Les bases de l’analyse algorithmique

Analyse d’algorithmes

- Un algorithme est une suite d’instructions qui
- sert à résoudre un problème donné dans un temps fini.

- Analyser un algorithme = évaluer son efficacité

Efficacité ?

- Temps d'exécution ....
- Espace mémoire occupé ...
- Qualité du résultat ....
- Simplicité ....

Temps d’exécution d’un algorithme

- Le temps d'exécution d’un algorithme dépend de la taille des données d’entrée

- Ex:Recherche d'un éléments dans un tableau

Temps d’exécution d’un algorithme

- Il dépend aussi de la nature des données à traiter (des entrées différentes peuvent avoir des temps d’exécution différents)
- Ex:
- f(x){
- si x est impair

return x

- si x est pair

calcul la somme S des premiers x entiers

return S }

Mesurer le temps d’execution

- Approche 1: Etude Expérimental

- Limitations:
- -Il est nécessaire de implémenter
- - Lors des tests, l’ensemble des données d’entrée est réduit et ne couvre pas la totalité des cas possibles
- -Afin de comparer deux algorithmes, les mêmes
- environnements matériel et logiciel devraient être utilisés.

Mesurer le temps d’execution

Approche 2: Analyse Théorique

- Nous avons besoin d’une méthodologie générale pour analyser le temps d’exécution d’algorithmes qui:
- – Utilise une description de haut niveau de
- l’algorithme (indépendant de l'implémentation)
- –Caractérise le temps d'exécution comme une fonction de la taille des données d’entree
- –Est indépendant des environnements matériels et logiciels

Complexité temporelle

- Réaliser un calcul de complexité en temps revient à décompter le nombre d’opérations élémentaires effectuées par l’algorithme.
- Toutes les opérations élémentaires sont à égalité de coût. En pratique ce n'est pas tout à fait exact mais cette approximation est cependant raisonnable.
- On pourra donc estimer que le temps d'exécution de l'algorithme est proportionnel au nombre d’opérations élémentaires.

Complexité spatiale

La complexité en espace est quand à elle la taille de la mémoire nécessaire pour stocker les différentes structures de données utilisées lors de l'exécution de l'algorithme.

Example

Moyenne de 2 flottants:

```
MOY(a,b)
  s := a + b
  m := s/2
  return (m)
```

- Coût de l'algorithme:
  - Coût temps : 2 opérations
  - Coût en espace mémoire :2 variables
  - Coût constant, indépendant des données

calculs de complexité

- Opérations primitives:
- opérations de bas niveau qui sont indépendantes du angage de programmation, par exemple:
  - Appel et retour d'une méthode
  - Effectuer une opération arithmétique
  - Comparer deux nombres, etc.
  - Affectation d'une variable...

calculs de complexité

- En observant le pseudo-code d’un algorithme on peut compter le nombre d’opérations primitives exécutées par cet algorithme et par la suite analyser son temps d’exécution et son efficacit

calculs de complexité

- conversion(n):
- h = n / 3600
- m = (n - 3600*h) /60
- s = n % 60
- return h,m,s
- **T(n)=8**

puissanceMoinsUn(n)

if n%2==0

res = 1

else

res = -1

return res

- **T(n)=3**

calculs de complexité

```
sommeEntiers(n):
somme = 0
i=0
while i <= n
    somme += I
    i=i+1
return somme
```

.T(n)=5n+2

- sommeEntiersBis(n):
- return n*(n+1)/2

.T(n)=3

Analyse d'algorithmes - Exemple1

- Trouver l'élément maximal d'un tableau d'entiers
- Algorithme TabMax(A, n){
- currentMax ← A[0]
- for i ← 1 to n -1 do
- if currentMax < A[i] then
  currentMax ← A[i]
  }
- return currentMax
- }

Analyse d’algorithmes - Exemple

- Meilleur cas:
- 1 affection+(n-1) comparaisons
- T(n)=n
- Pire cas:
- n affections+(n-1) comparaisons
- T(n)= 2n-1

Notation asymptotique

- Dans cette sous-partie de généralités, on supposera que toutes les fonctions considérées sont définies sur n et à valeurs dans R+.

- Les définitions suivantes permettent de comparer le comportement à l'infini de deux fonctions définies sur n.

- Plus précisément, il s'agit de critères pour affirmer qu'une fonction en **domine** une autre, ou au contraire est du **même ordre de grandeur**, voir même **équivalente**.

Big-Oh

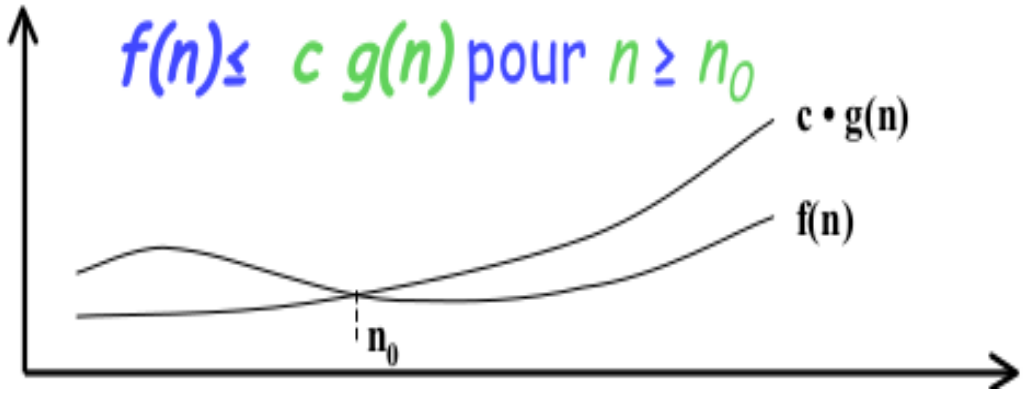
- **Borne supérieure asymptotique:**

- On dit qu'une fonction f est un big O d'une fonction g si et seulement si

$$\exists c > 0, \exists n_0 > 0 \text{ tel que } \forall n > n_0, f(n) < c \times g(n)$$

- Nous disons que:

$$f(n) \text{ est } O(g(n))$$
$$f(n) = O(g(n))$$
$$f(n) \in O(g(n))$$



Example 1 : Big- O

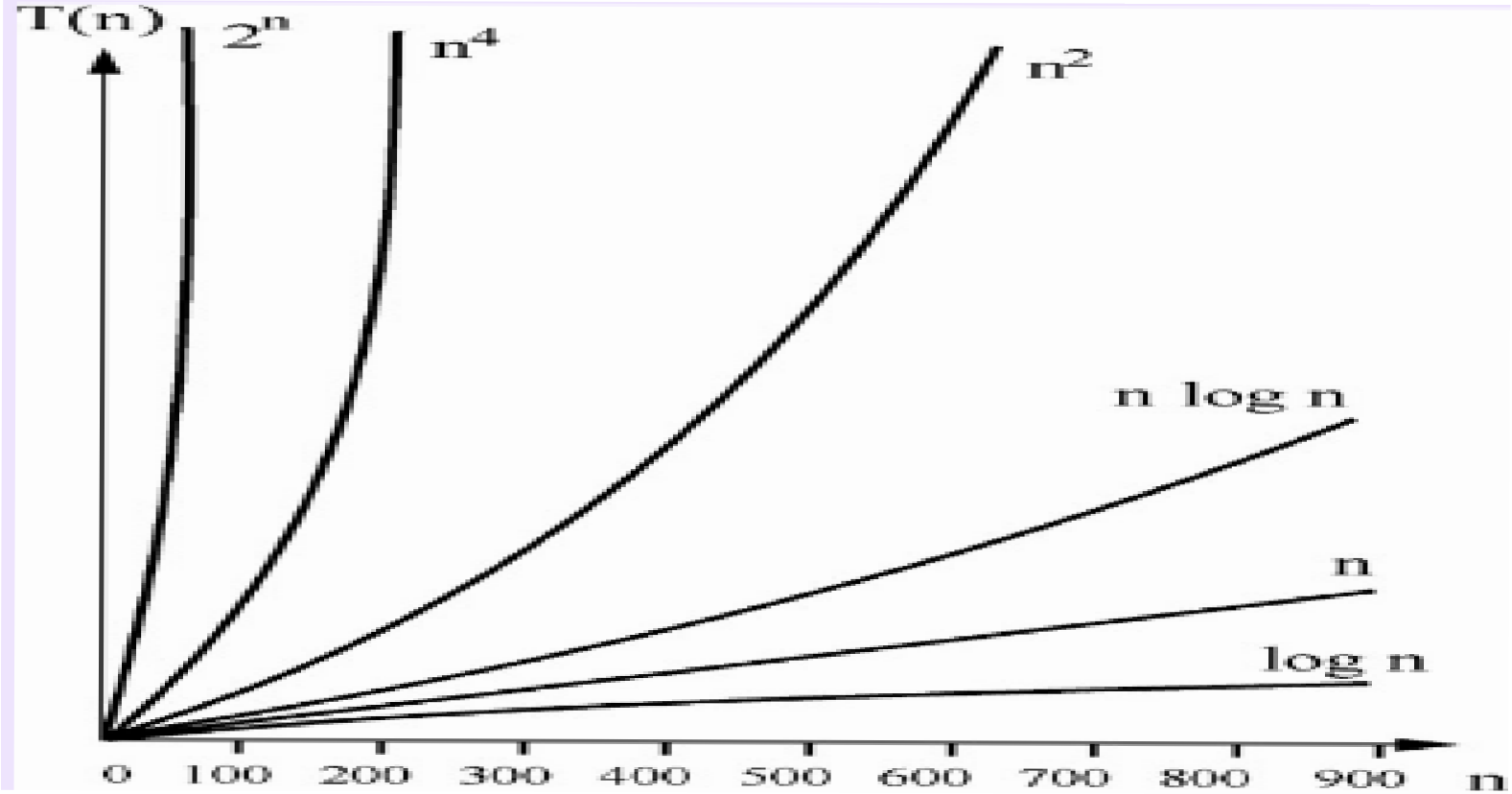
- Si T(n)=3 alors T(n)=O(1). Pour le prouver,prendre par exemple c=5 et n_0=0.
- Si T(n)=3n+2 alors T(n)=O(n). Pour le prouver, prendre par exemple c=4 et n_0=2.
- Si T(n)=2n+3 alors T(n)=O(n^2) . Pour le prouver, prendre par exemple c=3 et n_0=1.

Big-O

n =	2	16	256	1024
log log n	0	2	3	3.32
log n	1	4	8	10
n	2	16	256	1024
n log n	2	64	448	10 200
n^2	4	256	65 500	$1.05 * 10^6$
n^3	8	4 100	16 800 800	$1.07 * 10^9$
2^n	4	35 500	$11.7 * 10^6$	$1.80 * 10^{308}$

Big-O

$O(1) < O(\log n) < O(n) < O(n \log n) < O\left(n^2\right) < O\left(n^3\right) < O\left(2^n\right) \ldots$ memorisez !!



Types de complexité :

- Constant : $O(1)$
- Logarithmique : $O(\log n)$
- Linéaire : $O(n)$
- Quadratique : $O(n^2)$
- Cubique : $O(n^3)$
- Polynomial : $O(n^k), k \geq 1$
- Exponentiel : $O(a^n), n > 1$
- Factoriel: $O(n!)$

Propriété- Big O

1-Si f(n) est O(g(n)) , alors pour n'importe quelle

constante c >0 c.f(n) est aussi O(g(n))

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

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

Ex: $2n^3 + 3n^2 = O (max(2n^3 , 3n^2))$

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

3-Si $f_1(n)= O(g_1(n))$ et $f_2(n)=O(g_2(n))$ alors
$f_1(n)*f_2(n)=O(g_1(n)*g_{2(}n))$

Ex:

f est $O(n^2)$ g est O(log n) alors f.g est $O(n^2 \log n)$

Big-O

- Faire l'approximation la plus proche possible; utiliser la plus petite classe possible:
- Ex.: Il est correct de dire que 5n-3 est $O(n^2)$ mais la meilleure formulation est de dire 5n-3 est O(n)
- Utiliser l'expression la plus simple de la classe :
- Ex: Dire 10n+15 est O(n) au lieu de 10n+15 est O(10n)

Big-O

- Laisser tomber les termes d'ordre inférieur ainsi que les coefficients

- Ex.1: $7n-3$ est $O(n)$
- Ex.2: $6n^2 \log(n) + 3n^2 + 5n$ est $O(n^2 \log n)$
- Ex.3: $n^5 + 1000n^4 + 20n^3 - 8$ est $O(n^5)$

BigO : Règle 1-Unité-

- Clairement définir l'unité utilisée et les
- éléments pris en compte
- Temps d'exécution: O(1) pour :
- • une évaluation d'expression simple
  - h = j+2
  - m =T[i]*2
- • une instruction lire() ou affiche()

Big O : Règle 2:Séquence

- La complexité d'une séquence de 2 instructions est le maximum des complexités de chacune de ces instructions

Traitement	instruction 1	instruction 2	séquence
Temps max	$T_1(n)$	$T_2(n)$	$T_1(n)+T_2(n)$
$O(\ )$	$O(f_1(n))$	$O(f_2(n))$	$O(f_1(n)+f_2(n))$ $= O(max\ (f_1(n),f_2(n)))$
Exemple	$T_1(n) \in O(n^2)$	$T_2(n) \in O(n)$	$O(n^2+n) = O(n^2)$

Big O : Règle 3: if

- La complexité d'un

- **if (condition) instruction1 else instruction2** est le **maximum** des complexités de condition,instruction1 et instruction 2

Traitement	condition	instruction 1	instruction 2	if
Temps max	$T_0(n)$	$T_1(n)$	$T_2(n)$	$max(T_0(n)+T_1(n),\ T_0(n)+T_2(n))$
$O(\ )$	$O(f(n))$	$O(f_1(n))$	$O(f_2(n))$	$= O(max\ (f(n),f_1(n),f_2(n)))$
Exemple	$T_0(n) \in O(n)$	$T_1(n) \in O(n^2)$	$T_2(n) \in O(n)$	$O(n^2)$

Big O : Règle 4: while

- **La complexité d'un**
- **while (condition) instruction**
- est la complexité du nombre d'itérations * le
- maximum des complexités de condition et
- d'instruction
- **do ... while similaire au while**

Traitement	Nb. d'itérations	condition	instruction	while
Temps max	$T(n)$	$T_0(n)$	$T_1(n)$	$max(T_0(n)+T_1(n),\ T_1(n)+T_2(n))$
$O(\ )$	$O(g(n))$	$O(f(n))$	$O(f_1(n))$	$= O(g(n).max(f(n),f_1(n)))$
Exemple	$T(n) \in O(n)$	$T_0(n) \in O(1)$	$T_1(n) \in O(n^2)$	$O(n^3)$

Big O : Règle 5: for

- for(init; condition; changement) instruction
- traduit en:
- init;
- while(condition)
- {
- instruction;
- changement
- }

Big O : Règle 6: fonction

- L’appel à une fonction de complexité O(n) est en O(n’) où n’ est la taille des paramètres effectifs
- Exemple:
- x = minimum(V)
- Ayant
- – minimum en O(n) où n est la taille du vecteur et
- – V de taille m,
- => l’assignation est en O(m)

Exercice 1:Calcul de la complexité

```
int min = tab[0];
int i=1 ;
while (i<n)
{
if(V[i] < min)
min = tab[i];
++i;
}
Return min;
```

Exercice 2:Calcul de la complexité

```
k=0;
for (int i=0; i<n;++i)
for(int j=0; j<i; ++j)
{
k+=i*j;
}
```

Exercice 3:Calcul de la complexité

```
k=0;
for (int i=1;  i<=n; i*=2)
for(int k=1;  k<=n;  ++k)
{
 j*=2;
}
```

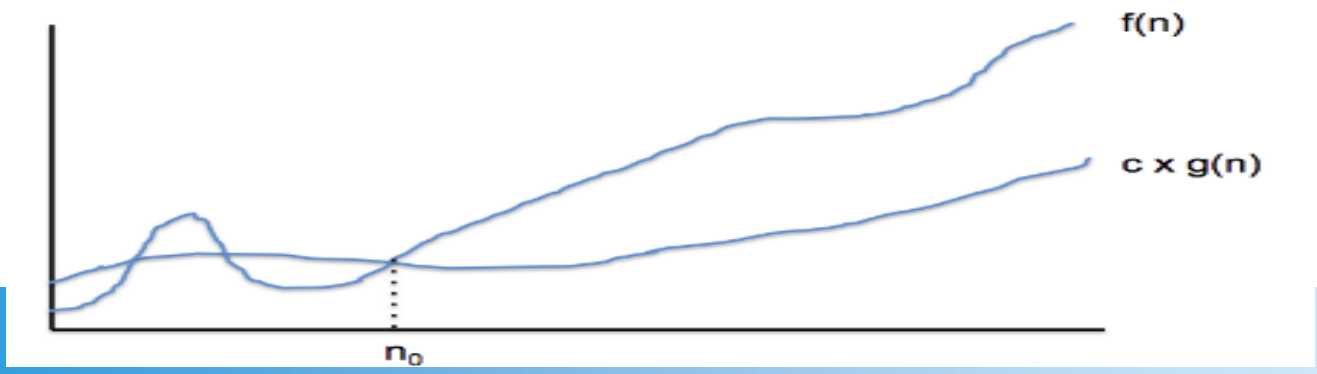
Exercie 4: Ex2+Ex3

- k=0;
- for (int i=0; i<n;++i)
- for(int j=0; j<i; ++j)
- {k+=i*j;}

- j =0;
- for (int i=1; i<=n;i*=2)
- for(int k=0; k<=n; ++k)
- {j*=2;}

Big Oméga

- **Borne inférieure asymptotique:**
- On dit qu'une fonction f est un grand Oméga d'une fonction g si et seulement si
- $c>0, \exists\ n_0>0$ tel que $\forall n>n_0, c \times g(n)<f(n)$
- On note $f(n)=\Omega(g(n))$.



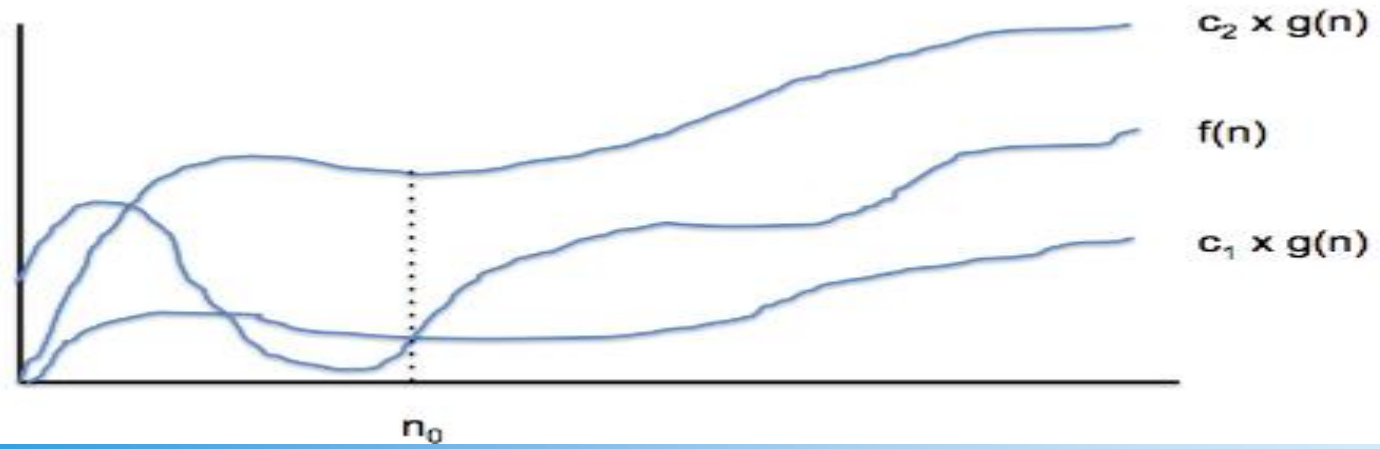
Big Oméga

- Si $T(n)=4$ alors $T(n)=\Omega(1)$.
- Si $T(n)=4n+1$ alors $T(n)=\Omega(n)$.
- Si $T(n)=4n^2+1$ alors $T(n)=\Omega(n)$.

Big Théta

- Borne asymptotique: On dit qu'une fonction
- f est un grand Théta d'une fonction g si et seulement si ∃ $c_1>0$,∃ $c_2>0$, ∃ $n_0>0$ tel que ∀ $n>n_0$, $c_1 \times g(n) < f(n)<c_2 \times g(n)$
- On note alors

$$f(n) = \Theta(g(n))$$



Big Théta

- Si $T(n)=4$ alors $T(n)=\Theta(1)$.
- Si $T(n)=4n+2$ alors $T(n)=\Theta(n)$.
- Si $T(n)=4n^2+1$ alors $T(n)=\Theta(n^2)$.

Règle de la limite

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}_+^\star \text{ ssi } [f(n) \in \Theta(g(n))]$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \text{ ssi } [f(n) \in \mathcal{O}(g(n)) \text{ mais } f(n) \notin \Theta(g(n))]$$

$$\lim_{n\to\infty} \frac{f(n)}{a(n)} = \infty \text{ ssi} [f(n) \in \Omega(g(n)) \text{ mais } f(n) \notin \Theta(g(n))]$$

- Règle de l'Hôpital

$$\lim_{x\to a} \frac{g(x)}{h(x)} = \lim_{x\to a} \frac{g'(x)}{h'(x)}$$

Intuition pour les notations asymptotiques

- Big-Oh
- – f(n) est O(g(n)) si f(n) est plus petite ou égal
- à g(n) quand n est grand
- big-Omega
- – f(n) est Ω(g(n)) si f(n) est plus grand ou égal
- à g(n) quand n est grand
- big-Theta
- – f(n) est Θ(g(n)) si f(n) est a peu près égal à
- g(n) quand n est grand

$$\Theta(f) = \mathcal{O}(f) \cap \Omega(f)$$

Complexité d’un algorithme récursif

- Soit l’algorithme :
- factorielle (n: Naturel) : Naturel
- début
- si n = 0 alors
- retourner 1
- sinon
- retourner n*factorielle(n-1)
- finsi
- fin

Complexité d’un algorithme récursif

- $T(0) = 2$
- $T(n) = T(n-1)+2$
- $\Rightarrow T(n) = 2 + 2n$
- Complexité en $O(n)$.
- Les calculs de complexité d’algorithmes récursifs induisent naturellement des suites.

Quelques équations récursives classiques

- Coût arithmétique (un seul appel qui s'effectue en temps constant)

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

- Si

$$T(n) = an + T(0) = O(n)$$

- Alros

Quelques équations récursives classiques

- Équations linéaires simples (un seul appel qui coûte f (n))
- Si $T(n) = T(n-1) + f(n)$
- Alors $T(n) = T(0) + f(0) + f(1) + \cdots + f(n) = T(0) + \sum_{i=0}^{n} f(i)$
- Comme C (0) est constant on peut généralement l'ignorer.
- En particulier
- $\sum_{i=1}^{n} 1 = n = \mathcal{O}(n)$
- $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \mathcal{O}\left(n^2\right)$
- $\sum_{i=1}^{n} i^k = \mathcal{O}\left(n^{k+1}\right)$
- et de manière plus générale

Quelques équations récursives classiques

- Coût géométrique (a appels; a>1)
- SI $$T(n) = a \times T(n-1)$$
- Alors $$T(n) = a^n \times T(0) = \mathcal{O}\left(a^n\right)$$

Quelques équations récursives classiques

- Coût arithmético-géométrique
- Si $T(n) = aT(n-1) + b$
- Alors $T(n) = (T(0) - l)\, a^n + l$
- ,avec $a \neq 1$ et $l = \frac{b}{1-a}$

Résolution de récurrences

- Il arrive souvent que pour analyser la complexité d'un algorithme on doive résoudre une récurrence.

Récurrences linéaires homogènes à coefficients constants 1/2

- Récurrences linéaires homogènes à coefficients constants:

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0$$

- Polynôme caractéristique:

$$p(x) = a_0 x^k + a_1 x^{k-1} + \cdots + a_k$$

$$p(x) = \prod_{i=1}^{k} (x - r_i)$$

Récurrences linéaires homogènes à coefficients constants 2/2

- Si ces racines sont distinctes

$$t_n = \sum_{i=1}^{k} c_i r_i^n$$

- Si ces racines ne sont pas toutes distinctes

$$t_n = \sum_{i=1}^{\ell} \sum_{j=0}^{m_i - 1} c_{ij} n^j r_i^n$$

- où on a ℓ racines r_i de multiplicité m_i

Example 1

- Soit la suite récurrente homogène

$$t_n = \begin{cases} 0 & \text{si} n = 0 \\ 5 & \text{si } n = 1 \\ 3t_{n-1} + 4t_{n-2} & \text{sinon} \end{cases}$$

$$t_n - 3t_{n-1} - 4t_{n-2} = 0$$

- L'équation caractéristique de cette suite

$$x^2 - 3x - 4 = (x+1)(x-4)$$

$$c_1 + c_2 = 0 \quad n = 0$$

$$-c_1 + 4c_2 = 5 \quad n = 1$$

- donc $t_n = 4^n - (-1)^n$

Example 2

- Soit la suite récurrente homogène

$$t_n = \begin{cases} n & \text{si } n = 0, 1 \text{ ou } 2 \\ 5t_{n-1} - 8t_{n-2} + 4t_{n-3} & \text{sinon} \end{cases}$$

$$t_n - 5t_{n-1} + 8t_{n-2} - 4t_{n-3} = 0$$

- L'équa$x^3 - 5x^2 + 8x - 4 = (x-1)(x-2)^2$

$$t_n = c_1 1^n + c_2 2^n + c_3 n 2^n$$

$$t_n = 2^{n+1} - n2^{n-1} - 2$$

- donc

Récurrences linéaires non-homogènes à coefficients constants:

- Récurrences linéaires non homogènes de la forme :

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = b^n p(n)$$

- Où b est une constante et p(n) un polynôme en n de degré d . Son polynôme caractéristique est

$$\left(a_0 x^k + a_1 x^{k-1} + \ldots + a_k\right)(x-b)^{d+1}$$

- Note : Une combinaison linéaire générale de solutions n'est plus nécessairement une solution : il faut la réinjecter dans la récurrence,ce qui précise certaines constantes.

Example

- Soit:

$$t_n = \begin{cases} 1 & \text{si } n = 0 \\ 4t_{n-1} - 2^n & \text{sinon} \end{cases}$$

- Polynôme caractéristique $(x-4)(x-2)$
- Solution $c_1 4^n + c_2 2^n$
- et donc, $c_1 = 0$ et $c_2 = 1$.
- Finalement $t_n = 2^n$ $\quad t_n \in \Theta\left(2^n\right)$

Récurrences linéaires non-homogènes à coefficients constants:

- De la même façon, une récurrence de la forme

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b_1^n p_1(n) + b_2^n p_2(n) + \ldots$$

- où les degré b_i sont des constantes et les $p_i(n)$ des polynômes en n de d_i respectivement ont pour polynôme caractéristique :

$$\left(a_0 x^k + a_1 x^{k-1} + \ldots + a_k\right)(x - b_1)^{d_1+1}(x - b_2)^{d_2+1}\ldots$$

Résolution de récurrence
Changement de variables

- Il arrive qu'il soit plus facile de commencer par faire un changement de variables lorsque l'on veut résoudre une récurrence

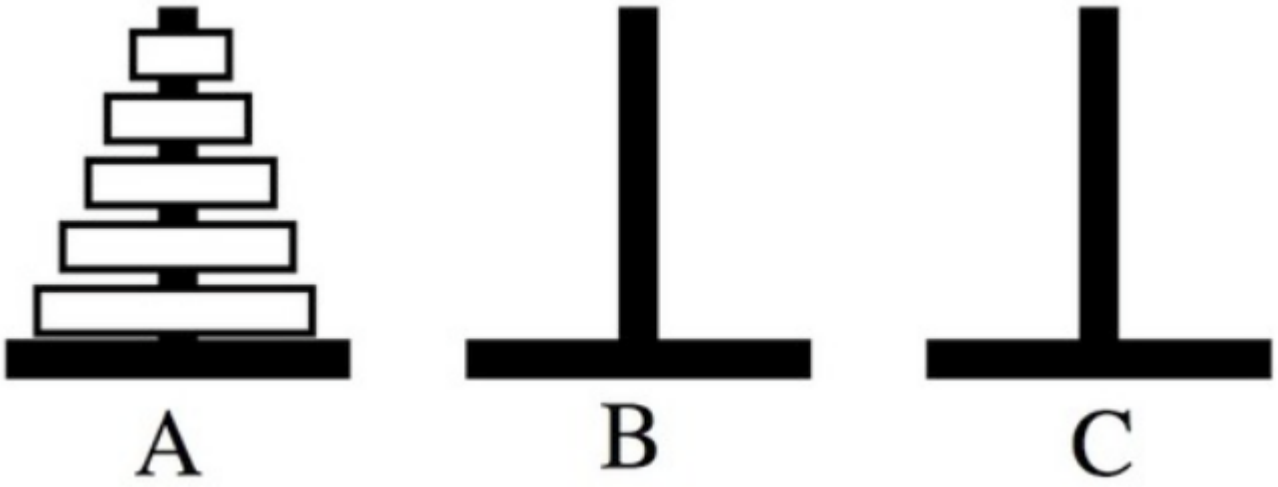
$$T(n) = 4T(n/2) + n^2$$

- Par exemple, soit $t_i = T\left(2^i\right)$.
- en faisant le changement de variables
- on obtient la récurrence linéaire non-homogène

$$t_i - 4t_{i-1} = 4^i$$

- On résout cette récurrence de la façon habituelle et ensuite on exprime la solution obtenue pour les t_i en fonction des T(n) en utilisant le fait que $n = 2^i$ et donc que $i = \log_2 n$

tours de Hanoï



tours de Hanoï

H(n,A,B,C)

début

si n = 1 alors écrire(A, ‘→’,B)

sinon

H(n-1,A,B,C);

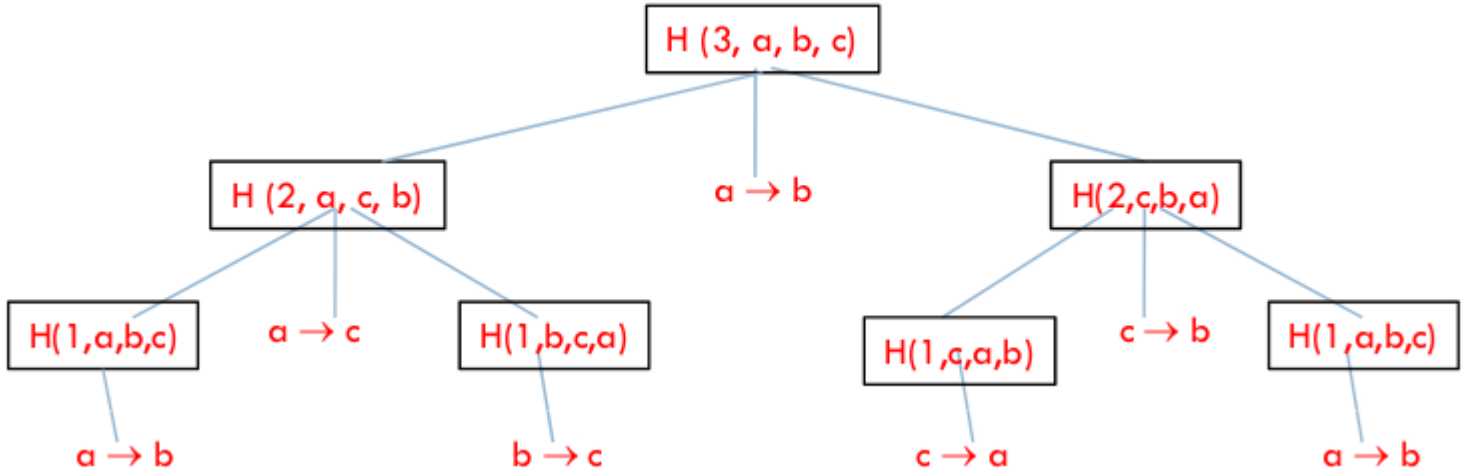
écrire(A,’→’,B)

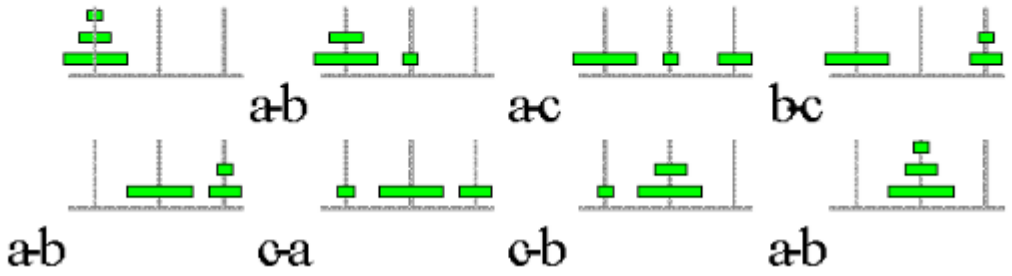
H(n-1,C,B,A);

fsi

fin

tours de Hanoï





tours de Hanoï

- Soit T(n) le temps pour déplacer les n disques. T(n) vérifie l'équation :
- $T(1) = 1$
- $T(n) = 2\,T(n-1) + 1$
- On a $T(2) = 2 + 1$
- $T(3) = 2(2 + 1) + 1 = 2^2 + 2 + 1$
- On montre, par récurrence, que
- $T(n) = 1 + 2 + \ldots + 2^{n-1} = 2^n - 1$