Algorithmique Avancée

Auditoire :

1ère Année en Master Professionnel
en Informatique et Réseaux
ISET de Sfax

Algorithmique Avancée

1. Introduction , Complexité des algorithmes
2. Algorithmes de Tri
3. Le concept « Diviser pour régner »
4. Structures Arborescentes de Recherche
3. Les graphes

Introduction : Définitions

- Un Algorithme =
  - une suite ordonnée d'opérations ou d'instruction écrites pour la résolution d'un problème donné.

- Algorithme =
  - une suite d'actions que devra effectuer un automate pour arriver à partir d'un état initial, en un temps fini, à un résultat

Introduction : Qualités d'un bon algorithme

- Correct
  - Il se termine
  - Le résultat qu'il donne est « correct »
- Complet
  - considère tous les cas possibles et donne un résultat dans chaque cas.
- Efficace
  - rapide (en termes de temps d'exécution) ;
  - peu gourmand en ressources (espace de stockage, mémoire utilisée)

Complexité Algorithmique : Motivation

- Besoin d'outils qui permettent
  - d‘évaluer la qualité théorique des algorithmes proposés
  - De comparer différentes solutions algorithmiques pour un même problème

- But du chapitre : examiner l’efficacité d’un algorithme en termes de :
  - Temps d’ exécution : Complexité temporelle
  - Espace mémoire : Complexité spatiale

Complexité Temporelle ?

- Unités de mesure :
  - On ne mesure pas la durée en minutes, secondes, ... :
- Pourquoi?
  - cela impliquerait d'implémenter les algorithmes qu'on veut comparer ;
  - ces mesures ne seraient pas pertinentes car le même algorithme sera plus rapide sur une machine plus puissante ;
- Solution
  - utiliser des *unités de temps abstraites* proportionnelles au nombre d'opérations effectuées ;
  - Au besoin, adapter ces quantités en fonction de la machine sur laquelle l'algorithme s'exécute

Calcul de la Complexité Temporelle : Principe

- Chaque instruction basique consomme une unité de temps :
  - affectation d'une variable, comparaison, +, -,* , =, …
- Chaque itération d'une boucle rajoute le nombre d'unités de temps consommées dans le corps de cette boucle ;
- Chaque appel de fonction rajoute le nombre d'unités de temps consommées dans cette fonction ;

→ Pour avoir le nombre d'opérations effectuées par l'algorithme on additionne le tout

Calcul de la Complexité Temporelle : Exemple1

- Exemple : calcul la factorielle d'un nombre N >= 0
  - N! = N * (N-1)*(N-2)* … * 2 * 1 (avec 0!=1)

```
Fonction Factorielle ( N : entier ) : entier
Var
 i, fact : entier
Début
        fact ← 1
        i ← 2
        tantque (i<=N) faire
                        fact ← fact * i
                        i ← i+1
        finfaire
Factorielle ← fact
Fin factorielle
```

Initialisation : 1
initialisation : 1
itérations : au plus N- 1
multiplication + affectation : 2
addition + affectation : 2

Dernier test +1
Renvoi d'une valeur : 1

- Pour chaque itération, il y a un test
- Nombre total d'opérations est :
  - 1 + 1 + (N -1) * 5 + 1 + 1 = 5N - 1

Calcul de la Complexité Temporelle : Exemple2

- Exemple : calcul du pgcd de deux entiers a et b

```
Fonction PGCD (a,b : entier) : entier
Var
   d : entier
Début
    d← min(a,b)                                          4+1
    tant que (reste(a,d)<>0 ou reste (b,d) <> 0 faire    5+5+3
        d← d-1                                           2
     fin faire
      PGCD ←d                                            1
Fin PGCD
```

Min (a,b) -1 itérations

```
Fonction MIN(a,b : entier ) : entier
Var
M:entier
Debut
            M← a
            Si b< a alors                                4
                        M←b
            Fin Si
            Min ←M
Fin MIN
```

```
Fonction Reste(i,j : entier) : entier
Var
            d : entier
Début                                                    5
            d← i/j
            Reste ←i – d * j
Fin Reste
```

- Calculer le nombre d'opérations de PGCD

5+ (min(a,b)-1)* (5+5+3+2) + 13+1 = 15 min(a,b) + 4

Calcul de la Complexité Temporelle : Remarques

- Le calcul n'est pas toujours exact !!!
  - Le nombre d'itération peut ne pas être connu d'avance

```
Lire(x)
i←1
S←0
Tant que (i<=X )faire
        s ←s+ 1
Fin faire
```

  - Lors du branchement conditionnel, le nombre de comparaisons à effectuer n'est pas toujours le même

```
if (i<= N && T[i] > T[i-1])
```

Calcul de la Complexité Temporelle : Définitions (1)

- Qu'en est il pour la recherche séquentielle dans un tableau ?

```
Fonction recherche ( T : Tab, N : entier , X : entier) : logique
Var
            i : entier
            trouve : logique
Début
I ← 1
Tant que (i<= N et T[i] <> X ) faire
            i ← i +1
Fin faire
Si i> N alors
        trouve ← faux
      sinon trouve  ← vrai
Fin si
Recherche ← trouve
Fin Recherche
```

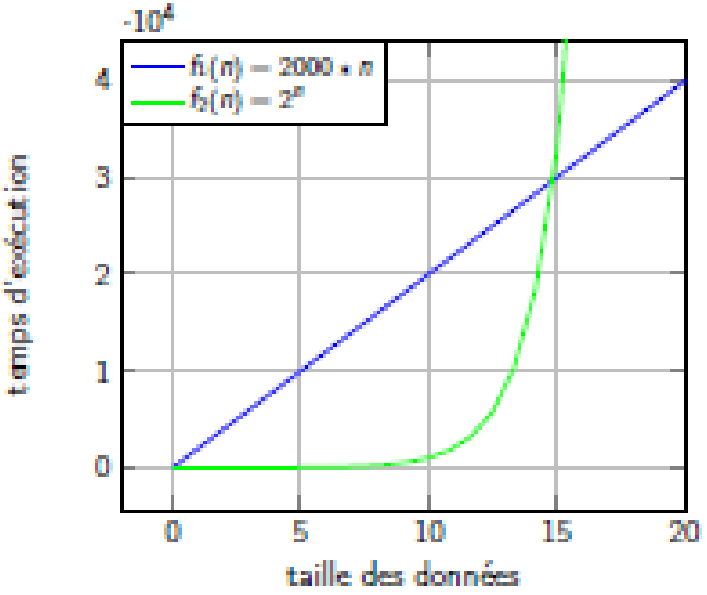
Le nombre d'itérations dépend de
- X
- T (en termes de valeurs)

Calcul de la Complexité Temporelle : Définitions (2)

- On définit 3 types de complexité :
  - Complexité au meilleur des cas : C'est le plus petit nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée
  - Complexité au pire des cas : C'est le plus grand nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée
  - Complexité en moyenne : C'est la moyenne des complexités de l'algorithme sur des jeux de données

Comportement Asymptotique (1)

- Soit
  - Un problème à résoudre de taille N, et
  - Deux algorithmes *A1* et *A2* résolvant ce problème ayant comme nombre d'opérations respectivement $f_1(N)$ et $f_2(N)$



- Que choisir?
  - A2 semble correspondre à l'algorithme le plus efficace
  - Mais seulement pour de très petites valeurs

Comportement Asymptotique (2)

- La complexité d'un algorithme est une mesure de sa performance asymptotique dans le pire cas
- Que signifie asymptotique ?
  - on s'intéresse à des données très grandes ;
  - pourquoi ?
    - les petites valeurs ne sont pas assez informatives ;
- Que signifie dans le pire cas" ?
  - on s'intéresse à la performance de l'algorithme dans les situations où le problème prend le plus de temps à résoudre ;
  - pourquoi ?
    - on veut être sûr que l'algorithme ne prendra jamais plus de temps que ce qu'on a estimé ;

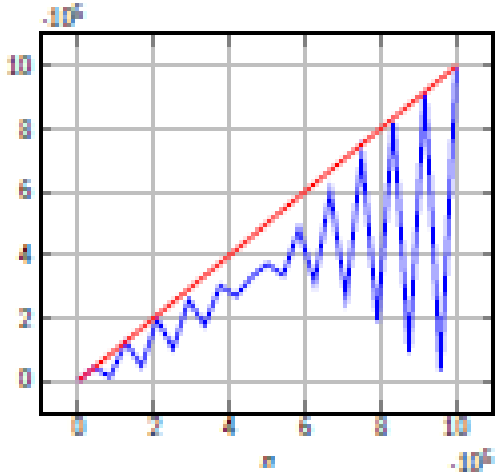
La notation *O(.)*

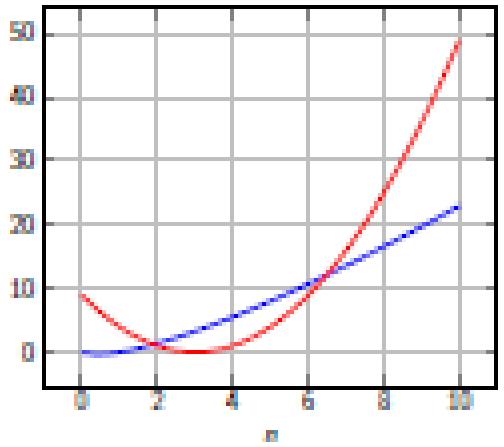
- Les calculs effectués
  - Peuvent être longs et pénibles
  - Leurs valeurs précises peuvent être inutiles
- On va faire recours à une approximation de ce calcul représentée par *O(.)*
- si :

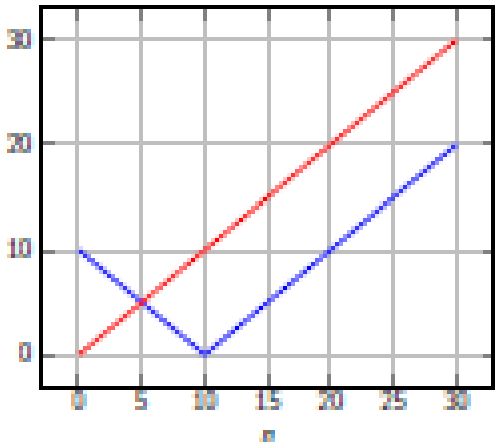
$$\exists\, n_0 \in \mathbb{N}, \exists\, c \in \mathbb{R}, \forall\, n \geq n_0 : \quad |f(n)| \leq c|g(n)|$$

Autrement dit : f (n) est en O(g(n)) s'il existe un seuil à partir duquel la croissance de la fonction f (.) est toujours dominée par la fonction g(.), à une constante multiplicative fixée près

La notation *O(.) :* Exemples







Quelques cas où f (n) est O(g(n))

La notation *O(.) :* Exemples d'utilisation

- Prouvons que la fonction $f_1(n) = 5n + 37$ est en O(n) :
  - but : trouver une constante $c \in R$ et un seuil $n_0 \in N$ à partir duquel $|f_1(n)| \leq c|n|$
  - on en déduit donc que c = 6 fonctionne à partir du seuil n0 = 37

$$|5 * 37 + 37| \leq 6 * |37| ;$$
$$|5 * 38 + 37| \leq 6 * |38| ;$$
$$\vdots$$

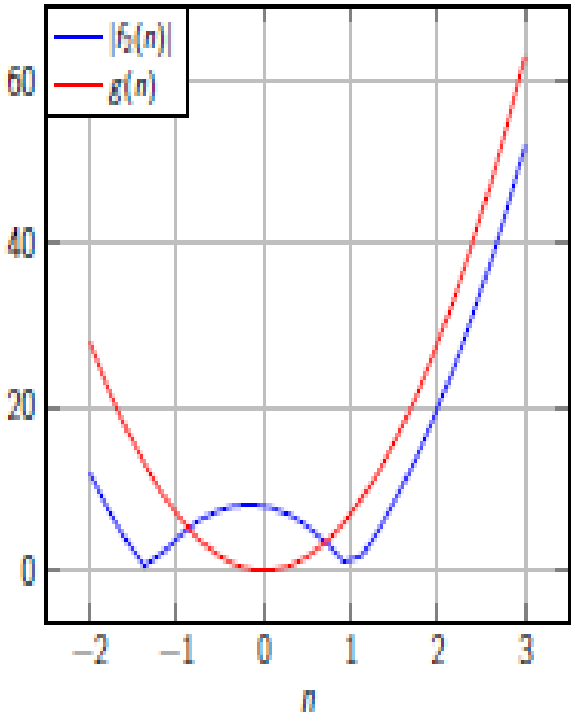
  - on remarque que $|5n + 37| \leq |6n|$ si $n \geq 37$
- Remarque
  - l'optimisation n'est pas demandée (le plus petit c où n_0 qui fonctionne). Il faut juste fournir des valeurs qui fonctionnent (c=10 et $n_0 = 8$ est aussi acceptable)

La notation *O(.) :* Exemples d'utilisation

- Prouvons que la fonction $f_2(n) = 6n^2 + 2n - 8$ est en $O(n^2)$
  - cherchons d'abord la constante c ;
    - c = 6 ne peut pas marcher,
    - essayons donc c = 7 ;
  - on doit alors trouver un seuil $n_0 \in N$ à partir duquel

$$|f_2(n)| \leq 7\,|n^2|$$

  - un simple calcul nous donne $n_1 = -(4/3)$ et $n_2 = 1$ comme racines de l'équation $6n^2 + 2n - 8 = 0$;
  - en conclusion, c = 7 et n0 = 1 nous donnent le résultat voulu



La notation *O(.) :* Règles de calcul

- Les processeurs actuels effectuent plusieurs millions d'opérations à la seconde ;
  - qu'une affectation requière 2 ou 4 unités de temps ne change donc pas grand-chose ;
  - un nombre constant d'instructions est donc aussi négligeable par rapport à la croissance de la taille des données ;
  - pour de grandes valeurs de n, le terme de plus haut degré l'emportera ;

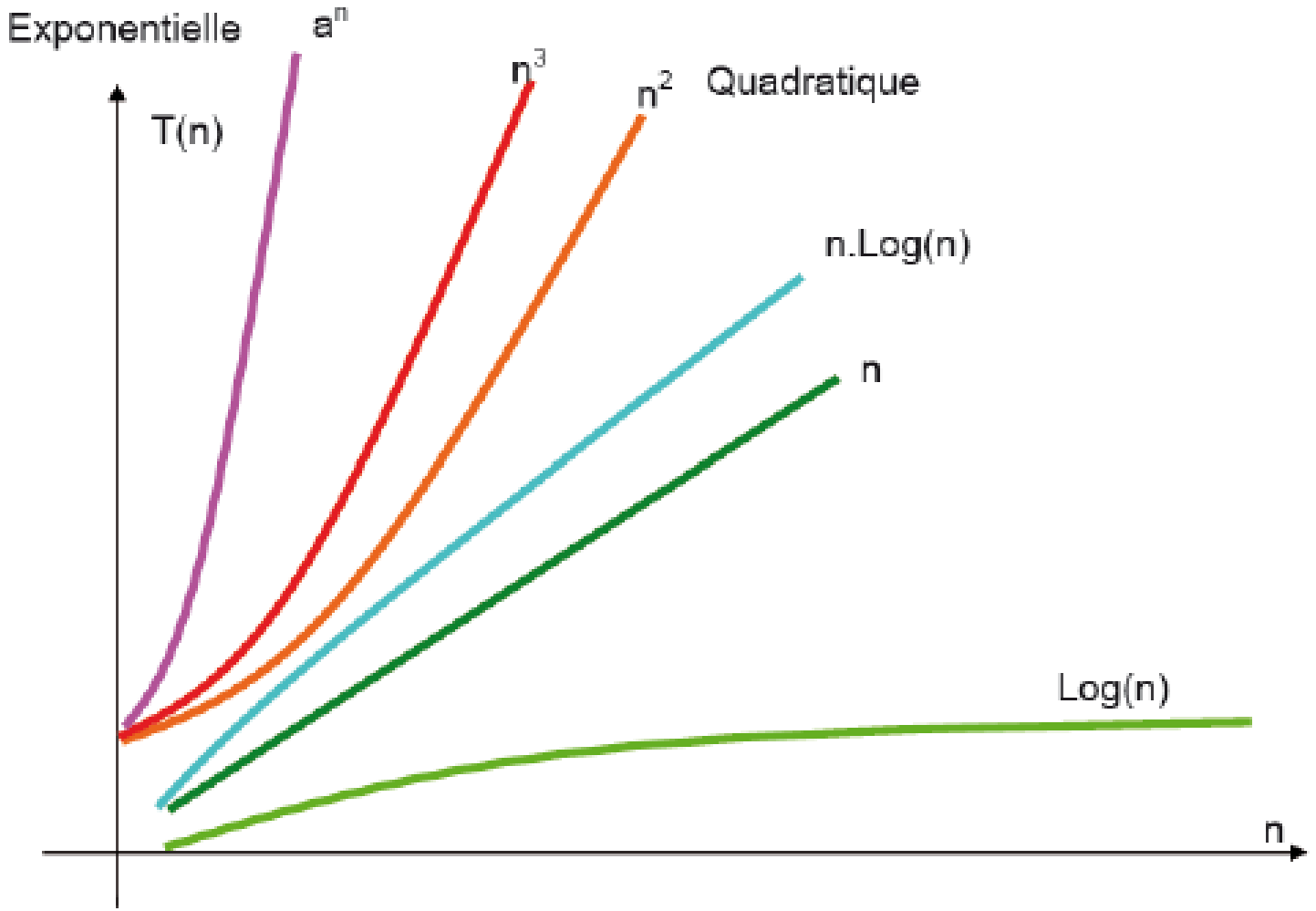
→ On préfère donc avoir une idée du temps d'exécution de l'algorithme plutôt qu'une expression plus précise mais inutilement compliquée

La notation *O(.)* : Règles de calcul & Hiérarchie

- Conséquences
  - 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 ;

- Exemple (simplifications)
  - Soit un algorithme effectuant $g(n) = 6n^2 + 2n - 8$ operations ;
    - on remplace les constantes multiplicatives par 1 : $1n^2 + 1n - 8$
    - on annule les constantes additives : $1n^2 + 1n + 0$
    - on garde le terme de plus haut degré : $n^2 + 0$
- et on a donc $g(n) = O(n^2)$.

Classes de complexité



Hiérarchie

- Pour faire un choix entre plusieurs algorithmes, il faut être capable de situer leurs complexités

- On fait une première distinction entre les deux classes suivantes :

  - les algorithmes dits polynomiaux, dont la complexité est en $O(n^k)$ pour un certain k ;

  - les algorithmes dits exponentiels, dont la complexité est en $O(a^n)$ pour une certaine valeur de a

Calcul de la complexité

- Cas d'un traitement Conditionnel

```
Si (condition) Alors
    | Traitement1
Sinon
    | Traitement2
Fin Si
```

O(condition) + max(O(traitement 1),O(traitement2)

- *Exemple*

```
Si (A>10) Alors
    x← b*2 – 1
    sinon
     x← (b+2) – A  * 5
      A←A*2
Fin Si
```

```
Si (A>10 et b<A+b) Alors
    x← b*2 – 1
    sinon
     x←A*2
Fin Si
```

Calcul de la complexité (suite)

- Cas d'un traitement itératif : Boucle TantQue

```
Tant que (condition) faire
    | Traitement
Fin faire
```

Nombre Répétition * (O(Condition) + (O(Traitement)) + O(Condition)

- *Exemple*

```
i←1
Tantque(i<10) Faire
      lire(A)
       S←S+A
       i← i+1
Fin faire
```

Calcul de la complexité (suite)

- Cas d'un traitement itératif : Boucle Pour

```
Pour i de indDeb à indFin faire
    | Traitement
Fin faire
```

$$\sum_{IndDeb}^{IndFin} O(Traitement)$$

- *Exemple*

```
Pour i de 1 à 10 faire
Faire
      lire(A)
       S←S+A
Fin faire
```

Exemples de Calcul de complexité

- Tri à Bulles

```
Procédure TriBulles (Entrée N : entier, E/S tab : TabEntier)
var
    i, k :entier ;
    tmp : entier ;
Début
    Pour i de N à 2 faire
        Pour k de 1 à i-1 faire
            Si (tab[k] > tab[k+1]) alors
                tmp ← tab[k];
                tab[k] ← tab[k+1];
                tab[k+1] ← tmp;
            Fin si
        Fin faire
    Fin  faire
Fin TriBulles
```

→ $O(n^2)$

Exemples de Calcul de complexité

- Tri par insertion

```
Procédure TriInsertion (Entrée N : entier, E/S tab : TabEntier)
var
    i, k :entier ;
    tmp : entier ;
Début
    Pour i de 2 à N faire
      tmp ← tab[i];
      k ← i;
      Tant que k > 1 ET tab[k - 1] > tmp faire
            tab[k] ← tab[k - 1];
            k ← k - 1;
      Fin Tant que
      tab[k] ← tmp;
    Fin faire

Fin TriInsertion
```

Exemples de Calcul de complexité

(suite)

- Tri par insertion : calcul de la complexité
  - La taille du tableau à trier est N
  - On a deux boucles imbriquées :
    - La première indique l'élément suivant à insérer dans la partie triée du tableau
    - Elle se répète n-1 fois puisque le premier élément n'est pas traité
    - Pour chaque élément de la première boucle, on fait un parcours dans la partie triée pour déterminer son emplacement (nombre de parcours dépend de la valeur courante de *tmp*)

Exemples de Calcul de complexité (suite)

Tri par insertion : calcul de la complexité

- Au meilleur des cas:
  - Le cas le plus favorable pour cet algorithme est quand le tableau est déjà trié → $O(n)$
- Au pire des cas :
  - Le cas le plus défavorable est quand le tableau est inversement trié :
    - I = 2 → 1 itération
    - I = 3 → 2 itérations
    -
    - I = N → N-1 itérations
  - Soit $1 + 2+3+ \ldots+(n-1) = (n(n+1)/2) - n = n(n-1)/2$ *Somme des éléments d'une suite arithmétique : (premier terme + dernier terme)*nombre de termes /2*

  → sa complexité est de $O(n^2)$
- En moyenne des cas:
  - La moitié des éléments sont triés, et sur l'autre moitié ils sont inversement triés

  → $O(n^2)$

Exemples de Calcul de complexité

- Recherche dichotomique

```
Fonction RechDicho(Tab :Tableau, borneinf :entier, bornesup :entier,
elemcherche :entier) : entier
        Trouve = false ;
        Tant que ((non trouve) ET (borneinf<=bornesup)) faire
          mil = (borneinf+bornesup) DIV 2
          Si (Tab[mil]=elemcherche) Alors
                    trouve=true
          Sinon
                    Si (elemcherche < Tab[mil]) Alors  bornesup = mil-1
                    Sinon     borneinf = mil+1
                    Fin Si
          Fin Si
        Fin faire
        Si (trouve) Alors Retourner (mil)
        Sinon Retourner (-1)
       Fin Si
. Fin RechDicho
```

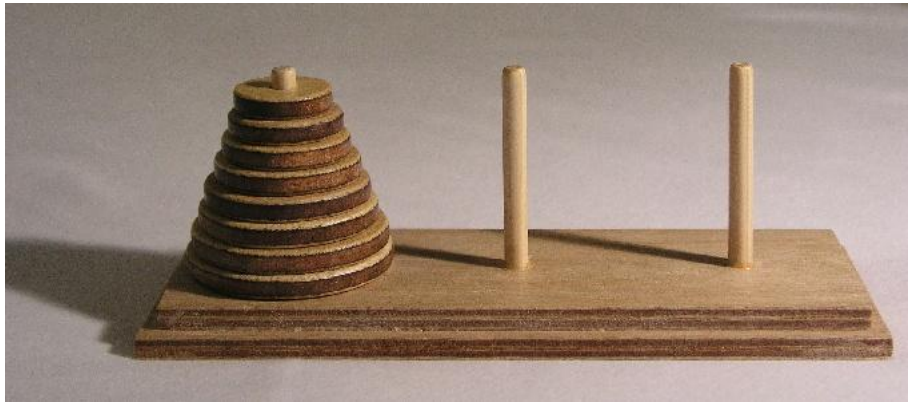
Exemples de Calcul de complexité

(suite)

- Recherche dichotomique : calcul de la complexité
  - Supposons que le tableau à trier est de taille n (une puissance de 2 ($n = 2^q$)
  - Le pire des cas pour la recherche d'un élément est de continuer les divisions jusqu'à obtenir un tableau de taille 1
  - Q est les nombre d'itérations nécessaires pour arriver à un tableau de taille 1
    - Itération 1 : $n/2 = n/2^1$
    - Itération 2 : $n/4 = n/2^2$
    - Itération 3 : $n/8 = n/2^3$
    -
    -
    - Itération q : $n/2^q$
  - Dernière itération → taille du tableau = 1
    - $n/2^q = 1$
    - $2^q = n$
    - $q = \log_2(n)$ → $O(\log_2(n))$

Exemples de Calcul de complexité (suite)

- Tour de Hanoi



Principe

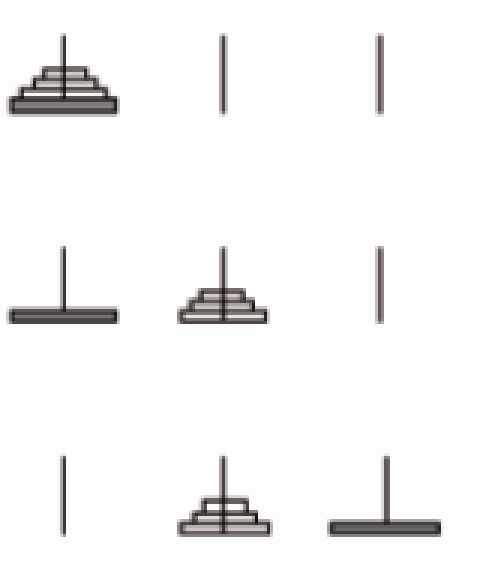
- On dispose d'une plaquette de bois où sont plantées 3 tiges numérotées 1,2 et 3
- Sur ces tiges sont empilées des disques de diamètres différents
- Règles de jeu :
  - On ne peut déplacer qu'un disque à la fois
  - Il est interdit de poser un disque sur un disque plus petit
  - A u début les disques sont sur la tige 1 (celle de gauche)
  - A la fin, les disques doivent être sur la tige 3 (celle de droite)

Exemples de Calcul de complexité

(suite)

Tour de Hanoi : Démarche

- On suppose que l'on sait résoudre le problème pour (n-1) disques,.
- Pour déplacer n disques de la tige 1 vers la tige 3 :
  - on peut déplacer les (n-1) disques les plus petits (ceux d'en haut) vers la tige 2
  - On déplace le plus grand disque de la tige 1 vers tige 3
  - On déplace les (n-1) plus petits disques de la tige 2 vers la tige 3



Exemples de Calcul de complexité
(suite)

- Tour de Hanoi : Algorithme

```
Procédure Hanoi (n, départ, intermédiaire, destination)
    Si n > 0 Alors
        Hanoi (n-1, départ, destination, intermédiaire)
        déplacer un disque de départ vers destination
        Hanoi (n-1, intermédiaire, départ, destination)
    Fin Si
Fin
```

Exemples de Calcul de complexité

(suite)

- Tour de Hanoi : Calcul de la complexité
  - On compte le nombre H(n) de déplacements pour passer n disques d'une tige de départ vers une tige destination
  - $H(n) = H(n-1) + 1 + H(n-1) = 2\,H(n-1) + 1$
  - De même : $H(n-1) = 2\,H(n-2) + 1$

$\rightarrow O(2^n)$

$$
\begin{aligned}
H(n) &= 2.H(n-1) + 1 \\
2.H(n-1) &= 2^2.H(n-2) + 2 \\
2^2.H(n-2) &= 2^3.H(n-3) + 2^2 \\
&\dots \qquad\qquad \dots \\
2^{n-2}.H(2) &= 2^{n-1}.H(1) + 2^{n-2}
\end{aligned}
$$

$$H(n) = 2^{n-1}.H(1) + 1 + 2 + \dots + 2^{n-2} = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1$$

http://championmath.free.fr/tourhanoi.htm