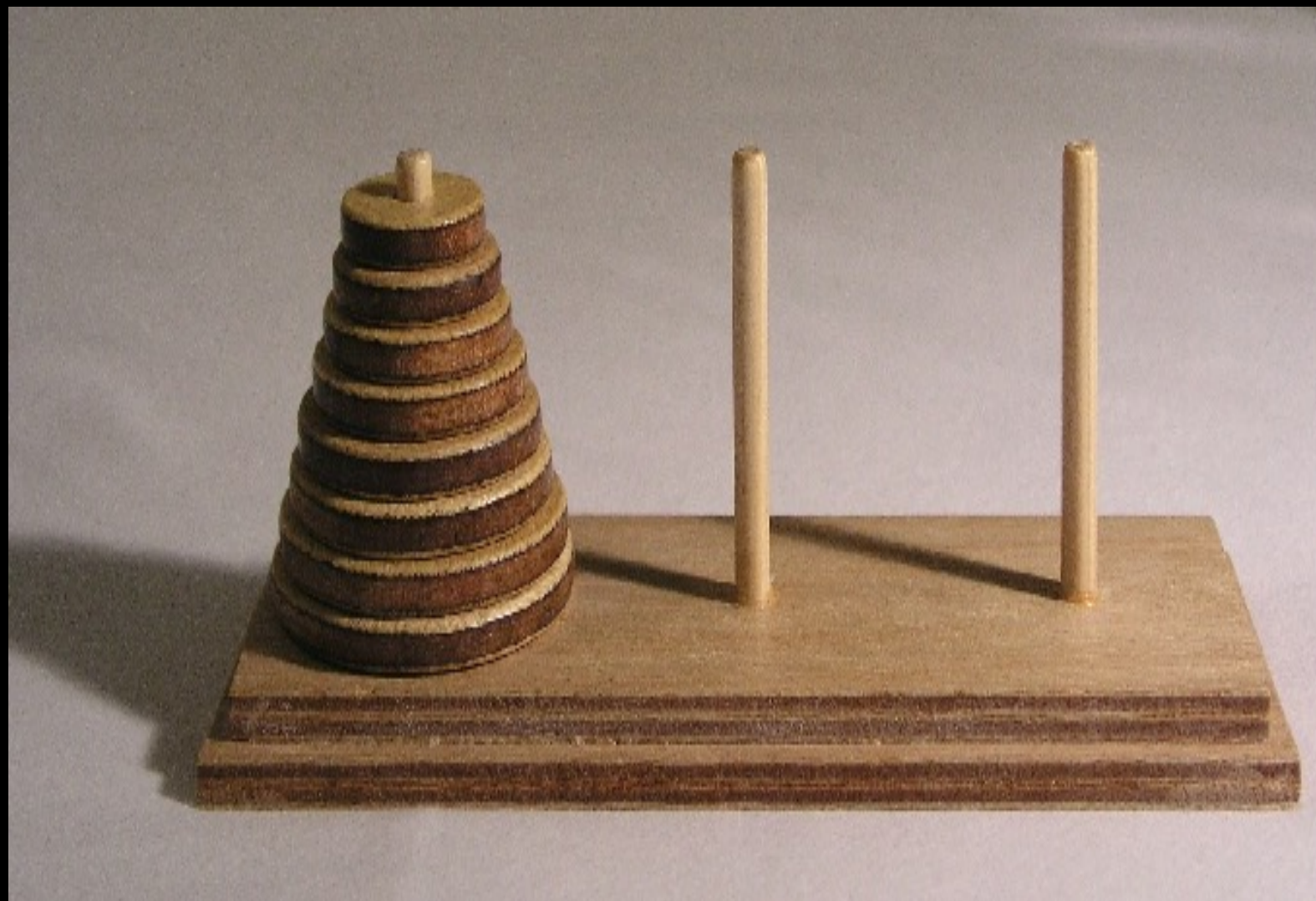
C'est la difficulté d'un algorithme à donner une réponse à un problème.



Pourquoi la complexité?

- Trouver un algorithme qui fournir une solution à un problème est une chose qui est un peu difficile.
 - Le plus difficile est de trouver l'algo qui fournir la solution la plus efficace « rapide ».
 - L'étude de la complexité nous permet de faire un choix entre plusieurs algorithmes.
- 

- 
- Plus un algorithme a des données à traiter plus il prend du temps à s'exécuter (évident!!)
 - on peut avoir des algorithmes qui s'exécutent sur des années voir des siècles (Tour de Hanoi)
- 





Déplacer (N, A, B, C) {

si $N > 0$ alors {

Déplacer (N - 1, A, C, B);

Bouger-un-disque (A, B);

Déplacer (N - 1, C, B, A);

}

}

Comment la calculer ?

Temps d'exécution =

nombre d'instructions exécutées

X nombre moyen de cycles/instruction

X durée d'un cycle d'horloge

- Le 2^{ième} et le 3^{ième} paramètres sont instables et dépendent de l'évolution technologique (process, temps d'accès à la mémoire, jeu d'instructions offertes par le processeur,)




Le temps d'exécution est fonction de N:

$$T = C * f(N)$$

C = facteur technologique.

Le calcul de $f(N)$ n'est pas aussi facile; on utilise une notation mathématique appelée « grand O » en se basant sur les opérations dites fondamentales, et chacune est exécutée en $O(1)$.



- Séquence d'instructions:

Début

$l_1;$ $O(1)$

$l_2;$ $O(1)$

$l_3;$ $O(1)$

.

.

donc la
complexité
 $= O(1)$



Fin

- Instruction conditionnelle:

SI condition
|
Alors E1
Sinon E2
FSI

La complexité de l'instruction conditionnelle est le maximum entre E1 et E2



Instruction itérative:

Les boucles itératives ont une complexité qui dépend directement la taille N : $O(n)$

Simplification pour la notation Grand O:

$$O(1) + O(1) + O(1) = O(1)$$

$$O(N+1) = O(N)$$

$$O(N*N + N + 4) = O(N*N)$$

Complexité (10^6 inst/sec)

	10	20	30	40	50	60
N	0.00001	0.00002	0.00003	0.00004	0.00005	0.00006
N^2	0.0001	0.0004	0.0009	0.0016	0.0025	0.0036
N^3	0.001	0.008	0.027	0.064	0.125	0.216
N^5	1	3.2	24.3	1.7mn	5.2mn	13.0mn
2^n	0.001	1.0	17.9mn	12.7j	35.7 ans	366 siècles
3^n	0.059	58mn	6.5 ans	3855 siècle	/	/

Types de complexité

Complexité logarithmique	$O(\log(n))$	
Complexité linéaire	$O(n)$	
Complexité quasi-linéaire	$O(n \log(n))$	
Complexité polynomiale	$O(N^k)$ ou $k > 1$	
Complexité exponentielle	$O(a^n)$ ou $a > 1$	Attention!!!
Complexité factorielle	$O(n!)$ ou pire $O(n^n)$	Révisez votre algorithme

Quelques exemples

- Déterminer la complexité de chaque algorithme?

1) Fonction Ycal(x){

Y = X + 2.5;


afficher y;

}

Réponse : $O(1)$


- Fonction `inial`(tableau de n éléments){
 /*pour chaque indice l du tableau*/
 tableau (l) = 0 ;
}

Réponse : $O(n)$




- Fonction initial(matrice[n,n]){
 pour i de 1 à n faire
 pour j de 1 à n faire
 matrice(i,j) = 1 ;
 }

Réponse : $O(N^2)$



- Fonction $A(n)$
total = 0;
pour k de 1 à n {
 afficher 1;
 pour l de 1 à n
 pour J de 1 à n
 total = l+J+K;
 }
}

Réponse : $O(n^3)$




- Fonction recherche(x){
trouve = faux ;
TQ I <= n FAIRE
 SI T[i] = X
 ALORS trouve = vrai;
 FSI
FTQ
 afficher trouve;
}

Réponse : $O(n)$



Pour comparer les solutions, plusieurs critères peuvent être pris en considération

- Exactitude des programmes (prouver que le résultat de l'implantation est celui escompté)
 - Simplicité des programmes
- 



Comparaison de solutions

Pour comparer des solutions entre-elles,
deux méthodes peuvent être utilisées:

1) Étude empirique: (exécuter le
programme)



2) Analyse mathématique



Facteurs affectant le temps d'exécution:

- machine,
- langage,
- programmeur,
- compilateur,
- algorithme et structure de données.

Le temps d'exécution dépend de la longueur de l'entrée.

Ce temps est une fonction $T(n)$ où n est la longueur des données d'entrée.



Exemple I:

`x=3;`

la longueur des données dans ce cas
est limitée à une seule variable.

Exemple II:


`sum = 0;`

`for (i=0; i<n; i++)`


`for (j=0; j<n; j++)`



`sum++;`


Dans ce cas, elle est fonction du paramètre `n`



La longueur des données d'entrée,
définissant le problème considéré, est
définie comme étant l'espace qu'elle
occupe en mémoire.




- 
- 
- Convergence et stabilité des programmes
 - (solutions qui convergent vers la solution exacte; la perturbation des données ne change pas d'une manière drastique la solution obtenue)
 - Efficacité des programmes (Temps et Espace)




Pire cas, meilleur cas et cas moyen

Toutes les entrées d'une longueur donnée ne nécessitent pas nécessairement le même temps d'exécution.





Exemple: soit à rechercher un élément C dans un tableau de n élément triés dans un ordre croissant.



Déterminer la complexité de l'algorithme de recherche d'un élément dans un tableau initialement trié par la méthode séquentielle et dichotomique.

Considérons les solutions suivantes:



- 
- 
1. **Recherche séquentielle** dans un tableau de taille n . Commencer au début du tableau et considérer chaque élément jusqu'à ce que l'élément cherché soit trouvé.
 2. **Recherche dichotomique**: tient compte du fait que les éléments du tableau sont déjà triés. Information ignorée par l'algorithme de la recherche séquentielle.

Recherche Séquentielle

```
int recherche1(int *tab, int C){  
    int i;  
    i = 0;  
    while (i < n && tab[i] != C )  
        i ++;  
    if (i == n)  
        return(-1);  
    else return(i);  
} /* fin de la fonction */
```

Recherche Dichotomique

```
1. int recherche2(int *tab, int C){
2.     int sup, inf, milieu;
3.     bool trouve;
4.     inf = 0; sup = n-1; trouve = false;
5.     while (sup >= inf && !trouve) {
6.         milieu = (inf + sup) / 2;
7.         if (C == tab[milieu])
8.             trouve = true;
9.         else if (C < tab[milieu])
10.            sup = milieu - 1;
11.        else inf = milieu + 1;
12.     if (!trouve)
13.         return(-1);
14.     return(milieu)
15. } /* fin de la fonction */
```




La méthode empirique

- Elle consiste à coder et exécuter deux (ou plusieurs) algorithmes sur une batterie de données générées d'une manière aléatoire;
- À chaque exécution, le temps d'exécution de chacun des algorithmes est mesuré.
- Ensuite, une étude statistique est entreprise pour choisir le meilleur d'entre-eux à la lumière des résultats obtenus.




Problème!


Ces résultats dépendent


- de la machine utilisée
 - du jeu d'instructions utilisées
 - de l'habileté du programmeur
 - du jeu de données générées
 - du compilateur choisi
 - de l'environnement dans lequel est exécuté les deux
 - algorithmes (partagé ou non)
- etc.
- 



Méthode mathématique

- Pour pallier à ces problèmes, une notion de complexité plus simple mais efficace a été proposée par les informaticiens.
 - Ainsi, pour mesurer cette complexité, la méthode mathématique, consiste non pas à la mesurer en unité de temps (par exemple les secondes), mais à faire le décompte des instructions de base exécutées par ces deux algorithmes.
- 

- 
- Cette manière de procéder est justifiée par le fait que la complexité d'un algorithme est en grande partie induite par l'exécution des instructions qui le composent.



Pour avoir une idée plus précise de la performance d'un algorithme, il convient de signaler que la méthode expérimentale et mathématique sont en fait complémentaires.






Comment choisir entre plusieurs solutions?

Décompte des instructions


Reconsidérons la solution 1 (recherche séquentielle) et faisons le décompte des instructions.

Limitons nous aux instructions suivantes:

- 
- Affectation notée par « e »
 - Test noté par « t »
 - Addition notée par « a »

- 
- 
- ce décompte dépend de la valeur « C » mais aussi de celles des éléments du tableau.
 - il y a lieu de distinguer trois mesures de complexité:
 - 1. dans le meilleur cas
 - 2. dans le pire cas
 - 3. dans la cas moyen


- Meilleur cas: notée par $t_{\min}(n)$ représentant la complexité de l'algorithme dans le meilleur des cas en fonction du paramètre n (ici le nombre d'éléments dans le tableau).
- Pire cas: notée par $t_{\max}(n)$ représentant la complexité de l'algorithme dans le pire cas en fonction du paramètre n (ici le nombre d'éléments dans le tableau).

- 
- Cas Moyen: notée par $t_{moy}(n)$ représentant la complexité de l'algorithme dans le cas moyen en fonction du paramètre n (ici le nombre d'éléments dans le tableau).

C'est-à-dire la moyenne de toutes les complexités, $t(i)$, pouvant apparaître pour tout ensemble de données de taille n






$t(i)$ représente donc la complexité de l'algorithme





dans le cas où C se trouve en position i du tableau).


Dans le cas où l'on connaît la probabilité P_i de réalisation de la valeur $t(i)$, alors par définition, on a: (Espérance mathématique)


$$t_{\text{moy}}(n) = \sum_{i=1}^n p_i \cdot t(i)$$


- 
- Pour certains algorithmes, il n'y a pas lieu de distinguer entre ces trois mesures de complexité. Cela n'a pas vraiment de sens.
- 




```
int recherche1(int *tab, int C){  
    int i;  
    i = 0;  
    while (i<n && tab[i] != C )  
        i ++;  
    if (i == n)  
        return(-1);  
    else return(i);  
} /* fin de la fonction */
```



Meilleur cas pour la recherche séquentielle:
Le cas favorable se présente quand la valeur
C se trouve au début du tableau




$t_{\min}(n) = e + 3t$ (une seule affectation et 3
test: deux tests dans la boucle et un autre à
l'extérieur de la boucle)



Pire cas: Le cas défavorable se présente quand la valeur C ne se trouve pas du tout dans le tableau.

Dans ce cas, l'algorithme aura à examiner, en vain, tous les éléments.



$$\begin{aligned} t_{\max}(n) &= 1e + n(2t+1e+1a) + 1t + 1t \\ &= (n+1)e + na + (2n+2)t \end{aligned}$$




Cas moyen: Comme les complexités favorable
et défavorable sont respectivement

$(e + 3t)$ et


$(n+1)e + na + (2n+3)t,$




la complexité dans le cas moyen va
se situer entre ces deux valeurs. Son calcul
se fait comme suit:



Pour plus de simplicité, on suppose que C existe dans le tableau. On suppose aussi que sa probabilité de présence dans l'une des positions de ce tableau est de $1/n$.





Si C est dans la position i du tableau, la complexité $t(i)$ de l'algorithme est:

$$t(i) = (i+1)e + ia + (2i+2)t$$

Par conséquent, la complexité moyenne de notre algorithme est :

$$T_{\text{moy}}(n) = \sum_{i=0}^n \frac{1}{n} (i+1)e + ia + (2i+2)t$$


$$= (3n+1)e/2 + (n+1)a/2 + n(n+4)t$$





Complexité asymptotique:


Le décompte d'instructions peut s'avérer fastidieux à effectuer si on tient compte d'autres instructions telles que:

- accès à un tableau,
- E/S, opérations logiques,
- appels de fonctions,.. etc.




De plus, même en se limitant à une seule opération, dans certains cas, ce décompte peut engendrer des expressions que seule une approximation peut conduire à une solution.



- 
- 
- Même si les opérations élémentaires ont des temps d'exécution constants sur une machine donnée, ils sont différents d'une machine à une autre.
 - Pour ne retenir que les caractéristiques essentielles, et rendre son calcul simple (mais indicatif!), il est légitime d'ignorer toute constante pouvant apparaître lors du décompte du nombre de fois qu'une instruction est exécutée.
 - Le résultat obtenu à l'aide de ces simplifications représente la complexité asymptotique de l'algorithme considéré.



Ainsi, si $t_{\max}(n) = (n+1)e + (n-1)a + (2n+1)t$,
alors on dira que la complexité de cet algorithme
est tout simplement en n .

On a éliminé tout constante, et on a supposé aussi
que les opérations d'affectation, de test et
d'addition ont des temps constants.




- 
- 
- La complexité asymptotique d'un algorithme décrit le comportement de celui-ci quand la taille n des données du problème traité devient de plus en plus grande, plutôt qu'une mesure exacte du temps d'exécution
 - Une notation mathématique, permettant de représenter cette façon de procéder, est décrite dans ce qui suit:



Notation grand-O

La notation grand-O indique une borne supérieure sur le temps d'exécution.

Exemple: Si $T(n) = 3n^2 + 2$
alors $T(n) = O(n^2)$.



Grand-O: Exemples

Exemple 1: Initialiser un tableau d'entiers

```
for (int i=0; i<n; i++) Tab[i]=0;
```

Il y a n itérations,

chaque itération nécessite un temps constant c , où c est une constante (accès au tableau + une affectation). Le temps est donc

$$T(n) = cn$$

$$T(n) = O(n)$$

Grand-O: Exemples

Exemple 2: $T(n) = c_1 n^2 + c_2 n$.

$$c_1 n^2 + c_2 n \leq c_1 n^2 + c_2 n^2 \leq (c_1 + c_2) n^2$$

pour tout $n \geq 1$.

$$T(n) \leq c n^2 \text{ où } c = c_1 + c_2 \text{ et } n_0 = 1.$$

Donc, $T(n) = O(n^2)$.

Exemple 3: $T(n) = c$. On écrit $T(n) = O(1)$.

Grand-Omega Ω

Définition: Soit $T(n)$, une fonction non négative.

On a $T(n) = \Omega(g(n))$ s'il existe deux constantes positives c et n_0 telles que $T(n) \geq cg(n)$ pour tout $n > n_0$.

Signification: Pour de grandes entrées, l'exécution de l'algorithme nécessite au moins $cg(n)$ étapes. \Rightarrow Borne inférieure.

Grand-Omega: Exemple

$$T(n) = c_1 n^2 + c_2 n .$$

$$c_1 n^2 + c_2 n \geq c_1 n^2 \text{ pour tout } n > 1.$$

$$T(n) \geq c n^2 \text{ pour } c = c_1 \text{ et } n_0 = 1.$$

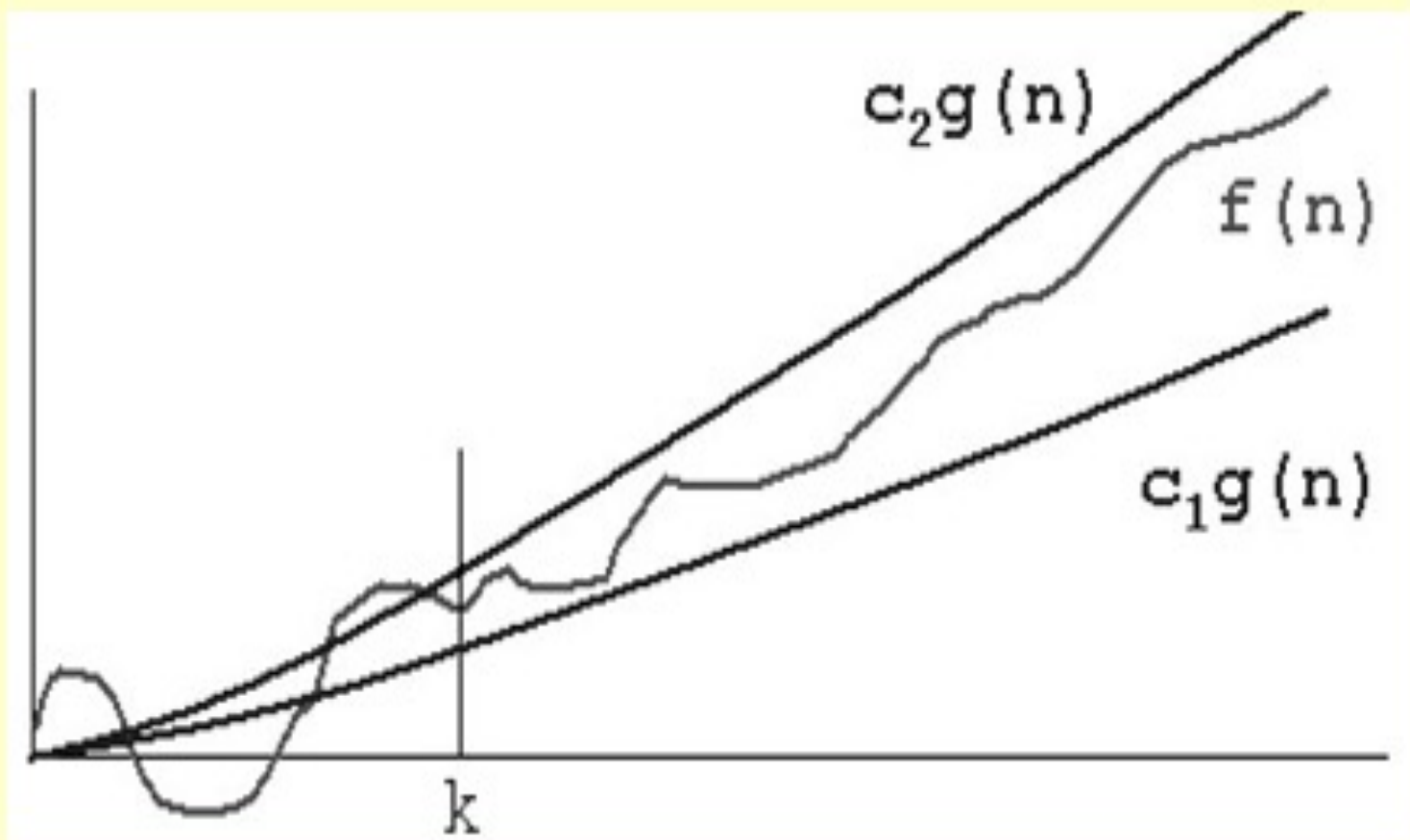
Ainsi, $T(n) = \Omega(n^2)$ par définition.

C'est la plus grande borne inférieure qui est recherchée.

La notation Theta Θ

Lorsque le grand-O et le grand-omega d'une fonction coïncident, on utilise alors la notation grand-theta.

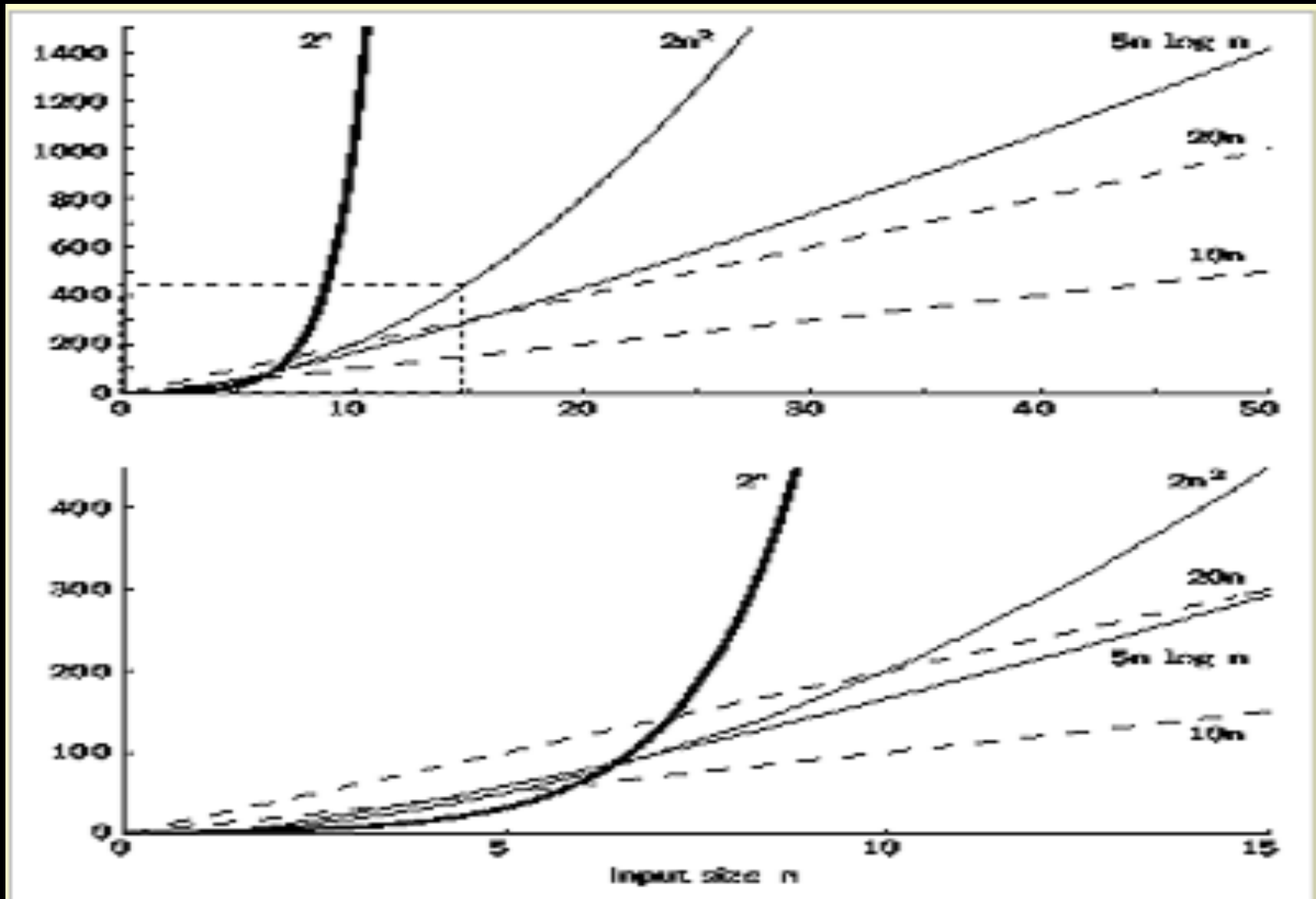
Définition: Le temps d'exécution d'un algorithme est dans $\Theta(h(n))$ s'il est à la fois dans $O(h(n))$ et dans $\Omega(h(n))$



Exemple

	n = 10	n = 1000	n = 100000	
$\Theta(n)$	n	10	1000	10^5
	10n	100	10^4	10^6
	100n	1000	10^5	10^7
				} secondes
$\Theta(n^2)$	n²	100	10^6	10^{10}
	10n²	1000	10^7	10^{11}
	100n²	10^4	10^8	10^{12}
				} heures
$\Theta(n^3)$	n³	1000	10^9	10^{15}
	10n³	10^4	10^{10}	10^{16}
	100n³	10^5	10^{11}	10^{17}
				} années
$\Theta(2^n)$	2ⁿ	1024	$> 10^{301}$	∞
	10 · 2ⁿ	$> 10^4$	$> 10^{302}$	∞
	100 · 2ⁿ	$> 10^5$	$> 10^{303}$	∞
$\Theta(\lg n)$	lg n	3.3	9.9	16.6
	10 lg n	33.2	99.6	166.1
	100 lg n	332.2	996.5	1661.0

Taux de croissance





Règles de simplification 1

Si $f(n) = O(g(n))$ et $g(n) = O(h(n))$,
alors $f(n) = O(h(n))$.

La notation O est transitive



Règles de simplification 2

Si $f(n) = O(kg(n))$ où $k > 0$, une constante
alors $f(n) = O(g(n))$.

Les constantes sont ignorées

Règles de simplification 3

Si $f_1(n) = O(g_1(n))$ et $f_2(n) = O(g_2(n))$,
alors

$$(f_1 + f_2)(n) = O(\max(g_1(n), g_2(n)))$$

$$(f_1 + f_2)(n) = O(g_1(n) + g_2(n))$$

Règles de simplification 4


Si $f_1(n) = O(g_1(n))$ Et $f_2(n) = O(g_2(n))$

alors


$$f_1(n)f_2(n) = O(g_1(n) g_2(n))$$

Règles pour dériver la complexité d'un algorithme

- Règle 1: la complexité d'un ensemble d'instructions est la somme des complexités de chacune d'elles.
- Règle 2: Les opérations élémentaires telles que l'affectation, test, accès à un tableau, opérations logiques et arithmétiques, lecture ou écriture d'une variable simple ... etc, sont en $O(1)$ (ou en $\Theta(1)$)



Règle 3: Instruction if: maximum entre
le bloc d'instructions de then et celui
de else




switch: prendre le maximum parmi les
complexités des blocs d'instructions des
différents cas de cette instruction




Règle 4: Instructions de répétition:

1. La complexité de la boucle for est calculée par la complexité du corps de cette boucle multipliée par le nombre de fois qu'elle est répétée.
1. En règle générale, pour déterminer la complexité d'une boucle while, il faudra avant tout déterminer le nombre de fois que cette boucle est répétée, ensuite le multiplier par la complexité du corps de cette boucle.




Règle 5: Procédure et fonction: le complexité est déterminée par celui de leur corps. Notons qu'on fait la distinction entre les fonctions récursives et celles qui ne le sont pas

- 
- Dans le cas de la récursivité, le temps de calcul est exprimé comme une relation de récurrence.



Pour les fonctions non récurives, leur complexité temporelle se calcule en sachant que l'appel à une fonction prend un temps constant en $O(1)$ (ou en $\Theta(1)$)



Exemple 1: $a = b;$

Temps constant: $\Theta(1)$.

Exemple 2:

```
somme = 0;  
for (i=1; i<=n; i++)  
    somme += n;
```

Temps : $\Theta(n)$

Exemple 3:

```
somme = 0;  
for (j=1; j<=n; j++)  
    for (i=1; i<=n; i++)  
        somme++;  
for (k=0; k<n; k++)  
    A[k] = k;
```

Temps : $\Theta(1) + \Theta(n^2) + \Theta(n) = \Theta(n^2)$

Example 4:

```
somme = 0;  
for (i=1; i<=n; i++)  
    for (j=1; j<=i; j++)  
        somme++;
```

Temps: $\Theta(1) + O(n^2) = O(n^2)$

On peut montrer aussi: $\Theta(n^2)$


Example 5:

```
somme = 0;  
for (k=1; k<=n; k*=2)  
    for (j=1; j<=n; j++)  
        somme++;
```

Temps: $\Theta(n \log n)$ pourquoi donc?




Efficacité des algorithmes

- Définition: Un algorithme est dit efficace si sa complexité (temporelle) asymptotique est dans $O(P(n))$ où $P(n)$ est un polynôme et n la taille des données du problème considéré.
- 



Définition: On dit qu'un algorithme A est meilleur qu'un algorithme B si et seulement si:

$$t_A(n) \leq O(t_B(n)); \text{ et} \\ t_B(n) \geq O(t_A(n))$$



Où $t_A(n)$ et $t_B(n)$ sont les complexités des algorithmes A et B, respectivement

Résumé Première partie: Type de complexité algorithmique


- On considère désormais un algorithme dont le temps maximal d'exécution pour une donnée de taille n en entrée est noté $T(n)$.
- Chercher la complexité au pire – dans la situation la plus défavorable – c'est exprimer $T(n)$ en général en notation O .






$T(n) = O(1)$, temps constant : temps d'exécution indépendant de la taille des données à traiter.

$T(n) = O(\log(n))$, temps logarithmique :

on rencontre généralement une telle complexité lorsque l'algorithme casse un gros problème en plusieurs petits, de sorte que la résolution d'un seul de ces problèmes conduit à la solution du problème initial.



- 
- 
- $T(n) = O(n)$, temps linéaire : cette complexité est généralement obtenue lorsqu'un travail en temps constant est effectué sur chaque donnée en entrée.
 - $T(n) = O(n \cdot \log(n))$: l'algorithme scinde le problème en plusieurs sous-problèmes plus petits qui sont résolus de manière indépendante. La résolution de l'ensemble de ces problèmes plus petits apporte la solution du problème initial.




$T(n) = O(n^2)$, temps quadratique : apparaît notamment lorsque l'algorithme envisage toutes les paires de données parmi les n entrées (ex. deux boucles imbriquées)

$O(n^3)$ temps cubique

$T(n) = O(2^n)$, temps exponentiel : souvent le résultat de recherche brutale d'une solution.



- 
- L'analyse de la complexité en place mémoire revient à évaluer, en fonction de la taille de la donnée, la place mémoire nécessaire pour l'exécution de l'algorithme, en dehors de l'espace occupé par les instructions du programme et des données.

Conclusion : Il s'agit donc d'obtenir le meilleur compromis espace-temps. Complexité en place mémoire

Compter le nombre d'opérations do-it() exécutées par chacun des algorithmes suivants.

(1) pour $i = 1$ à n faire

(2) pour $j = 1$ à n faire

(3) do-it()

La boucle (1) s'exécute
 n fois et elle effectue
donc $n \times n = n^2$ do-it().

Compter le nombre d'opérations do-it()
exécutées par chacun des algorithmes suivants.

La boucle (2) s'exécute i fois et elle effectue donc i do-it.

- (1) pour i = 1 à n faire
 - (2) pour j = 1 à i faire
 - (3) do-it()

La boucle(1) s'exécute n fois,
Pour i allant de 1 à n.
Elle effectue donc

$$\sum_i^n i = \frac{n(n+1)}{2} \text{ do-it.}$$

Compter le nombre d'opérations do-it()
exécutées par chacun des algorithmes suivants.

(1) pour i de 5 à n-5 faire

(2) pour j de i-5 à i+5 faire

(3) do-it()

le nombre d'itérations de la boucle interne ne
dépend pas de i.

La boucle (2) s'exécute $(i + 5) - (i - 5) + 1 = 11$
fois et elle effectue donc 11 do-it.

La boucle (1) s'exécute $(n - 5) - 4 = n - 9$ fois et
elle effectue donc $11(n - 9) = 11n - 99$

Do-it.

Compter le nombre d'opérations do-it()
exécutées par chacun des algorithmes suivants.

pour $i = 1$ à n faire

(2) pour $j = 1$ à i faire

(3) pour $k = 1$ à j faire

(4) do-it()

La boucle (3) s'exécute j fois et elle effectue donc j do-it().

La boucle (2) s'exécute i fois, pour j allant de 1 à i. Elle effectue donc

$$\sum_{j=1}^i j = \frac{i(i+1)}{2} = \frac{i^2}{2} + \frac{i}{2}$$

Do-it().

La boucle (1) s'exécute n fois, pour i allant de 1 à n. Elle effectue donc

$$\sum_{i=1}^n \frac{i^2}{2} + \frac{i}{2} = \frac{1}{2} \sum_{i=1}^n i^2 + \frac{1}{2} \sum_{i=1}^n i =$$
$$\frac{1}{2} \frac{n(n+1)}{2} + \frac{1}{2} \frac{n(n+1)(2n+1)}{6}$$

do-it() . Expression en $O(n^3)$.

Fiche récapitulative : Complexités algorithmiques

Complexité	Description	Exemple
$O(1)$	Constante	Accès à un élément de tableau
$O(\log n)$	Logarithmique	Recherche dichotomique
$O(n)$	Linéaire	Parcours d'un tableau
$O(n \log n)$	Quasi-linéaire	Tri fusion, Tri rapide (moyenne)
$O(n^2)$	Quadratique	Tri à bulles, double boucle
$O(n^3)$	Cubique	Multiplication de matrices (naïve)
$O(2^n)$	Exponentielle	Problème du sac à dos (naïf)
$O(n!)$	Factorielle	Voyageur de commerce (brute force)

Graphique comparatif

