Calculabilité, logique et complexité

Chapitre 6

Introduction à la complexité algorithmique

Chapitre 6 : Introduction à la complexité algorithmique

1. Notions de complexité
2. Calcul de complexité
3. Exemple
4. Notation “grand O”
5. ~~Exemple de calcul de complexité~~
6. Hiérarchie de complexités
7. Problèmes intrinsèquement complexes

Acquis d'apprentissage

A l'issue de ce chapitre, les étudiants seront capables de

- Expliquer et illustrer les concepts de complexité spaciale et temporelle d'un algorithme
- Comprendre et expliquer les concepts de complexité spaciale et temporelle d'un problème
- Expliquer le concept de complexité dans le pire des cas
- Comprendre, expliquer et justifier la notation grand O
- Comprendre, expliquer, manipuler et appliquer la définition de O(f) ainsi que ses propriétés
- Illustrer la hiérarchie de complexités
- Expliquer et justifier les concepts de problème pratiquement faisable, pratiquement infaisable (intrinsèquement complexe)
- Expliquer et justifier l'influence potentielle du modèle de calcul et de la représentation des données sur la complexité d'un problème

1. Notions de complexité

On doit exiger qu'un programme soit **correct** par rapport à sa spécification

On peut exiger qu'un programme correct soit **efficace**

Efficacité : mesure des ressources nécessaires à mettre en œuvre pour que le programme produise les résultats attendus

Complexité : “prix” à payer pour résoudre le problème

Différence entre

- théoriquement calculable (**calculabilité**)
- et pratiquement calculable (**complexité**)

un problème calculable peut être pratiquement infaisable...

1. Notions de complexité

Exemples de ressources

- **CPU**

Mesure : nombre d'opérations à effectuer; temps d'exécution

- **Mémoire**

Mesure : nombre de cellules mémoires nécessaires (RAM, disque)

- **Entrée/sortie**

Mesure : nombre de transferts depuis/vers le disque par unité de temps

- **Temps de développement, maintenance**

Mesure : peu mesurable…

1. Notions de complexité

Définition

Mesure de l'efficacité d'un programme pour un type de ressources

- Complexité **spatiale** (espace mémoire)
- Complexité **temporelle** (temps d'exécution)

En pratique, complexité temporelle souvent plus importante que complexité spatiale

1. Notions de complexité

Algorithme vs problème

Etant donné un algorithme, on peut étudier sa complexité

Etant donné un problème, comment étudier sa complexité ?

Complexité d'un problème : complexité d'un algorithme résolvant le problème

- existence de plusieurs algorithmes résolvant le problème ?
- algorithmes de complexités différentes ?

Définition

Complexité d'un problème : complexité de l'algorithme ***le plus efficace*** résolvant le problème

1. Notions de complexité

Questions

- Comment **définir** la complexité d'un algorithme ?
- Comment **mesurer** la complexité d'un algorithme ?
- Comment **comparer** la complexité d'algorithmes ?
- Comment **déterminer** la complexité d'un *problème* ?

Complexité dépend de

- la "taille" des données
- de la représentation des données
- du modèle de calcul utilisé

Types de mesures

- pire des cas (worst case)
- meilleur des cas (best case)
- moyenne (average case)

2. Calcul de complexité

Complexité d'un algorithme

- Jeux d'essais (benchmarks)
- Analyse d'algorithmes

Complexité d'un problème

- Complexité d'un algorithme résolvant le problème
- Techniques de réduction

2. Calcul de complexité

Jeux d'essais

- Mesure du temps d'exécution
  - de programmes donnés
  - sur des machines données
- permettent une comparaison
  - de programmes différents sur la même machine
  - d'un programme sur des machines différentes
- Un programme peut avoir des complexités différentes sur des données différentes, mais de même taille (e.g. programme de tri)

⇒ **Hypothèse** implicite

les jeux d'essais sont représentatifs de l'éventail des données réelles possibles

2. Calcul de complexité

Analyse d'algorithmes

- Exprimer la complexité du programme comme une fonction de la taille des données
- Trouver une fonction ***T(n)*** exprimant la complexité du programme pour une donnée de taille n
- ***T(n)*** : **temps d'exécution** du programme
- ***T(n)*** obtenu par analyse du texte du programme

2. Calcul de complexité

Réduction

Certains problèmes peuvent être résolus si l'on dispose d'un algorithme résolvant un **autre** problème

Exemple

Problème 1: trouver le maximum d'une suite de nombres

Problème 2: trier en ordre décroissant une suite de nombres

Si l'on dispose d'un algorithme résolvant le problème 2,
alors on a aussi un algorithme résolvant le problème 1

(pas nécessairement le meilleur)

Techniques de réduction déjà beaucoup utilisées pour déterminer la (non) calculabilité de problèmes

3. Exemple

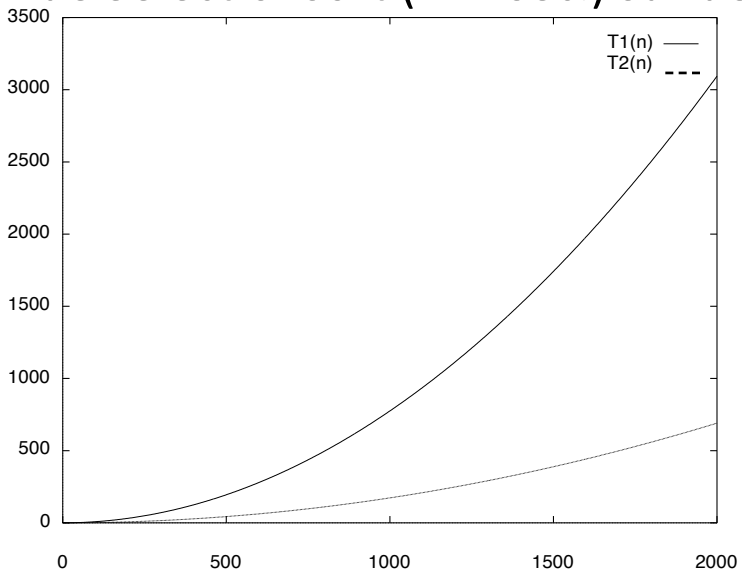
```
void sort(int A[])
// Cette méthode trie le tableau A en ordre décroissant
{
      int n = A.length;
      for (int i = 0; i < n-1; i++)
      // Les positions 0 à i-1 du tableau A ont les i premiers plus petits éléments du tableau
      //  et sont triées
      {
         // Trouver la position du plus petit élément pas encore trié (entre i et n-1)
         int min_idx = i;
         for (int j = i+1; j < n; j++)
            if (A[j] < A[min_idx])
               min_idx = j;
         // Swapper l'élément trouvé avec celui en position i
         int temp = A[min_idx];
         A[min_idx] = A[i];
         A[i] = temp;
      }
   }
```

3. Exemple

Exemple (cont.)

taille n tableau	petit ordinateur	ordinateur
125	12.5	2.8
250	49.3	11.0
500	195.8	43.4
1000	780.3	172.9
2000	3114.9	690.5

Temps d'exécution de Selectionsort (millisec.) sur deux ordinateurs différents



3. Exemple

Exemple (cont.)

$T_1(n) = 0.000772\ n^2 + 0.00305\ n + 0.001$

$T_2(n) = 0.0001724\ n^2 + 0.0004\ n + 0.1$

$$T(n) = a\ n^2 + b\ n + c$$

Peu importe l'ordinateur utilisé, le temps d'exécution sera une fonction quadratique de la taille (n) du tableau à trier

3. Exemple

Complexité

- Différentes complexités :
  - temps de calcul moyen
  - temps de calcul dans le meilleur des cas
  - temps de calcul dans le pire des cas
- Complexité moyenne est souvent difficile à déterminer
- Nécessite des considérations statistiques

Nous ne considérerons que la complexité *dans le pire des cas*

3. Exemple

Classes de complexité

Analyse d'un programme

⇒ forme générale de la courbe

(nous ne sommes pas intéressés par la valeur des coefficients)

Une ***classe de complexité*** caractérise une **famille** de courbes

Notation "grand O": permet une caractérisation concise des classes de complexité

$T(n) = a\ n^2 + b\ n + c$

$T(n)$ est $O(n^2)$

4. Notation “grand *O*”

Objectifs

Mesure indépendante des caractéristiques techniques de l'environnement technologique

- nombre d'instructions machine générées par le compilateur
- vitesse du CPU
- nature des instructions (comparaison, affectation, arithmétique, …)

Mesure dans le pire des cas

- mesure de complexité pour une donnée de taille n
- borne supérieure pour **toute** donnée de taille n

4. Notation “grand O”

Terme dominant

$T(n) = a\ n^2 + b\ n + c$

$T(n) = 0.0001724\ n^2 + 0.00040\ n + 0.100$

n	*T(n)*	*a n²*	*n² vs % du total*
125	2.8	2.7	94.7
250	11.0	10.8	98.2
500	43.4	43.1	99.3
1000	172.9	172.4	99.7
2000	690.5	689.6	99.9

Seul le *terme dominant* est utilisé dans la notation “grand O”

4. Notation “grand *O*”

Pas de constante de proportionnalité

Exécution sur différents ordinateurs donnerait différentes constantes a pour le terme $a.n^2$

⇒ ignorer différentes constantes de proportionnalité

$T(n)$ est $O(n^2)$

Propriétés de la notation “grand *O*”

- Se concentrer sur le terme dominant
- Ignorer les termes plus petits
- Ignorer les constantes de proportionnalité
- Se concentrer sur les problèmes de grande taille
- Ignorer ce qui se passe pour les problèmes de petite taille

4. Notation “grand *O*”

Définition

O(g(n)) : le nombre d’étapes exécutées par le programme (i.e. temps d’exécution) pour une donnée de taille *n* n’est pas plus élevé que *c g(n)* (où *c* est une constante), pour des données de tailles *n* suffisamment grandes

T(n)* est *O(g(n)) ssi

$$\exists\, c, n_0\ \forall\, n \geq n_0 : |T(n)| \leq c\ |g(n)|$$

Il existe un facteur multiplicatif *c* tel que pour des valeurs de *n* suffisamment grandes, *g(n)* est une borne supérieure de *T(n)*

Notations

Lorsque *T(n)* est *O(g(n))* on écrit

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

(mais on devrait utiliser $T(n) \in O(g(n))$)

4. Notation “grand O”

Preuve de complexité

$$2n^2 + n - 1 = O(n^2)$$

Montrons que

$\exists\, n_0, K > 0$

tel que

$\forall\, n \geq n_0 : 2n^2 + n - 1 \leq K n^2$

Prenons

$n_0 = 5$

$K = 4$

On a :

$2 n^2 + n - 1 \leq 4 n^2$

$2 n^2 - n + 1 \geq 0$

Toujours vrai pour $n \geq 5$

(Note: le choix de n_0 est ici sans importance)

4. Notation “grand O”

Propriétés

Grand O est une borne supérieure

$3n^2 = O(n^5)$

$n = O(n^5)$

Facteurs constants inutiles

$1000\ n^3 = O(0.0001\ n^3)$

$3n^5 = O(n^5)$

Termes d'ordre inférieur négligeables

$n^5 + n^3 = O(n^5)$

⇒ ne pas mettre de facteurs constants ni de termes inutiles

Base de logarithmes inutile

$\log_2 n = O(\log_{10} n) = O(\log n)$

4. Notation “grand *O*”

Transitivité

Si $T(n) = O(f(n))$

et $f(n) = O(g(n))$

Alors $T(n) = O(g(n))$

$$4n^2 = O(n^2) \quad \text{et} \quad n^2 = O(n^3)$$

Donc $4n^2 = O(n^3)$

Limites

Si $\lim_{n \to \infty} \frac{g(n)}{f(n)} = a$ *(avec* a $\neq \infty$*)*

Alors $g(n) = O(f(n))$

$g(n)$ ne croît pas plus vite que $f(n)$

(Attention, l'inverse n'est pas nécessairement vrai)

4. Notation “grand O”

Application

$n^2 = O(2^n)$

$n^{1000} = O(2^n)$

$\log_2 n = O(\sqrt{n})$

$(\log_{10} n)^{100} = O(n)$

$2^n = O(3^n)$

Sommation

Si

- $T_1(n) = O(f_1(n))$
- $T_2(n) = O(f_2(n))$
- $f_2(n) = O(f_1(n))$

Alors $T_1(n) + T_2(n) = O(f_1(n))$

Très utile pour déterminer le temps d'exécution de plusieurs fragments de programmes exécutés les uns après les autres

4. Notation “grand *O*”

Problèmes avec notation “grand O”

Borne supérieure

La notation *O* n'est pas très précise

Pour un programme donné, $T(n)=O(n^3)$ signifie que n^3 est une borne supérieure du temps d'exécution

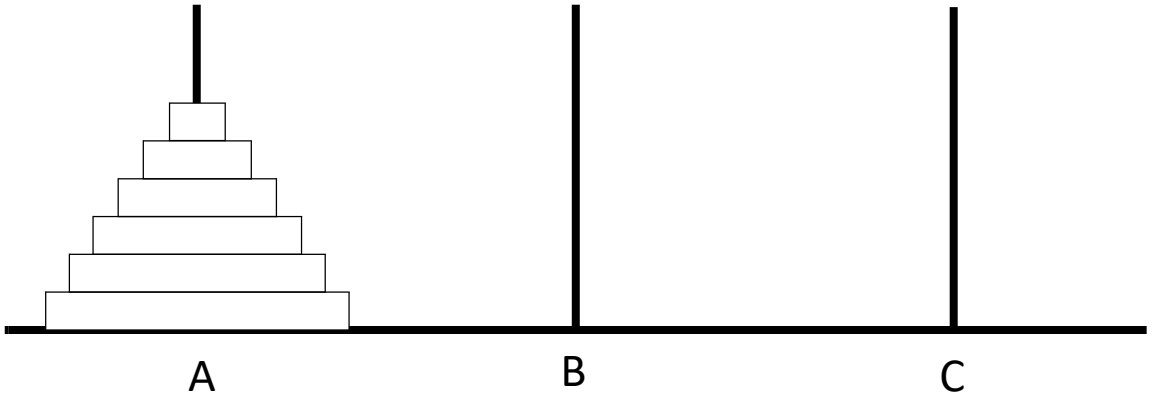
Il est possible qu'on ait également $T(n)=O(n^2)$, qui est une estimation plus favorable du temps d'exécution du même programme

Existence d’autres notations

- $\Omega\ (g(n))$: borne inférieure de complexité
  - Si f(n) = O(g(n))
    Alors g(n) = Ω(f(n))
- $\Theta\ (g(n))$: ordre (exact) de complexité
  - Si f(n) = O(g(n))
    et f(n) = Ω(g(n))
    Alors g(n) = Θ(f(n))

5. Calcul de complexité un exemple: les tours de Hanoi

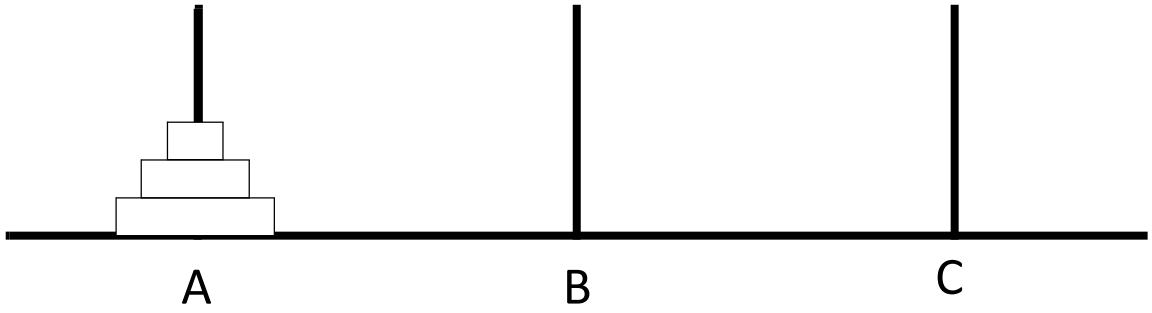
Le problème



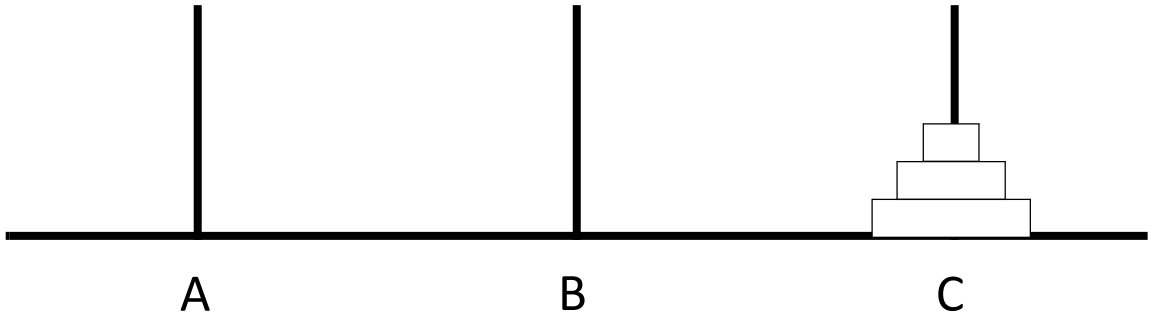
- Objectif: transférer ***tous*** les disques de A vers C
- Règles:
  - Ne bouger qu'un disque à la fois
  - ne jamais placer un disque sur un disque plus petit
- Problème: écrire un programme qui trouve les mouvements àréaliser pour atteindre l'objectif

5. Calcul de complexité un exemple: les tours de Hanoi

Exemple



MOVE A C
MOVE A B
MOVE C B
MOVE A C
MOVE B A
MOVE B C
MOVE A C



5. Calcul de complexité un exemple: les tours de Hanoi

Application

Le problème des tours de Hanoi est un exemple de problème de planification (intelligence artificielle)

Spécification

procedure hanoi(n: integer; start, aux, finish : char)

{* *PRE: n > 0*

POST: sortie standard : mouvements pour réaliser le transfert de n disques de start vers finish

*}

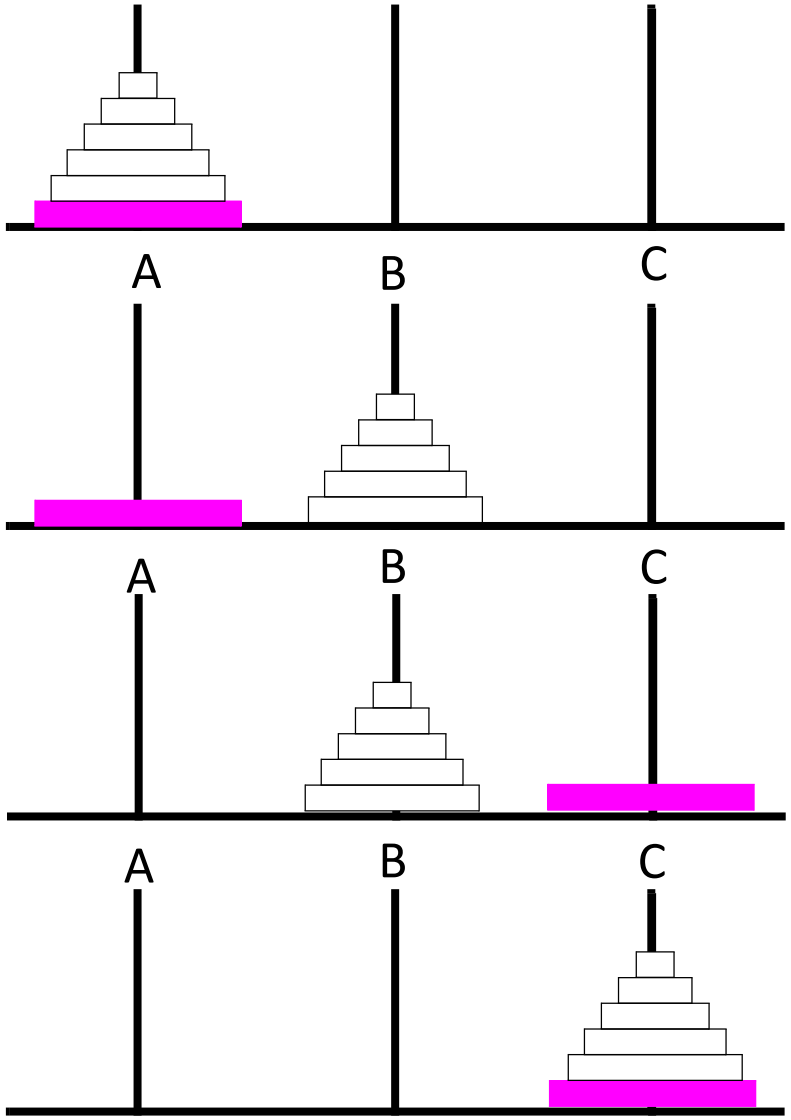
5. Calcul de complexité un exemple: les tours de Hanoi

Comment résoudre hanoi (n, 'A', 'B', 'C') ?

hanoi(n-1, 'A','C','B')

MOVE A C

hanoi(n-1,'B','A','C')



5. Calcul de complexité un exemple: les tours de Hanoi

Programme

```
procedure hanoi( n: integer; start,aux,finish : char) ;
begin
  if (n<>0) then
  begin
     hanoi (n-1,start,finish,aux);
     writeln("MOVE",start,"  ", finish);
     hanoi(n-1, aux, start, finish);
   end;
end;

...

{* appel dans programme principal *}
readln("nombre de disques ? : "; n) ;
Hanoi (n,'A','B','C');
```

5. Calcul de complexité un exemple: les tours de Hanoi

Complexité

La complexité de hanoi (n, ...) est le nombre de mouvements de la solution

M(n) : nombre de mouvements de la solution de hanoi(n)

La complexité de hanoi(n) est *O(M(n))*

Relation de récurrence

$M(0) = 0$

$M(n) = 2 * M(n-1) + 1$ (pour $n > 0$)

Comment résoudre une relation de récurrence ?

- Imaginer une solution et la prouver
- Méthode itérative
- Application d'une formule générale
- Méthodes mathématiques

5. Calcul de complexité un exemple: les tours de Hanoi

Méthode itérative

$M(n) = 2 * M(n-1) + 1$

$= 2 * (2 * M(n-2) + 1) + 1$

$= 4 * M(n-2) + 3$

$= 8 * M(n-3) + 7$

$= 2^3 * M(n-3) + (2^3 - 1)$

$= \ldots$

$= 2^k * M(n-k) + (2^k - 1)$

$= \ldots$

$= 2^n * M(0) + (2^n - 1)$

$= 2^n - 1$

$M(n) = 2^n - 1$

Complexité de hanoi(n,...) est $O(2^n)$

5. Calcul de complexité un exemple: les tours de Hanoi

Formule générale

$$C(n) = a * C(n-1) + b$$

Forme normale

$C(n) = a^n * C(0) + b * \sum (0 \leq i < n)\ a^i$

Cas particuliers :

- a=1: $C(n)$ est $O(n)$
- a=2: $C(n)$ est $O(2^n)$
- a≠1: $C(n)$ est $O(a^n)$

5. Calcul de complexité un exemple: les tours de Hanoi

Autre formule générale

$T(0) = c$

$T(n) = a * n + b + T(n-1)$

Forme normale:

$T(n) = a/2\ n^2 + (a/2 + b)*n + c$

Cas particuliers :

- **a=0:** T*(n)* est O*(n)*
- **a≠0:** T*(n)* est O*(n^2)*

Exemple:

- SelectionSort est *O(n^2)*

5. Calcul de complexité un exemple: les tours de Hanoi

Autre formule générale

$$T(n) = a * T(n/b) + c * n$$

avec $a \geq 1$, $b > 1$, $c \neq 0$

Cas particuliers :

- **b < a :** *T(n) est* $O(n^{(\ln a / \ln b)})$
- **b = a:** T*(n)* est O*(n log n)*
- **b > a:** T*(n)* est O*(n)*
- **a = 1, c = 0:** T*(n)* est O*(log n)*

Exemple:

MergeSort est *O(n log n)*

BinarySearch est *O(log n)*

6. Hiérarchie de complexités

Nom	notation
Constante	$O(1)$
Logarithmique	$O(\log n)$
Linéaire	$O(n)$
n log n	$O(n \log n)$
Quadratique	$O(n^2)$
Cubique	$O(n^3)$
Exponentielle	$O(2^n)$
Exponentielle	$O(10^n)$

Les algorithmes de complexité exponentielle sont intrinsèquement complexes

6. Hiérarchie de complexités

Complexité : temps d'exécution

- Hypothèses:
  - ordinateur 100.000 Mips
  - traitement de 1 élément = 1.000 instructions machine

	10	50	100	500	1000	10000
n	.0000001 sec.	.0000005 sec.	.000001 sec.	.000005 sec.	.00001 sec.	.0001 sec.
n^2	.000001 sec.	.000025 sec.	.0001 sec.	.0025 sec.	0.01 sec.	1 sec.
n^3	.00001 sec.	.00125 sec.	0.01 sec	1.25 sec.	10 sec.	2.8 heures
n^5	.001 sec.	3.13 sec.	1.67 min.	3.6 jours	115.7 jours	31.7 siècles
2^n	.0000102 sec.	130 jours	$4x10^{12}$ siècles	…	…	…
3^n	.00059 sec.	$2x10^5$ siècles	…	…	…	…

6. Hiérarchie de complexités

Exemples

- **Algorithmes constants**
  - Imprimer un élément d'un tableau
  - Ajouter un élément à une queue
  - Rechercher un élément dans une table de hachage
- **Algorithmes logarithmiques**
  - Recherche dichotomique dans un tableau
  - Recherche dans un arbre binaire de recherche
- **Algorithmes linéaires**
  - Recherche d'un élément dans un tableau non trié
- **Algorithmes n log n**
  - Tri par tas, quicksort

6. Hiérarchie de complexités

- **Algorithmes quadratiques**
  - Tri par échange
  - Tri à bulle (bubble sort)
  - Tri par insertion
  - Addition de 2 matrices n*n
- **Algorithmes cubiques**
  - Multiplication de 2 matrices n*n
  - Perspective d'une image 3D
- **Algorithmes exponentiels**
  - Voyageur de commerce
  - Coloration de graphes
  - Problèmes d'ordonnancement
  - Problèmes de planification

7. Problèmes intrinsèquement complexes

Certains problèmes, quoique théoriquement solubles par programmes, sont en pratique *infaisables*, car nécessitant des ressources incompatibles avec les réalités physiques.

Si il existe un algorithme de complexité polynomiale,

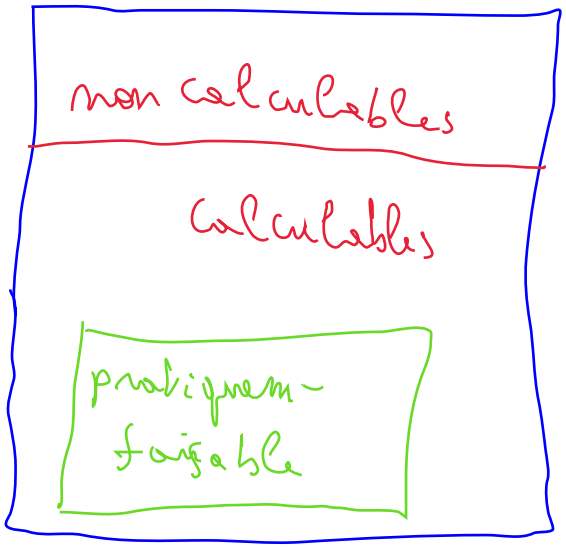
alors le problème est *pratiquement faisable*

Si il n'existe pas d'algorithme de complexité polynomiale,

alors le problème est *pratiquement infaisable*

Existence d'algorithmes de complexité exponentielle (ou pire)

Problèmes



7. Problèmes intrinsèquement complexes

Définition

Un problème est **intrinsèquement complexe**
ssi il ***n'***existe ***pas*** d'algorithme de ***complexité polynomiale***
résolvant le problème

Un problème intrinsèquement complexe est pratiquement infaisable

Beaucoup de problèmes intéressants sont intrinsèquement complexes ...

7. Problèmes intrinsèquement complexes

Propriétés

Exponentiel vs polynomial

Pour toute complexité exponentielle E, pour toute complexité polynomiale P,

$$\exists k: \forall n > k \; E(n) > P(n)$$

Complexité exponentielle devient toujours pire qu'une complexité polynomiale

Influence de l'évolution technologie

- Hypothèses:
  - ordinateur 100.000 Mips
  - traitement de 1 élément = 1.000 instructions machine
- N_i = taille du "plus grand" exemple dont la solution peut être calculée en *1 heure* de temps calcul

7. Problèmes intrinsèquement complexes

Complexité	Ordinateur d'aujourd'hui	Ordinateur 100x	Ordinateur 1000x
n	$N_1 = 3.6x10^{11}$	$100\ N_1$	$1000\ N_1$
n^2	$N_2 = 60000$	$10\ N_2$	$31.6\ N_2$
n^3	$N_3 = 7110$	$4.64\ N_3$	$10\ N_3$
n^5	$N_4 = 205$	$2.5\ N_4$	$3.98\ N_4$
2^n	$N_5 = 38$	$N_5 + 6$	$N_5 + 10$
3^n	$N_6 = 24$	$N_6 + 4$	$N_6 + 6$

Complexité exponentielle est intrinsèquement complexe

Peu importe les évolutions technologiques, un problème intrinsèquement complexe ne peut et ne pourra être résolu que pour de petits exemples

7. Problèmes intrinsèquement complexes

Influence du modèle de calcul

Si un algorithme, exprimé dans un modèle de calcul particulier, est de complexité polynomiale, alors il sera également de complexité polynomiale dans un autre modèle de calcul (complexité spatiale et temporelle)

Hypothèse de modèle de calcul réaliste (pas de non déterminisme)

La classe de problèmes intrinsèquement complexes est donc ***indépendante*** du modèle de calcul

7. Problèmes intrinsèquement complexes

Influence de la représentation des données

En pratique, différents choix de représentation ou de codage des données induisent une variation *polynomiale* du temps d'exécution et de l'espace

Hypothèses:

- représentation concise; pas d'information inutile
- représentation effective (peut être décodée par un programme)
- nombres non représentés en base 1
  taille de la représentation est logarithmique par rapport au nombre.

La classe de problèmes intrinsèquement complexes est donc ***indépendante*** de la représentation des données

7. Problèmes intrinsèquement complexes

Remarques

- Un problème est intrinsèquement complexe car parmi toutes les données de taille *n*, il existe au moins *une* pour laquelle le temps d'exécution est exponentiel, ou pire
- Pour certains problèmes intrinsèquement complexes, il existe des algorithmes polynomiaux pour *presque* toutes les données de taille *n*.
- Les cas où une donnée de taille *n* donne lieu à une complexité exponentielle peuvent parfois correspondre à des cas particuliers que l'on rencontre peu en pratique
- Il existe des algorithmes (efficaces) permettant de calculer des **approximations** de la solution exacte