Stratégies de Résolution de problèmes

ECL – 2A – MI

2022-2023

Chapitre 1 (transparents)

Analyse d'algorithmes

Éléments de calcul de complexité

Alexandre Saidi (ECL-LIRIS)

I- Quelques références bibliographiques

Ce ne sont que quelques références !

1. *Optimisation Combinatoire :* M. Sakarovitch, Hermann, 1984
2. *Design and Analysis of Algorithms : cours notes :* Samir Khuller, 1994
3. *Graphes & Algorithmes :* M. Gondran & M.Minoux, Eyrolles, 1995
4. *Foundation of Algorithms :* R. Neapolitan, K. Naimipour, D.C. Health & Co.1996
5. *The algorithm design Manual :* S.S. Skiena, Springer Verlag, 1997
6. *Complexity of Algorithms* : Peter Gacs, LNCS 1999
7. *The Art of Computer Programming :* 2nd edition, Knuth, 1999
8. *Data structures & Algorithm Analysis in C++* : M.A; Weiss, Addison-Wesley, 1999
9. *The algorithm design* : G. Kleinberg & E. Tardos, Addison Wesley 2005
10. *The Standard Template Library,* Alexander Stepanov (le site du SGI)
11. *Algorithm + Data structure = Programs*, N. Wirth, Prentice Hall 1976
12. *The Design and Analysis of Computer Algorithms*, Aho, Hopcroft, Ullman, 1974 Ad.w.
13. Notes personnelles *" Cahier de Complexité "* (A.S.) 1995-2005

II- Objectifs du cours et éléments abordés

Rigueur dans les algorithmes (preuve)

Calcul du coût (*complexité*) des algorithmes

On entends souvent : **" Mon programme marche donc il est juste !"**

⇨ Pouvez-vous le prouver (esprit Mathématiques, rigueur de l'Ingénieur) ?

La Connaissance de quelques méthodes importantes aident.

Éléments abordés dans ce module :

- Sensibilisation aux exigences des (bons !) algorithmes
- Analyse, preuve, calcul de **Complexité** des algorithmes

  --> outils mathématiques de mesure de complexité
- Différents stratégies de résolution de problèmes :

✓ Stratégies de résolution : Diviser / séparer pour Régner, Gloutonne, Programmation Dynamique, B&B, A*, etc.
✓ Graphes et algorithmes remarquables, TAS & Arbres, Dijkstra, MST, etc.
✓ Récursivité & analyse récursive

- A travers l'Étude de quelques problèmes combinatoires connus.

III- Pourquoi faut-il des algorithmes ?

- Certains problèmes (p. ex. TSP, tournée de véhicules, …) n'ont pas de formulation Mathématique.
- Ils peuvent avoir une 'Modélisation Mathématique (p.ex. Coloration minimale),
- Pour d'autres, on n'a pas de 'formule' pour calculer une sortie / décision / etc.

  → Il faut les résoudre par un algorithme (+ heuristiques dans certains cas)
- Cet algorithme va décrire la démarche de résolution du problème.
- Il doit satisfaire quelques conditions (pour être 'utile' et 'effectif').
- Il décrira le QUOI (données, méthode) : **La Logique**

  et le COMMENT (description des étapes) : **Le Contrôle**

IV- Qu'est-ce qu'un algorithme ?

***Exemple** : trier une séquence de N entiers **S[1..N]** dans l'ordre ascendant.*

→ La solution pour *S de taille 6 = [10, 7, 11, 5, 13, 8]* → [5, 7, 8, 10, 11, 13]

→ S est une **instance** (1 exemplaire) du problème traité par un algorithme de tri.

Un **algorithme** de TRI décrit la méthode sous forme d'une suite d'instructions précises exécutée pour S de taille N.

Parmi une pléthore de méthodes, on dira par exemple (*Tri Insertion*):

- *Soit Si , i=1..N un élément de S, commencer à partir de S_1 jusqu'à $S_{(N-1)}$*
- *Itérer en comparant S_i aux $S_{j, j=i+1..N}$ et permuter S_i et S_j si $S_i > S_j$.*
- Recommencer avec S_2 .... S_N

→ Notion d'assertion/invariant : ici, après une itération sur i : $S_i = \min(S_{i..N})$

IV.1- Exemple 2 :Tours de HANOI

- Un **algorithme** pour HANOI décrit la méthode de déplacement de N disques :

Les N disques au départ sur le pieu 'A' doivent aller sur B en s'aidant de C

Exemple d'algorithme de solution :

Hanoi(N, Dep, Arr, Aux) :

- Commencer par déplacer N-1 disques depuis Dep vers Aux en s'aidant de Arr
- Il restera un disque sur Dep : déplacez-le sur Arr
- Recommencer : déplacer les N-1 disques restants de Aux vers Arr à l'aide de Dep

Appel initial : ***Hanoi(N, Dep=A,Arr = B, Aux =C)***

IV.2- Exemple 3 : mariages stables (stable matching)

Construction / vérification d'un couplage stable dans un graphe.

Un **couplage** ("mise en correspondance" = matching) **est *instable*** s'il contient 2 personnes A et B qu'on n'a pas pu connecter (mettre ensemble) alors qu'ils se préféraient (p/r aux conjoints que l'on leur a affectés).

P. Ex., on a 'fêté' ces mariages (f,F,g,G) : F est mariée avec g
G est mariée avec f

Mais 'ça' ne tiendra pas (instable) car : F préfère G à g
G préfère F à f

- Commet vérifier qu'un couplage est stable?

- Un couplage stable existe-il toujours? Si oui, peut-on le trouver (par un algo?)

☞ Voir solutions et applications en section "Compléments" : Mariages stables (stable matching)

V- Ce que l'on peut attendre des algorithmes

- Un algorithme a un domaine de définition :

  l'ensemble potentiellement infini des instances (In/Out) du problème.

- Un algorithme doit être juste

  Un algorithme juste doit fonctionner sur toutes les instances d'un pb.

  De plus, les solution trouvées doivent toutes être justes

  --> Ex. : un algorithme de recherche d'un élément dans un tableau

- Un algorithme complet doit trouver toutes les solutions justes.

- Selon les cas, on pourra apporter les preuves de :

  Existence / Unicité (éventuelle) de la solution, un Délai donné, Etc...

Autres (non moins importantes) :

- Un algorithme doit "finir " (**terminaison**) en un temps fini.

- Notion de décidabilité / *calculabilité* (traçabilité, en anglais : *tractable*)

  ☞ *Ex. : un programme pour détecter si un programme boucle ?*

- **Itératif / Récursif** : indépendant de la complexité, transformable

  ☞ Souvent on a le choix entre récursif/itératif, parfois pas.

  (e.g. Hanoï, …).

VI- Propriétés

Algorithme = Logique + Contrôle (une logique + une stratégie)

Programme = Algorithme + Structures de données

Un algorithme décrit une solution dans un espace d'états (initiaux / finaux).

- ☞ Un état final = une solution
- ☞ Le **contrôle** dit : comment aller d'initial au final (étant donné la Logique).
- ☞ Parfois, une **logique juste** ne donne pas forcément un algorithme :

Exemple : $(n+1)! = (n+1).n!$ $d'où\ n! = \frac{(n+1)!}{n+1}$

Même si le "Contrôle" de l'algo. du calcul du **n!** respecte sa logique.

VII- Idée de la complexité

- Il y a souvent plusieurs méthodes (algorithmes) pour résoudre un problème.

  --> Dans **Logique + Contrôle :**

  plusieurs Contrôles possibles pour une même logique.

Exemple de TRI de S : on peut permuter S jusqu'à tomber juste !!

Un Exemple (trivial):

On a 15 boîtes de vis de différentes longueurs rangées dans un bloc de rangement avec 15 tiroirs.

→ **Comment faut-il** ranger ces vis afin de les retrouver facilement ?

1- Naïve : ranger n'importe comment, rechercher de gauche à droite.

→ En moyenne : 8

1'-Variante de 1 : ranger les vis selon la fréquence des demandes.

→ Toujours 8 en moyenne.

2- Naïve 'Las Vegas' : Naïve mais plus *démocratique*

Recherche aléatoire (au lieu de gauche-droite) → tjs 8 en moyenne.

3- Tri : on range les vis selon la longueur de la plus petite à la plus grande,

Puis on cherche par la méthode Dichotomique.

→ En moyenne 3,26 comparaison

☞ Voir détails de cet exemple en section "Compléments" Détails Ex. "Idée de la complexité"

VIII- Complexité et Contrôle

Définition : la **complexité** d'un algorithme A est une définition $C^A(N)$ donnant le nombre d'instructions caractéristiques exécutées par A (dans le pire des cas = le plus défavorable) pour une donnée de taille N.

Cette mesure donne un **ordre de croissance** de l'algorithme A.

N.B. la fonction de complexité d'un algorithme en temps est noté **$T^A(N)$ ou $T(N)$**

- L'analyse de la complexité peut être

Pessimiste (le cas pire : W(n)) : → $T_{max}(n) = max(Temps(d)|$ d donnée de taille n)

Optimiste (meilleur cas : B(n)) : → $T_{min}(n) = min(Temps(d)|$ d donnée de taille n)

Moyenne (*A(N)*) → $T_{moy}(n) = \sum p(d)*Temps(d)$ | p(d) : proba de d

- La connaissance du **cas pire** est **critique** pour les applications **Temps Réels** (contrôle aérien, robots, automatismes, freinage, systèmes d'alarme, etc.) où **il faut borner le temps nécessaire aux calculs.**

N.B. : accompagné de la complexité du cas pire, la complexité moyenne (≠ la moyenne des complexités) donne une indication intéressante.

N.B. : Problème Q-sort (Tri rapide)

Dans Q-sort, le cas *favorable* est $N.log(N)$ mais le cas pire est $O(N^2)$.

--> On considère quand-même le **cas pire** (e.g. si le tableau est déjà trié).

VIII.1- Exemple 1 : Comparatif recherche d'une "vis"

- Recherche séquentielle ou Dichotomique dans l'ensemble S (de vis) :

  Leur complexité : **O(N)** pour séquentielle et **O($\log_2$ N)** pour dichotomique.

Taille de S	Recherche séquentielle	Recherche binaire
128 (2^7)	128	8
1024 (2^{10})	1024	11
1.048576 (2^{20})	1048576	21
4.294.967.296 (2^{32})	4.294.967.296	33

L'efficacité d'un algorithme de recherche binaire semble évidente.

VIII.2- Exemple 2 : séquence de Fibonacci

fib(0) = 0,
fib(1) = 1,
fib(n) = fib(n-1) + fib(n-2)

1 - La solution récursive naïve (cf. la déf.) a beaucoup de calculs redondants.

--> Le nombre de termes calculés **T(n)** est de l'ordre de $2^{n/2}$ (voir plus loin).

2- Il existe une solution récursive (de complexité $O(2^n)$, voir + loin)

3- Une autre définition récursive :

$fib(2n) = fib(n)^2 + 2fib(n).fib(n-1)$ *cas pair*

$fib(2n+1) = fib(n)^2 + fib(n+1)^2$ *cas impair*

4- La solution utilisant un tableau pour stocker les résultats (PrD) :

Le coût T(n) = n+1

VIII.2.1- Comparatif Fibonacci

Comparatif du **temps d'exécution** de Fib(n) sur une machine qui calcule chaque terme en une nanoseconde (10^{-9} sec.) pour deux des méthodes précédentes

n	$2^{n/2}$	Temps Algorithme Linéaire (n+1 termes)	Temps Algorithme Récursif (naïf, non Pr. D.)
60	$1.1 * 10^{9}$	61 ns	1 s
80	$1.1 * 10^{12}$	81 ns	18 min.
100	$1.1 * 10^{15}$	101 ns	13 jours
120	$1.2 * 10^{18}$	121 ns	36 années
200	$1.3 * 10^{30}$	201 ns	$4 * 10^{13}$ années

☞ Déduction :

Pour une même Logique, certaines **stratégies** (contrôles) peuvent mettre à mal les capacités (ressources, temps, espace) des machines récentes.

Même logique : définition de fib(N), Contrôle varié : v. p. précédente.

--> V. "Compléments" : fib(200) prendra 32×10^{20} ans sur la machine la plus puissante en 2022.

Exemples :

fib(N),

prime(p) **:** $(2^{p-1} \% p == 1)$ pour p > 2,

Tri par permutation successives !,

$n! = \frac{(n+1)!}{n+1}$, ...

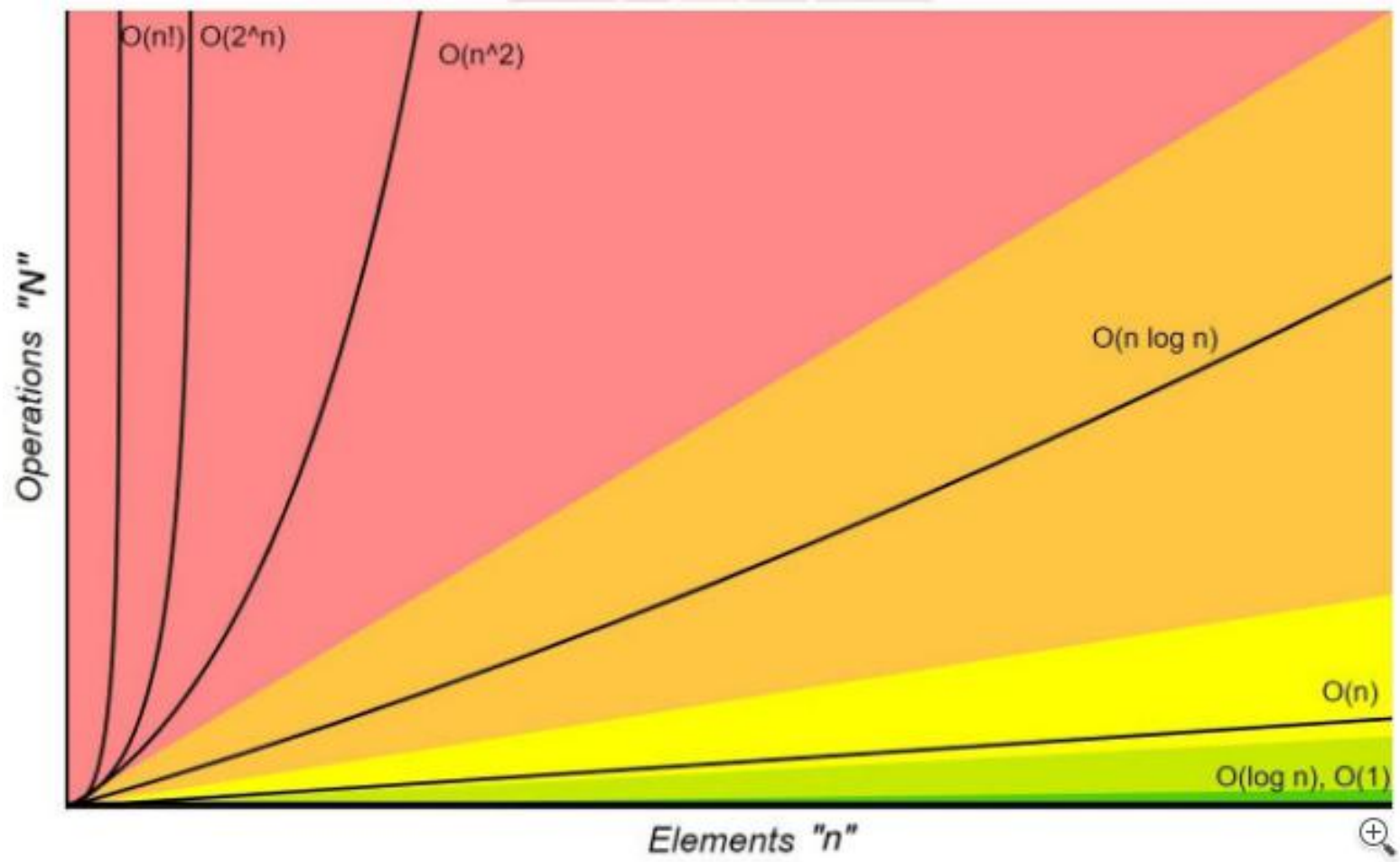
☞ **Remarque** : dans ce cours, on ne traitera pas la complexité en espace.

IX- Tableau des croissances relatives

Dans l'ordre croissant de complexité (avec des exemples) :

c	*constante*	→ accès à un élément dans un tableau
log N	*logarithmique*	→ couper un ensemble en 2 parties égales, recouper ....
$\sqrt{N}$		→ utilisation dans le calcul des nombres premiers
$\log^2$ N	*log. au carré*	→ recherche dans un B-arbre, dans une forêt
N	*linéaire*	→ parcours linéaire d'un ensemble de données
N.logN		→ couper un ens. en 2 + parcours de chaque partie
N^2	*quadratique*	→ parcourir un ensemble une fois par élément d'un autre ensemble de la même taille (cf. tri bulle)
N^3	*cubique*	→ triple boucle (voir exemple réf. des sommes)
2^N	*exponentiel*	→ générer tous les sous-ens. d'un ens. de données
N!	*exponentiel*	→ toutes les permutations d'un ens. (ex. tri bête !)

IX.1- Comparaisons des courbes des croissances usuelles



IX.2- Comparaison de qq. ordres de de complexité

--> Le temps de calcul suppose une microseconde par instruction de haut niveau

(i.e. pas en assembleur) sur une machine avec un processeur >=386/486.

Les cases vides : > 1000 milliards d'années (> l'age estimé de l'univers).

complexité \	20	50	100	200	500	1000
$10^3 . n$	0.02 s	0.05 s	0.1 s	0.2 s	0.5 s	1 s
$10^3 . n \log_2 n$	0.9 s	0.3 s	0.6 s	1.5 s	4.5 s	10 s
$100 n^2$	0.04 s	0.25 s	1 s	4 s	25 s	2 mn
$10 n^3$	0.02 s	1 s	10 s	1 mn	21 m	27 h
$n^{\log n}$	0.4 s	1.1 h	220 j	12500 ans	5.10^{10} ans	
$n^{n/3}$	0.001 s	0.1 s	2.7 h	3.10^6 ans		
2^n	1 s	36 ans				
3^n	58 m	2.10^{11} ans				
$n!$	77100 ans					

N.B. : pour se faire une idée, lancer le calcul récursif de **fib(100)** --> $O(5/3)^{100}$

X- Éléments d'analyse de la complexité

- Comment un algorithme se comporte et quelles ressources (temps et espace) il demande.
- Sous quelle forme le temps d'exécution augmente si la taille des données en entrée augmente.
- Cette analyse doit être **indépendante** des aspects d'un **contexte** particulier.
- Dans la **complexité** d'un algorithme, on :

  --> ne calcule pas le nombre de cycles du CPU (dép. d'une machine particulière).
  --> ne compte pas le nombre d'instructions exécutées dépendant d'un langage.
  --> ne tient pas compte de la classe du langage (impérative /fonctionnelle / logique) ni du compilateur employé.

- On décide en général d'une **opération de base caractéristique**.

  --> **L'opération de base** peut changer d'un algorithme à un autre.

Exemples d'opération de base pour différents algorithmes :

- nombre d'appels (récursifs) d'une fonction fib : $T(N) = O(2^N)$
- l'ajout d'un élément à un tableau pour *fib* linéaire : $T(n)=n$
- la comparaison de deux éléments pour le *tri* par échange : $T(n) = n.(n-1)/2$
- la multiplication de 2 valeurs pour la *multiplication de mat.* : $T(n) = n^3$

- On peut rendre un algorithme plus efficace (par une cst.) sans forcément modifier sa complexité (Exemple BE1, AES et la fonction *prometteur*).
- La **complexité** notée ***T(n)*** doit tenir compte de tous les cas possibles

  (*Every-case complexity* ≠ *Average-case complexity*).

XI- Sensibilité à la puissance des machines

Principe d'invariance *(opinionem dissimilis)* :

Malgré les différences technologiques, la complexité d'un même algorithme sur deux machines différentes *ne varie que* par un facteur constant.

Exemple (ce n'est pas ce que l'on croit !)

Soit T = le temps nécessaire pour exécuter un programme sur une machine M1.

Supposons disposer du même temps T pour exécuter le même programme sur une machine M2 10 fois plus rapide que M1.

Questions légitimes : de quel facteur (p/r à M1) peut-on augmenter la taille des données traitées sur M2 ?

Et de quel facteur (p/r à M1) augmenterait la vitesse d'exécution pour la même taille de données traitées sur M2 ?

- Soit ***n*** la taille des données sur M1, ***n'*** celle sur M2 pour la même durée **T**.

  - Pour une complexité linéaire, on a n' =10 n (10 fois plus de données)

  - Pour une complexité en n^2, on a $n'^2 = 10\ n^2$

    d'où **n'** = $\sqrt{10}n$ = **3,16 n**

  - Pour une complexité 2^n, on a $2^{n'} = 10\ 2^n$

    d'où **n' = n + log 10 = n + 3,3**

✍ Pour ce cas, on peut augmenter les données de seulement 3 !!

Détaillons :

- l'évolution de **n** pour M2 (pour le même temps t)
- l'évolution du temps **t** si **n** → 10n (sur la même M1)

Complexités	1	$\log_2(n)$	n	$n.\log_2(n)$	n^2	n^3	2^n
Évolution du temps t quand la taille des données n → 10n	1	log(10n) = t+3,3	10 t	(10+ε) t	10^2 t	10^3 t	t^{10}
volution de la taille n quand le temps alloué (sur M2) t → 10t	1	n^{10}	10 n	(10-ε)n	$n\sqrt{10}$= 3,16 n	2,15 n	n+3,3

La valeur de ε est négligeable devant la croissance de la fonction.

Quelques détails :

- dans la complexité **nlog(n)**, lorsque n → 10n, on aura :
  (10n)log(10n)=10n[log(n)+3,3]=10n.log(n)+33n
  =nlog(n)[10+ 33/log(n)]=n.log(n)[10+c] = t (10+c), c=33/ln(n)
- dans **log(n)**, lorsque t → 10t, on aura : log(n) → 10 log(n) = log(n^{10}) → n'= n^{10}
- dans **n log(n)**, lorsque t → 10t, on aura : n.log(n) → 10 n.log(n)
  = 10 n.log[10 n / 10] = 10nlog(10n) -10n*3,3=10nlog(10n) -ε(n.log(n))= (10-ε)t
- le cas de la complexité n^2 a été traité plus haut...

On constate que :

- Les progrès en puissance des machines sont négligeables face aux progrès en algorithmique

  ⇨ Les algorithmes remarquables sont plus rares.

- Parfois, ces progrès valent *révolution* (e.g. Fourrier) !

XII- Déf. de big-oh (upper bound, pessimiste)

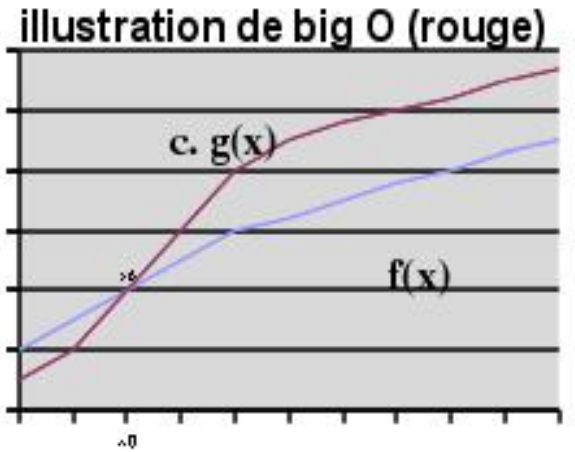
Idée informelle : la limite supérieure d'une fonction

(modulo un facteur constant)

Si **g(n)** est une limite supérieure de **f(n)**, il est alors

possible de trouver une valeur (n_0) telle que :

f(n) $\leq$ **c.g(n)**, pour tout n $\geq n_0$ et **c** une constante.



Définition : soit f et g deux fonctions de $\mathbb{R}$ dans $\mathbb{R}$.

On dit que f est d'ordre inférieur ou égal à g (ou d'ordre au plus g) si l'on peut trouver un réel x0 et un réel positif c tels que $\forall x \geq x0, f(x) \leq c.\ g(x)$.

--> **g** devient "+grand" que **f** à partir d'une certaine valeur $\mathbf{x_0}$ à un facteur **c** près.

--> On remarque que des cas **d'égalité** sont possibles.

Remarques : on écrit

- *f est* ***O(g)***, *f est en O(g)* ou *f=O(g)* prononcé "**grand-O de g**".
- Pour le calcul de complexité, on utilise plutôt des fonctions de $\mathbb{N} \rightarrow \mathbb{R}$.

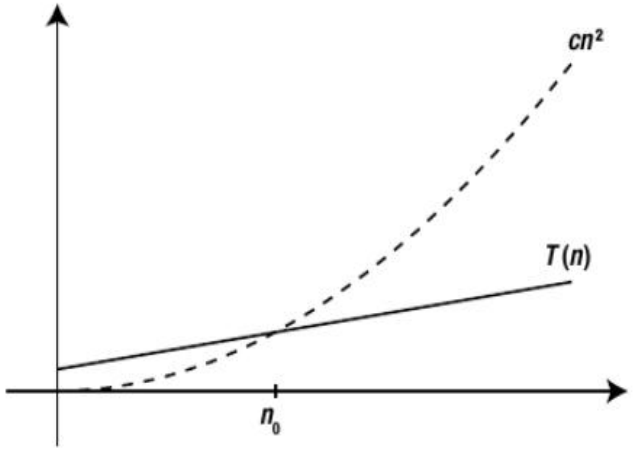
  → O(g) est un ensemble de fonctions, celles d'ordre au-plus g :

  → On préfère écrire $\mathbf{f} \in \mathbf{O(g)}$.

- **Un exemple simple :** $5N = O(0.5\ N^2)$ car $5N \leq 0.5\ N^2$ àpd. $N \geq 10$.

- O(f(n)) veut dire qu'une fonction est ≤ à une autre fonction modulo une constante au sens "asymptotique" lorsque **n** croît (par f) vers l'infini (cf. $n \geq n_0$).

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



XII.1- Exemples de calcul de big-oh

I- Recherche du maximum dans un tableau de taille n

Cette recherche peut prendre $(2n-1)$ opérations de base (comparaisons).

On dira que $2n-1$ est $O(n)$ car on peut trouver une constante réelle $c > 0$ et

$(n_0 \geq 1)$ tels que $2n-1 \leq c.n$, pour tout $n \geq n_0$.

--> Ici, on peut choisir par exemple $c=2$ et $n_0=1$.

Ce choix n'est qu'une possibilité car tout réel $c > 2$ et tout $n_0 \geq 1$ convient.

II- Montrer que $20\ n^3 + 10\ n \log n + 5$ est $\mathbf{O(n^3)}$

→ $20\ n^3 + 10\ n \log n + 5 \leq c.\ n^3$ pour $n \geq 1$

→ Ici, par ex. $c=35$ et $n_0=1$ conviennent (35 choisi "empiriquement" via 20+10+5)

III) Montrer que $n^2 + 2n + 1$ is $O(n^2)$

Il faut montrer : $n^2 + 2n + 1 \leq c * n^2$ avec $n >= n_0$.

Avec $n_0 >= 1$ un entier positif, posons $n_0=1$

$\rightarrow$ $n^2 + 2n + 1 \leq c \, . \, n^2$ avec $n \geq 1$. Divisons par n^2

On a : $(n^2 + 2n + 1) / n^2 \leq c$ quand $n \geq 1$.

Sachant que $(n^2 + 2n + 1) / n^2 \leq (n^2 + 2n^2 + n^2) / n^2$ avec $n \geq 1$

On cherche c tq $(n^2 + 2n + 1) / n^2 \leq (n^2 + 2n^2 + n^2) / n^2 \leq c$ avec $n \geq 1$.

--> $(n^2 + 2n^2 + n^2) / n^2 \leq c$ lorsque $n \geq 1$.

On simplifie sur n^2 : $(1 + 2 + 1) / 1 \leq c$ pour $n \geq 1$

--> $4 \leq c$ et $n \geq 1$

D'où : $n^2 + 2n + 1 \leq c * n^2$ pour $c = 4$, $n \geq 1$

IV) Montrer que $n^2/2 - 3n \in O(n^2)$

Il faut trouver un réel n_0 et un réel $c>0$ tels que

$$\forall n \geq n0,\ n^2/2 - 3n \leq c.\ n^2$$

Divisons par n^2 (avec $n >= 1$), on a :

--> $1/2 - 3/n \leq c$ pour $n \geq 1$ et $c \geq 1/2$

Et $0 \leq n^2/2 - 3n \leq c.\ n^2$ pour $n \geq n_0 = 6$ et $c \geq 1/2$

D'où $n^2/2 - 3n \in O(n^2)$

(V) Montrer que $f1(n) = 5n^3 + 2n^2 + 22n + 6$ est $O(n^3)$.

Pour c=6 et n_0=10, on a $5n^3 + 2n^2 + 22n + 6 \leq 6n^3$ pour $n \geq 10$.

En plus, f1 (n) = $O(n^4)$ car $n^4 >= n^3$ --> n4 est une borne sup. Asymp. pour f1 .

N.B. : f1 (n) n'est pas $O(n^2)$ car qq soit c et n_0, la déf de O(.) ne se vérifie pas.

(VI) Montrer que $3n \log_2 n + 5n \log_2 \log_2 n + 2$ est $O(n \log n)$

Pour n>=2, c=6, on a $3n \log_2 n + 5n \log_2 \log_2 n + 2 <= n \log n$

N.B. sachant que $\log_b n = \log_2(n)/\log_2(b)$, on peut omettre la base dans O(n log n).

tout polynôme de degré k est $O(n^k)$.

XIII- La fonction Ω (lower bound : optimiste)

La fonction **Oméga** (Ω) place une **borne inférieure asymptotique.**

Définition de la fonction oméga :

Pour une fonction de complexité f(x), Ω(f(x)) est un ensemble de fonctions de complexité g(x) pour lequel il y a un réel positif c et une constante non négative x0

tels que $\forall x \geq x0$, $f(x) >= c.\ g(x)$.

N.B. : on dira que **f(n) = Ω(g(n))** si **g(n) = O(f(n))**

La notation Ω est plus précise mais le big-oh est plus simple à calculer.

→ cherche à donner une meilleure estimation de la complexité.

XIII.1- Exemples de calcul de Ω

Montrer que $n^2/2 - 3n \in \Omega(n^2)$

Par définition :

$$\Omega(n^2) = \{f : N \rightarrow N \text{ tq. } \exists n0 \geq 0, \exists c > 0, \forall n \geq n0, f(n) \geq cn^2 \geq 0\}$$

On peut trouver les constantes n_0 et c telles que :

$0 \leq c.n^2 \leq 1/2n^2 - 3n$ pour $n_0 \geq 7$ et $c \geq 1/14$

D'où :

$1/2n^2 - 3n \in \Omega(n^2)$

XIV- La fonction Θ (égalité entre O et Ω)

L'ordre Θ dit que 2 fonctions sont asympt†. égales (modulo un facteur cst).

Si une fonction **f** est à la fois **O**(n^2) et Ω(n^2), sa courbe se place à la fois au-dessous (O(.)) d'une certaine fonction quadratique **pure** (de la forme ax^2+bx+c) et au-dessus (Ω(.)) d'une certaine autre fonction quadratique (pure).

--> La fonction f est donc aussi bonne / mauvaise (que ces 2 fonctions).

--> Donc, l'accroissement de f est similaire à une fonc. quadratique pure : ordre Θ

Définition : pour une fonction de complexité f(x), Θ(f(x)) = O(f(x)) ∩ Ω(f(x)).

Ce qui veut dire que Θ(f(x)) est l'ensemble de fonctions de complexité g(x) pour lequel il y a les réels positifs c et d et une constante non négative x0 tels que :

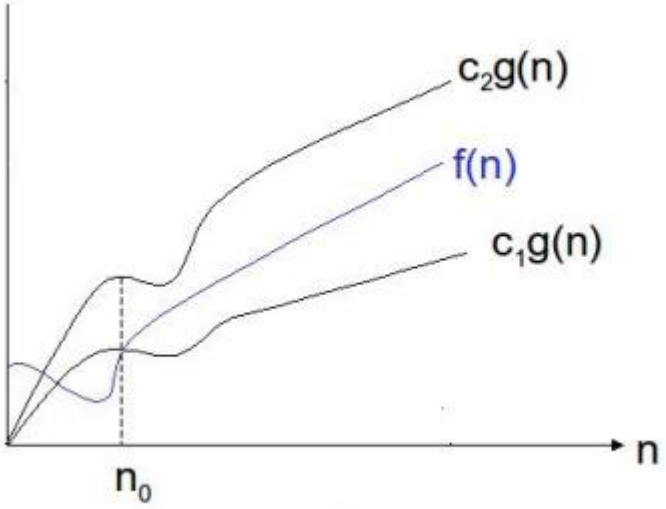
$$\forall x \geq x0,\ c.\ g(x) \leq f(x) \leq d.\ g(x).$$

Exemples de Θ (Théta)

$f \in \Theta(g)$

- $f(n) = n(n-1)/2$ est à la fois $O(n^2)$ et $\Omega(n^2)$
  (cf. sections précédentes)
  → dans ce cas, $f(n) = \Theta(n^2)$

- $n^2 + 10n \in \Theta(n^2)$ car $n^2 + 10n \in O(n^2)$
  et $n^2 + 10n \in \Omega(n^2)$ pour n grand.
  On a également $n^2 \in \Theta(n^2 + 10n)$ cf (3) ci-dessous



g est une **borne approchée asymptotique** de *f*

Remarques :

(1) Si $f(n) = \Theta(g(n))$, alors $f(n)$ est à la fois $O(g(n))$ et $\Omega(g(n))$.

(2) Si $f(x) = \Theta(g(n))$, on dira que $g(x)$ est l'ordre de $f(x)$.

(3) Si l'on a $O(N)$ et $N = \Theta(x)$, alors on a $O(x)$.

(4) Si $g(n) \in \Theta(f(n))$ ssi $f(n) \in \Theta(g(n))$ (Θ porte l'égalité)

XIV.1- Exemples de calcul de Θ

(I) Montrer que $1/2n^2 - 3n$ est $\Theta(n^2)$.

On doit montrer que :

$c_1 n^2 <= n^2/2 - 3n <= c_2 n^2$ pour tout $n >= n0$.

On divise par n^2 :

$c_1 <= 1n/2 - 3/n <= c_2$

--> Pour $c_2 >= 1/2$, on a l'inégalité à droite.

--> Pour la partie gauche, $n >= 7$ et $c1 <= 1/14$.

Donc, avec $n_0=7$, $c1 = 1/14$ et $c_2 = 1/2$, nous aurons

$T(1/2n^2 - 3n) = \Theta(n^2)$.

(II) Montrer que $6n^3 \neq \Theta(n^2)$.

Facile ! : preuve par contradiction / réfutation.

Supposons que c'est le cas.

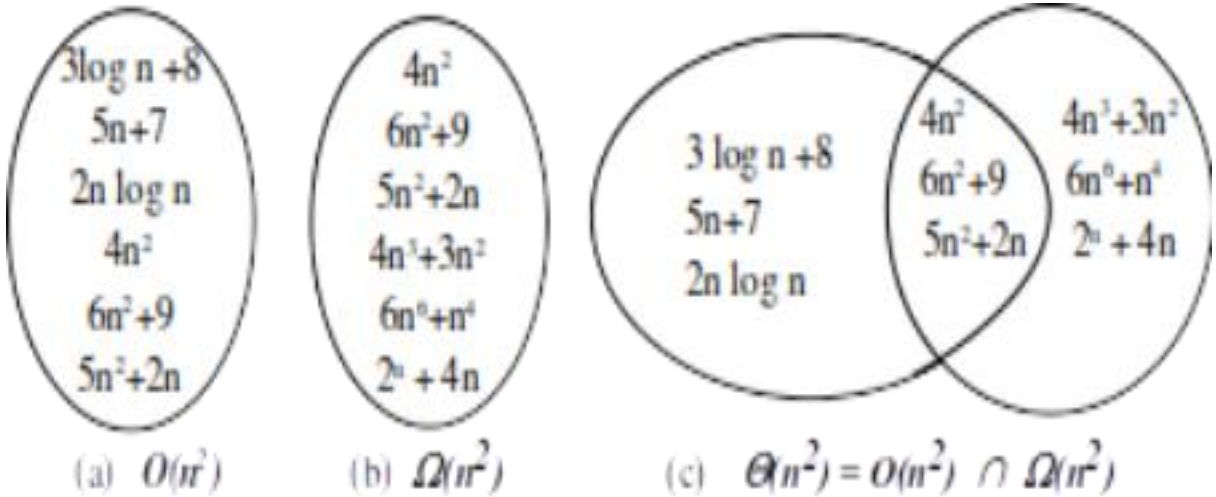
Il existe alors c_2 and n_0 tels que $6n^3 <= c_2 n^2$.

En divisant par n^2 (pour $n>= 1$), on aura $6n <= c_2$.

Or, pour une constante c_2 donnée, la relation $6n <= c_2$ n'est pas valide pour tout n.

XV- Illustration des 3 fonctions O, Ω, Θ

Quelques exemples courants de ces 3 fonctions de complexité pour n^2.



- N.B. : Réflexivité :
  - On a $f \in O(f)$ et $f \in \Omega(f)$ donc $f \in \Theta(f)$.
  - Il n'y a pas de réflexivité pour little-oh : f(n) n'est jamais o(f(n))

- Voir section "Compléments" pour les famille "little-oh & oméga" : o(N) ω(N)

XVI- Calculs : le Modèle (de la machine)

Nécessité : il faut un modèle de calcul pour estimer la complexité.

En générale, on considère que les hypothèses suivantes sont vérifiées :

- **Ordinateur normal** (séquentiel) avec des instructions simples (+,-,*,/).
- **Chaque opération prend une unité de temps.**
- Les entiers sont de taille fixe (32 / 64 bits)
- L'opération caractéristique de base n'est pas compliquée comme la multiplication ou inversion de matrices ou le tri (qui demandent >> 1 unité).
- La quantité de RAM est **illimitée** (dans la limite du raisonnable !).
- On est **indépendant du langage** : certains langages peuvent diminuer/augmenter le temps (paramètres par réf, copie de tableau en paramètre, etc.), voir + loin.

XVII- Règles basiques et empiriques de calcul

On repère les instructions *caractéristiques* (on dit aussi *fondamentales*)

- **Règle des opérations simples :**

```
if (A > B):
    A = A - 1
    B = 2 * B
```

Trois instructions (test+2 instruction) si A>B et, une (le test) instruction sinon.

Le traitement est **O(1) car le nombre d'opérations est borné par une cste.**

- **Règle des Séquences** :

On additionne les complexités on prend le maximum selon la règle de somme

- **Règle de la Conditionnelle :** *if (Cond) {S1} else {S2}*

  O(conditionnelle) = max (O(S1), O(S2)) + cout(Cond)

Règle de la boucle (for, while, ...) :

(Le temps d'exécution des instructions à l'intérieur) * (le nombre d'itérations).

Boucles imbriquées :

Le temps de la boucle interne multiplié par le nombre d'itérations externes

Exemple (de complexité = $O(N^2)$)

```
for i in range(n):
        for j in range(n):
            k = k + 1
```

Autres règles générales :

- On analyse **de l'intérieur vers l'extérieur.**
- Pour un appel de fonction, on analyse d'abord la fonction

Cas de la récursivité :

Parfois, une fonction récursive est en fait une itération «cachée»

Exemple (O(N)) où l'opération caractéristique est "*" (ou un appel à "fact")

```
def fact(n: int)→ long :
  if (n <= 1): return 1
  return  n * fact(n-1)
```

Remarque : le calcul de complexité formelle des algorithmes récursifs est souvent plus simple ! Leur preuve de justesse aussi !

Remarques :

- Si T1 (N) = $O(f(N))$ et T2 (N) = $O(g(N))$, alors

  (a) T1 (N) + T2 (N) = $O(f(N) + g(N)) = O(\max(f(N), g(N)))$,

  (b) T1 (N) $*$ T2 (N) = $O(f(N) * g(N))$.

- Si T(N) est polynomial de degré k, alors

  $T(N) = \Theta(N^k)$.

- $\log^k N = O(N)$ pour toute constante k.

  --> Cela veut dire que la croissance de la fonction est très lente.

XVIII- A propos de la complexité moyenne A(n)

Un exemple : la recherche de X dans S[1:n]

- La probabilité pour que S[k]=X, $1 \leq k \leq n$ est 1/n,
- Pour arriver à l'indice k=1..n, on aura fait les comparaisons :

$$A(n) = \sum_{k=1}^{n} \left(k . \frac{1}{n}\right) = \frac{1}{n} \cdot \sum_{k=1}^{n} k = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}$$

opérations de base si X est dans S

Mais pour être complet, il faut tenir compte du cas où X n'est pas dans S :

- Si $Pr(X \in S) = p$ alors $Pr(X \notin S) = 1-p$ *et pour k donné,* $Pr(S[k] = X) = p/n$
- On fait *k* comparaisons pour arriver à k tel que S[k]=X

  et *n* comparaisons si X n'est pas dans S.

La complexité moyenne (dépendra de p) :

$$A(n) = \sum_{k=1}^{n} \left(k.\frac{p}{n}\right) + n(1-p) = \frac{p}{n}.\frac{n(n+1)}{2} + n(1-p) = n\left(1-\frac{p}{2}\right) + \frac{p}{2}$$

→ si **p=1**, A(n) = (n+1)/2

→ si p= $\frac{1}{2}$, A(n) = 3n/4 + $\frac{1}{4}$ --> seul $\frac{3}{4}$ de S est recherché en moyenne

→ si p= 0, A(n) = O(n) = n.

Remarques sur A(n) :

On a supposé une **probabilité identique** pour chaque élément du tableau.

--> à changer si l'on connaît une distribution différente de taille/ valeurs :

↳ c-à-d. telle valeur est plus fréquente que telle autre...

☞ La complexité moyenne est un **très bon indicateur** mais son calcul est souvent **difficile** (et donc **rare**) car les probas nécessaires ne sont pas toujours connues/disponibles.

XIX- Complexité et Stratégies

Selon la stratégie (Logique et Contrôle) choisie, la complexité de la solution à un problème peut grandement varier.

Ci-dessous, deux exemples. Voir également la section "Compléments".

- Calcul de la médiane
- Sous séquence de somme maximale
- ...

XIX.1- Exemple 1 : calcul de la médiane d'une suite

But : Calculer la médiane M d'une suite d'entiers S de taille N telle que *la moitié des nombres de S soient plus petits que M et l'autre moitié plus grande.*

N.B. : on notera |S| = N

Sol 1- Trier S puis de choisir le milieu.

→ Complexité : celle de l'algorithme de tri (au mieux *O(N.Log(N))*).

Sol 2- Construire un **TAS** (v. + loin, O(N)) puis retirer k éléments (k= ⌊ N /2 ⌋).

→ Complexité : $O(N.\log_2 N)$ pour N >= 4

Sol 3- Mieux (et on généralise) : trouver le $k^{ième}$ plus petit élément de S.

Pour la médiane, k= ⌊ |S| /2 ⌋ —> O(N)

→ Point fort : ne trier que la plus petite sous séquence de S contenant le kième plus petit élément.

XIX.1.1- Meilleure Solution : un stratégie Diviser pour régner

Pour une séquence S, la méthode se généralise pour k = 1..|S|
(k = |S|/2 pour la médiane)

Principe : pour trouver le kième plus petit élément, il suffit de repérer le sous tableau susceptible de contenir cet élément et de traiter celui-ci.

Ce qui donne :

Choisir **aléatoirement** v ∈ S sera (en général, on choisit V= le 1e élément).

Pour une valeur **v** appartenant à S, on peut scinder S en 3 sous tableaux :

- Sg : contenant les éléments plus petits que **v**
- Sv : contenant les éléments = **v** (contenant **v** lui-même)
- Sd : contenant les éléments de S plus grands que **v**

Par exemple : S = <2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1> , |S|=11 et v = 5 (p. ex.) :
→ Sg=<2, 4, 1>, Sv = <5, 5>, Sd=<36, 21, 8, 13, 11, 20>

Exemple : si k=8, on sait que le 8e plus petit élément ne peut être que dans Sd
car pour **v** = 5, la somme des tailles de Sg et Sv : |Sg| + |Sv| = 5.

On notera :

selection(S, k=8) = selection(Sd, k=3) sachant |Sg| = 3 et |Sv|=2.

Généralisation : selection(S,k) =	selection(Sg , k)	si k =< |Sg|
	= **v**	si |Sg| < k =< |Sg| + |Sv|
	= selection(Sd, k-(|Sg| + |Sv|))	si k > |Sg| + |Sv|

On répétera ce traitement (récursif) sur le sous-tableau concerné jusqu'à aboutir à un singleton qui est le résultat recherché (voir l'algo. en section "Compléments").

☞ La complexité (Θ) souffrira d'un découpage déséquilibré en Sd et Sg.

☞ **Aho & al** montrent : en moyenne, 2 divisions suffisent pour trouver la médiane :

V. *[Aho & al], [Wirth]* pour 2 algorithmes de complexité moyenne O(n)

XIX.1.2- Détails du déroulement pour cet exemple

La médiane de S=<2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1> avec |S|=11 k=6

Appel initial de l'algo : selection(S,6) (6e plus petit élément)

selection(S,6) :V1=2, Sg1=<1> Sv1=<2> Sd1=<36, 5, 21, 8, 13, 11, 20, 5, 4>

k > |Sg1| + |Sv1| → selection(Sd1, 6 - |Sg1| - |Sv1|) = selection(Sd1, 4)

selection(Sd1,4) : V2=36, Sg2=<5, 21, 8, 13, 11, 20, 5, 4> Sv2=<36> Sd2=<>

k <|Sg2| → selection(Sg2, 4)

selection(Sg2,4) : V3=5, Sg3=<4> Sv3=<5,5> Sd3=<21, 8, 13, 11, 20>

k > |Sg3| + |Sv3| → selection(Sd3, 4 - |Sg3| - |Sv3|) = selection(Sd3, 1)

selection(Sd3,1) : V4=21, Sg4=<8, 13, 11, 20> Sv4=<21> Sd4=<>

k <|Sg4| → selection(Sg4, 1)

selection(Sg4,1) : V5=8, Sg5=<> Sv5=<8> Sd5=<13, 11, 20>

|Sg5| < k =< |Sg5| + |Sv5| car 0 < 1 =< 1

→ le résultat est = V5 = 8 (8 est le 6^{e} plus petit élément de S)

Vérification :

si on trie S, on aura S_trié= <1, 2, 4, 5, 5, 8, 11, 13, 20, 21, 36> dont '8' est le 6e PP.

XIX.1.3- L'algorithme Aho & al.

Rappel : dans cet algorithme (Aho, Hopcraft, Ulman), la recherche de la médiane revient

à la recherche d'une valeur (A : choix random) dans la séquence.

```
def select(k,S) :  # S : la séq. de nombre, k : le rang du kème plus petit élément dans S
 if |S|==1 : return S[0]
 else:
      A = Choisir un élément aléatoire dans S (en général on prend le 1er)
      Soient   S1 : les éléments de S < A
               S2 : les éléments de S = A
               S3 : les éléments de S > A
      if (|S1| >= k) : return select(k,S1)
      elif (|S1|+|S2| >= k) : return A
      else : return select(k-|S1|-|S2|, S3)

def find_median(S) :  return select(|S|/2, S)
```

Quelques applications du calcul de la médiane :

- En traitement du signal, on l'utilise pour supprimer les "anomalies" (valeurs aberrantes) dans une distribution lisse ds valeurs.

  --> Un filtre médian (filtre non linéaire de bruit) permet de maintenir des fréquences élevées.

- Il peut également être utilisé pour estimer la moyenne d'une liste de valeurs numériques, indépendamment de fortes valeurs aberrantes.

- En traitement d'image, un filtre médian est calculé par une *convolution* avec un noyau <2N+1, 2N+1> (la taille W de fenêtre du noyau pour N=0, 1, ...).

  --> Si n=0 --> W=1 (pas de changement), si n=1 --> W=3 (un masque 3x3)
  Ex de noyau pour la **netté** : [[0,-1,0], [-1, 5, -1], [0,-1,0]]

  --> Pour chaque pixel de l'image en entrée, le pixel de sortie est remplacée par la médiane des valeurs de pixel du noyau.

  --> Voir par ex. "The Image Processing Handbook, J. C. Russ, CRC Press, 2e éd. 1995".

XIX.2- Exemple 2 : séquence de somme maximale

- Soit une suite d'entiers *A1* ... *An* (peut contenir des entiers négatifs)

  --> **But** : trouver la séquence contiguë de somme maximale.

- Hypothèse : on pose la somme maximum = 0 si tous les entiers sont négatifs.

- Exemples : pour (on commence à l'indice 1) :

  → la séquence : **-2, 11, -4, 13, -5, -2** La réponse = 20 (A2..A4)

  → la séquence : **4, -3, 5, -2, -1, 2, 6, -2** La réponse = 11 (A1..A7)

- Ce problème a de multiples applications (vi=oir + loin)

- Ex. intéressant car il y a plusieurs algorithmes possibles (4 cités ici) :

  --> **O(N^3)** à **O(N)**. Comparons les temps en fonction de N ⮒.

Rappel : le tableau suivant donne une idée comparative de ces temps en **sec.** :

↓N / T(N)→	$O(N^3)$	$O(N^2)$	O(N. Log N)	O(N)
10	0.00103	0.00045	0.00066	0.00034
100	0.47015	0.01112	0.00486	0.00063
1000	448.77	1.12330	0.05843	0.00333
10000	NA	111.13	0.68631	0.03042
100000	NA	NA	8.01130	0.29832

- $O(n^3)$: naïf, basique sans opti. (aucun résultat/somme intermédiaire conservé)
- $O(n^2)$: optimisation dans les sommes partielles
- O(n log n) : approche dichotomique : stratégie "Diviser-pour-régner"

  la somme recherchée est soit dans la moitié gche, soit dte, soit au milieu des 2.
- O(n) : approche récursive

Détails : soit une séquence $X_1, X_2, X_3, \ldots X_n$

- Les 2 premières approches sont triviales :

1) **Solution $O(n^3)$** : calculer max(

X_1 et les autres : $X_1, (X_1+X_2), (X_1+X_2+X_3), \ldots, (X_1+X_2+X_3+\ldots+X_n),$
X_2 et les autres : $X2, (X2+X3), (X2+X3+X4), \ldots., (X2+X3+X4+\ldots+Xn),$
....
$X_{n-1}, (X_{n-1} + X_n),$
Et enfin X_n $X_n)$

2) Solution $O(n^2)$: optimiser le calcul des sommes partielles :

P. Ex. : $X_1+X_2+X_3+\ldots+X_n = (X_1+X_2+X_3+\ldots+X_{n-1}) + X_n$

--> conserver la somme partielle précédente et y ajouter un nouvel élément !

3) Solution Dichotomique (de complexité **$O(n \log n)$**)

la somme recherchée est soit dans la moitié gche., soit dte., soit au milieu des 2.

- Les 2 premiers cas (soit à gche, soit à dte) ont une solution récursive directe

- Le 3e peut être obtenu en calculant la somme de

  la plus grande somme dans la 1e moitié qui inclut le dernier élément de cette moitié

  et **la plus grande somme** de la 2e moitié qui inclut le 1e élément de cette moitié

  (raison : continuité de la sous-séquence).

- Ces deux sommes sont ensuite être additionnées (étape Régner).

- On prend la max des 3 sommes (1/2 gche, 1/2 dte, à cheval au milieu)

- L'expression de la complexité (v. + loin) du 3e méthode :

  $T(1)=1$ et

  $T(n) = 2. T(n/2) + n$

Exemple1 :

la séquence A :

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*<== indices*
4	**-3**	**5**	**-2**	**-1**	**2**	**6**	**-2**	<== éléments

- Meilleure somme de la 1e moitié = 6 (A1..A3)

  celle de la 2e moitié = 8 (A6..A7).

- Meilleure somme de la 1e moitié qui inclut le dernier élément de cette moitié

  est 4 (A1..A4) ;

- Pour l'autre moitié, ce sera 7 (A5..A7)

  On fait la somme des deux :

  ➠ la meilleure somme : 4+7=11 (A1..A7).

- **Exemple 2 :**

la séquence précédente (tableau B):

1	*2*	*3*		*4*	*5*	*6*	*<== indices*
-2	**11**	**-4**		**13**	**-5**	**-2**	<== éléments

- La meilleure somme de la 1e moitié = 11 (B[2]) et

celle de la 2e moitié = 13 (B[4]).

- La meilleure somme de la 1e moitié qui inclut le dernier élément de cette

moitié est 7 (B2..B3)

et pour l'autre moitié, c'est 13 (B4).

leur somme = 7+13=20 (B2...B4)

➠ la meilleure somme =**20 (B2..B4).**

- **Exemple 3 :**

nombre impaire de données (C) : *1 2 3 4 5 6 7 <== indices*

4 -3 5 -2 -1 2 -6 <== élés.

- La meilleure somme de la 1e moitié = 6 (C1..C3)

et celle de la 2e moitié = 2 (C6).

- La meilleure somme de la 1e moitié qui inclut le dernier élément de cette moitié

est 4 (C1..C4)

et pour l'autre moitié, c'est 1 (C5..C6).

Leur somme donne 4+1=5 (C1..C6)

➠ la meilleure somme est dans la **première moitié** =6 (C1..C3).

Et enfin, pour la 4e solution (le cas linéaire) :

```
def max_sous_sequence(V) :
    max_finissant_ici = meilleur_max_jsq_ici = 0
    for x in V :
        max_finissant_ici = max(0, max_finissant_ici + x)
        meilleur_max_jsq_ici = max(meilleur_max_jsq_ici, max_finissant_ici)
    return  meilleur_max_jsq_ici
```

Explication : considérons l'indice de chaque élément x dans le vecteur V :

A chaque indice, on calcule le max (somme > 0) du sous-séquence finissant à cet indice.

Cette sous-séquence est soit vide (somme=0) soit contient un élément de plus que la sous-séquence finissant à l'indice précédent.

NB : la version ci-dessus de l'algorithme ne donne pas les indices.

N.B. : L'algorithme ci-dessus ne donne pas les indices.

Pour pallier cet inconvénient (de l'algo. précédent) :

- Modifier l'algorithme pour trouver le début et la fin de la sous-séquence de somme max.
- Pour le début, on doit conserver le dernier indice de la dernière somme négative calculée.

⇨ Du fait de conserver la meilleur somme max jusqu'à la position précédente, on est en présence de la méthode PrD (voir le chapitre 2).

Notes sur les applications de max_sum :

- Le pb. d'appariement *de sous-intervalles :*

→ E.g., pour estimer le max de vraisemblance d'un motif dans une image.

→ En dimension 2, l'intervalle de *max_sum* est un estimateur du MLE de certains motifs dans les images numériques.

- Utilisation en méthodes de segmentation dans l'analyse des séquences de protéine (et l'ADN).

→ On y utilise (également) *min_sum* pour supprimer des séquences alignées de Protéine (source : plusieurs papiers recherche en bio-info).

XX- Propriétés des limites de fonctions

- On peut calculer le taux relatif de croissance de 2 fonctions **f** et **g** en calculant **Lim** $_{n \to \infty} \frac{f(N)}{g(N)}$

Si **f** et **g** admettent des limites, on a alors les propriétés suivantes :

- Si $\lim_{n \to \infty} \frac{f}{g} = c > 0$, f et g sont du même ordre, f=O(g) et g=O(f) --> f= Θ(g).
- Si $\lim_{n \to \infty} \frac{f}{g} = 0$, alors f = o(g) et f est d'ordre inférieur à g.
- Si $\lim_{n \to \infty} \frac{f}{g} = +\infty$, f est d'ordre supérieur à g.

On note f=Ω(g) qui est équivalent à g=o(f)

- On pourra (également) utiliser la règle de l'"**Hôpital**" :

  Si f et g sont définies sur [a,b[, dérivables en a telles que f(a)=g(a) ≠0

  Alors $\lim_{n\to\infty} \frac{f(N)}{g(N)} = \lim_{n\to\infty} \frac{f'(N)}{g'(N)}$ si la limite existe.

- Plus précisément, si g'(N) ≠0 Alors

  Si $\lim_a f(N) = \lim_a g(N) = 0$ et $\lim_{n\to\infty} \frac{f'(N)}{g'(N)} = L$ Alors $\lim_{n\to\infty} \frac{f(N)}{g(N)} = L$

  Si $\lim_a f(N) = \lim_a g(N) = +\text{infini}$ et $\lim_{n\to\infty} \frac{f'(N)}{g'(N)} = L$ Alors $\lim_{n\to\infty} \frac{f(N)}{g(N)} = L$

Utilisation des limites

On peut utiliser les propriétés des limites pour trouver une idée de la complexité (**lorsqu'elle est difficile à trouver**).

XX.1- Exemple trivial d'utilisation des limites

- On a vu : **pour a > 0, $a^n \in o(n!)$** car

  avec les propriétés des limites : $\lim_{n \to \infty} \frac{a^n}{n!} = 0$

  --> **a^n** est d'ordre inférieur à **n!**.

- On peut montrer que **log n ∈ o(n)** :

$$lim \frac{log\ x}{x} = lim \frac{\frac{d(log\ x)}{dx}}{\frac{dx}{dx}} = lim \frac{\frac{1}{xlog\ 2}}{1} = lim \frac{1}{(xlog\ 2)} = lim \frac{1}{x} = 0$$

XX.2- Application : approximation de la complexité par programme

- On utilise la règle "limite" pour calculer la complexité empirique par programme.

Pour montrer qu'un algorithme est O(g(N)), on calcule les valeurs de T(N)/f(N) pour un intervalle de N habituellement espacé par un **facteur de 2**, puis on étudie la limite de T(N)/g(N) .

Rappel des règles de la limite :

- Si $\lim_{n \to \infty} \frac{f}{g} = c > 0$, f et g sont du même ordre, f=O(g) et g=O(f) --> f= Θ(g).
- Si $\lim_{n \to \infty} \frac{f}{g} = 0$, alors f = o(g) et f est d'ordre inférieur à g.
- Si $\lim_{n \to \infty} \frac{f}{g} = +\infty$, f est d'ordre supérieur à g.

On note f=Ω(g) qui est équivalent à g=o(f)

- Dans la méthode utilisée, f(N) = T(N) = le temps empirique observé.
- On calcule également différentes fonc. de complexité g(N) (e.g. log N, N^2, N^3…)
- Calculer ensuite f(N)/g(N) et observez sa limite (empirique)

→ Si la limite converge vers une cste > 0 alors g est une estimation de la complexité **Θ**

→ Si f(N) est surestimée, les valeurs convergent vers 0, c-à-d. **f=o(g)**.

→ Si f(N) est sous estimée, les valeurs divergent (tendent vers l'infini) :

g=o(f)

- Le taux de convergence / divergence signale aussi le degré de justesse de T(n).

XX.3- Exemple 1

1- Montrer que la probabilité pour que deux entiers distincts et aléatoires I, J $\leq$ N soient premiers entre eux approche $6/\Pi^2 = 0.608$ (pour N grand).

2- Estimer la complexité de la solution.

On propose l'algorithme (simplifié) suivant :

```
Fonction proba_prime(N)=          // renvoie un double
  Rel=0                           // Rel : nombre de fois où i et j sont premiers
  Total_essais=0                  // Total : nombre d'essais
  Pour (i=1; i <= N; i++)         // Simulation du tirage aléatoire sans remise
        Pour (j=i+1; j <= N; j++)
          Total_essais ++
          Si (pgcd(i,j) == 1) Rel++   // On sait : pgcd est O(log N)
  renvoyer Rel/Total_essais
```

- La complexité : 2 Boucles ($O(N^2)$) et la complexité de pgcd est $O(\log N)$

  --> $N^2 \log N$ --> $O(N^2 \cdot \log(N))$

XX.3.1- Estimation empirique de la complexité de l'exemple

- Le tableau obtenu sur une machine Centrino 1,6 avec 512Mo de mémoire

Taille	Ln	Ln * Ln	N	N ln	N+N Ln	N * N	N * N * N
50	0,0000702961	0,0000179692	0,0000055000	0,0000014059	0,0000011197	0,0000001100	0,0000000022
75	0,0001528667	0,0000354064	0,0000088000	0,0000020382	0,0000016549	0,0000001173	0,0000000016
112	0,0003401506	0,0000720887	0,0000143304	0,0000030371	0,0000025060	0,0000001279	0,0000000011
168	0,0008050408	0,0001571129	0,0000245536	0,0000047919	0,0000040094	0,0000001462	0,0000000009
252	0,0016952926	0,0003065945	0,0000371984	0,0000067274	0,0000056970	0,0000001476	0,0000000006
378	0,0175406665	0,0029555146	0,0002754021	0,0000464039	0,0000397125	0,0000007286	0,0000000019
567	0,0194570992	0,0030687692	0,0002175750	0,0000343159	0,0000296409	0,0000003837	0,0000000007
8500	0,7441067491	0,0822415378	0,0007920641	0,0000875420	0,0000788294	0,0000000932	0,0000000000
1275	0,0227007100	0,0031746130	0,0001273145	0,0000178045	0,0000156201	0,0000000999	0,0000000001
1912	0,0419171490	0,0055476013	0,0001656496	0,0000219232	0,0000193608	0,0000000866	0,0000000000
2868	0,0851187651	0,0106914718	0,0002362838	0,0000296788	0,0000263669	0,0000000824	0,0000000000
4302	0,1815649459	0,0217005521	0,0003531204	0,0000422048	0,0000376990	0,0000000821	0,0000000000
6453	0,3971829320	0,0452769414	0,0005399362	0,0000615501	0,0000552517	0,0000000837	0,0000000000
9679	0,8774637252	0,0956080934	0,0008320189	0,0000906564	0,0000817491	0,0000000860	0,0000000000
14518	1,9480021329	0,2032737924	0,0012858511	0,0001341784	0,0001214999	0,0000000886	0,0000000000
21777	4,4913626210	0,4496484276	0,0020600849	0,0002062434	0,0001874746	0,0000000946	0,0000000000

Rappel : la vitesse de la machine n'a pas d'effet car les rapports $T(N)/g(N)$ et les constantes C (de la définition de big-Oh) sont négligées.

Ici, chaque colonne représente **$T(N) / g(N)$ et on étudie la limite**.

Ici, $N^2 \log(N)$ n'est pas prise en compte, on sera $\Omega(N^2)$.

Deuxième tableau (des écarts types) : vers une décision automatique

Taille	Ln	Ln * Ln	N	N ln	N+N Ln	N * N	N * N * N
100	0,0002692626	0,0000584696	0,0000124000	0,0000026926	0,0000022122	0,0000001240	0,0000000012
200	0,0010877038	0,0002052923	0,0000288150	0,0000054385	0,0000045750	0,0000001441	0,0000000007
300	0,0186707681	0,0032734012	0,0003549800	0,0000622359	0,0000529522	0,0000011833	0,0000000039
400	0,0175603142	0,0029308884	0,0002630300	0,0000439008	0,0000376216	0,0000006576	0,0000000016
500	0,0195906159	0,0031523494	0,0002434960	0,0000391812	0,0000337504	0,0000004870	0,0000000010
600	0,0209661521	0,0032775336	0,0002235317	0,0000349436	0,0000302195	0,0000003726	0,0000000006
700	0,0204386137	0,0031198844	0,0001912786	0,0000291980	0,0000253313	0,0000002733	0,0000000004
800	0,0231905766	0,0034692481	0,0001937750	0,0000289882	0,0000252160	0,0000002422	0,0000000003
1000	0,0255741544	0,0037022380	0,0001766600	0,0000255742	0,0000223401	0,0000001767	0,0000000002
1200	0,0304775258	0,0042986171	0,0001800733	0,0000253979	0,0000222585	0,0000001501	0,0000000001
1500	0,0377177202	0,0051574707	0,0001838920	0,0000251451	0,0000221204	0,0000001226	0,0000000001
2000	0,0513148277	0,0067511493	0,0001950195	0,0000256574	0,0000226743	0,0000000975	0,0000000000
2500	0,0671867215	0,0085872094	0,0002102688	0,0000268747	0,0000238291	0,0000000841	0,0000000000
3500	0,1237624829	0,0151660077	0,0002885617	0,0000353607	0,0000315006	0,0000000824	0,0000000000
6500	0,4037839057	0,0459913735	0,0005453914	0,0000621206	0,0000557685	0,0000000839	0,0000000000
10000	0,9451901502	0,1026227167	0,0008705523	0,0000945190	0,0000852618	0,0000000871	0,0000000000
15000	2,0837667777	0,2167022598	0,0013358064	0,0001389178	0,0001258318	0,0000000891	0,0000000000
25000	5,7303253575	0,5658669007	0,0023211556	0,0002292130	0,0002086126	0,0000000928	0,0000000000
45000	18,5556198471	1,7318365074	0,0044180592	0,0004123471	0,0003771471	0,0000000982	0,0000000000
90000	76,7338421380	6,7265750823	0,0097260699	0,0008525982	0,0007838822	0,0000001081	0,0000000000
150000	213,5554605627	17,9181458481	0,0169682493	0,0014237031	0,0013134956	0,0000001131	0,0000000000

La colonne N*N représente le minimum d'écart type par rapport aux autres.

--> A titre d'indication, une fois ces valeurs harmonisées (toute valeur multipliée par $\frac{1}{min}$ où *min* est le minimum ≠ 0 de la colonne), les écarts types seront :

30749140888, 4628790867, 103855, 15560, 19636, **9**, 1361751

avec la valeur 9 pour la colonne N*N d'où $O(N^2)$.

XX.4- Exemple 2

- Tableau Tri Bulle (Bubble-Sort) de complexité $O(N^2)$. Colonne N^2 quasi cste.

Taille	Ln(N)	Ln^2(N)	N	N.Ln(N)	N+N.Ln(N)	N^2	N^2.Ln(N)	N^3	2^N
20	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000
26	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000
33	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000
42	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000
54	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000
70	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000
91	0,221687241324	0,049145232966	0,010989010989	0,002436123531	0,001994064805	0,000120758363	0,000026770588	0,000001327015	0,000000000000
118	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000
153	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000
198	0,189097864044	0,035758002186	0,005050505051	0,000955039717	0,000803163260	0,000025507601	0,000004823433	0,000000128826	0,000000000000
257	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000
334	0,344166490267	0,059225286511	0,005988023952	0,001030438594	0,000879151372	0,000017928215	0,000003085145	0,000000053677	0,000000000000
434	0,329324112275	0,054227185463	0,004608294931	0,000758811319	0,000651529184	0,000010618191	0,000001748413	0,000000024466	0,000000000000
564	0,473555534160	0,074751614644	0,005319148936	0,000839637472	0,000725168316	0,000009431115	0,000001488719	0,000000016722	0,000000000000
733	0,606322822134	0,091906841160	0,005457025921	0,000827179839	0,000718299495	0,000007444783	0,000001128485	0,000000010157	0,000000000000
952	0,729015468197	0,106292710574	0,005252100840	0,000765772551	0,000668328227	0,000005516913	0,000000804383	0,000000005795	0,000000000000
1237	0,983084711277	0,138065078507	0,005658852061	0,000794732992	0,000696864827	0,000004574658	0,000000642468	0,000000003698	0,000000000000
1608	1,760862314384	0,238510468478	0,008084577114	0,001095063628	0,000964430591	0,000005027722	0,000000681010	0,000000003127	0,000000000000
2090	2,616116547153	0,342203289414	0,009569377990	0,001251730405	0,001106936642	0,000004578650	0,000000598914	0,000000002191	0,000000000000
2717	4,173367445764	0,527787752647	0,012145748988	0,001536020407	0,001363574971	0,000004470279	0,000000565337	0,000000001645	0,000000000000
3532	6,487450192701	0,794094528354	0,015005662514	0,001836763928	0,001636454208	0,000004248489	0,000000520035	0,000000001203	0,000000000000
4591	10,199418624677	1,209629538156	0,018732302331	0,002221611550	0,001986068066	0,000004080223	0,000000483906	0,000000000889	0,000000000000
5968	16,447808935155	1,891821110261	0,023961126005	0,002756000157	0,002471705476	0,000004014934	0,000000461796	0,000000000673	0,000000000000
7758	26,237986797065	2,929497664523	0,030291312194	0,003382055529	0,003042371667	0,000003904526	0,000000435944	0,000000000503	0,000000000000
10085	173,449822979231	18,814784922781	0,158552305404	0,017198792561	0,015515739260	0,000015721597	0,000001705383	0,000000001559	0,000000000000
13110	70,350259876718	7,420028582791	0,050877192982	0,005366152546	0,004854170322	0,000003880793	0,000000409318	0,000000000296	0,000000000000
17043	114,024794809692	11,702658714125	0,065188053746	0,006690418049	0,006067676739	0,000003824917	0,000000392561	0,000000000224	0,000000000000
22155	207,379336672363	20,725874351166	0,093658316407	0,009360385316	0,008509891068	0,000004227412	0,000000422495	0,000000000191	0,000000000000
28801	310,571546092015	30,246059969265	0,110725322037	0,010783359817	0,009826384174	0,000003844496	0,000000374409	0,000000000133	0,000000000000
37441	637,575253131689	60,545457760789	0,179322133490	0,017028798727	0,015551953250	0,000004789459	0,000000454817	0,000000000128	0,000000000000
48673	702,129568954109	65,054886724688	0,155692067471	0,014425442626	0,013202209375	0,000003198736	0,000000296375	0,000000000066	0,000000000000
63274	1117,841984700321	101,114314837251	0,195309289756	0,017666687497	0,016201208380	0,000003086723	0,000000279209	0,000000000049	0,000000000000
82256	1594,067060805795	140,848611182641	0,219327465474	0,019379340848	0,017806034844	0,000002666401	0,000000235598	0,000000000032	0,000000000000
106932	2352,946581879067	203,191456570499	0,254806793102	0,022004138910	0,020254995096	0,000002382886	0,000000205777	0,000000000022	0,000000000000
139011	3559,018965707158	300,534225407817	0,303191833740	0,025602426899	0,023608826823	0,000002181064	0,000000184176	0,000000000016	0,000000000000
180714	5407,664551152306	446,741970390010	0,362218754496	0,029923882771	0,027640431101	0,000002004376	0,000000165587	0,000000000011	0,000000000000
234928	8066,525657665044	652,260309202863	0,424636484370	0,034336161112	0,031767441665	0,000001807518	0,000000146156	0,000000000008	0,000000000000
305406	12292,510568139573	973,325191905950	0,508329895287	0,040249735002	0,037296579110	0,000001664440	0,000000131791	0,000000000005	0,000000000000
397027	19167,282689962205	1486,785616013455	0,622375808194	0,048277025719	0,044801798158	0,000001567591	0,000000121596	0,000000000004	0,000000000000
516135	30073,991311182079	2286,278595098182	0,766460325302	0,058267684445	0,054151026577	0,000001485000	0,000000112892	0,000000000003	0,000000000000
670975	47661,954466643336	3552,491341985035	0,953025075450	0,071033875281	0,066106608707	0,000001420359	0,000000105867	0,000000000002	0,000000000000
872267	76850,315251526772	5618,185043586380	1,205163097996	0,088104118637	0,082102005833	0,000001381645	0,000000101006	0,000000000002	0,000000000000

Tableau de Merge_sort : O(N.log N) : Colonne N.log N quasi constante.

Taille	Ln(N)	Ln^2(N)	N	N.Ln(N)	N+N.Ln(N)	N^2	N^2.Ln(N)	N^3	2^N
20	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000
26	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000
33	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000
42	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000
54	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000
70	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000
91	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000
118	0,209613520641	0,043937828035	0,008474576271	0,001776385768	0,001468556475	0,000071818443	0,000015054117	0,000000608631	0,000000000000
153	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000
198	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000
257	0,180210179983	0,032475708970	0,003891050584	0,000701206926	0,000594137331	0,000015140275	0,000002728432	0,000000058912	0,000000000000
334	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000	0,000000000000
434	0,164662056138	0,027113592731	0,002304147465	0,000379405659	0,000325764592	0,000005309096	0,000000874207	0,000000012233	0,000000000000
564	0,157851844720	0,024917204881	0,001773049645	0,000279879157	0,000241722772	0,000003143705	0,000000496240	0,000000005574	0,000000000000
733	0,151580705534	0,022976710290	0,001364256480	0,000206794960	0,000179574874	0,000001861196	0,000000282121	0,000000002539	0,000000000000
952	0,145803093639	0,021258542115	0,001050420168	0,000153154510	0,000133665645	0,000001103383	0,000000160877	0,000000001159	0,000000000000
1237	0,280881346079	0,039447165288	0,001616814875	0,000227066569	0,000199104236	0,000001307045	0,000000183562	0,000000001057	0,000000000000
1608	0,406352841781	0,055040877341	0,001865671642	0,000252706991	0,000222560906	0,000001160244	0,000000157156	0,000000000722	0,000000000000
2090	0,523223309431	0,068440657883	0,001913875598	0,000250346081	0,000221387328	0,000000915730	0,000000119783	0,000000000438	0,000000000000
2717	0,505862720699	0,063974273048	0,001472211999	0,000186184292	0,000165281815	0,000000541852	0,000000068526	0,000000000199	0,000000000000
3532	0,734428323702	0,089897493776	0,001698754247	0,000207935539	0,000185258967	0,000000480961	0,000000058872	0,000000000136	0,000000000000
4591	0,830185236892	0,098458218222	0,001524722283	0,000180828847	0,000161656703	0,000000332111	0,000000039388	0,000000000072	0,000000000000
5968	1,150196429032	0,132295182536	0,001675603217	0,000192727284	0,000172846537	0,000000280765	0,000000032293	0,000000000047	0,000000000000
7758	1,451463099412	0,162057317612	0,001675689611	0,000187092434	0,000168301411	0,000000215995	0,000000024116	0,000000000028	0,000000000000
10085	1,952530840292	0,211798704572	0,001784828954	0,000193607421	0,000174661230	0,000000176979	0,000000019198	0,000000000018	0,000000000000
13110	2,320398376743	0,244738573945	0,001678108314	0,000176994537	0,000160107567	0,000000128002	0,000000013501	0,000000000010	0,000000000000
17043	2,873712200424	0,294936493245	0,001642903245	0,000168615396	0,000152920746	0,000000096398	0,000000009894	0,000000000006	0,000000000000
22155	3,697848412953	0,369569807708	0,001670051907	0,000166908076	0,000151742636	0,000000075380	0,000000007534	0,000000000003	0,000000000000
28801	4,674642274198	0,455255841494	0,001666608798	0,000162308332	0,000147904183	0,000000057866	0,000000005636	0,000000000002	0,000000000000
37441	5,792685499111	0,550085332649	0,001629229989	0,000154715032	0,000141297162	0,000000043515	0,000000004132	0,000000000001	0,000000000000
48673	7,690255241910	0,712530429948	0,001705257535	0,000157998382	0,000144600604	0,000000035035	0,000000003246	0,000000000001	0,000000000000
63274	9,497767308103	0,859119846084	0,001659449379	0,000150105372	0,000137653899	0,000000026226	0,000000002372	0,000000000000	0,000000000000
82256	12,635197034268	1,116421007656	0,001738475005	0,000153608211	0,000141137575	0,000000021135	0,000000001867	0,000000000000	0,000000000000
106932	15,889535400806	1,372159430725	0,001720719710	0,000148594765	0,000136782732	0,000000016092	0,000000001390	0,000000000000	0,000000000000
139011	149,379660482026	12,614066119687	0,012725611642	0,001074588777	0,000990913105	0,000000091544	0,000000007730	0,000000000001	0,000000000000
180714	25,279497580931	2,088408490014	0,001693283310	0,000139886769	0,000129212196	0,000000009370	0,000000000774	0,000000000000	0,000000000000
234928	31,939751148044	2,582652413668	0,001681366206	0,000135955489	0,000125784535	0,000000007157	0,000000000579	0,000000000000	0,000000000000
305406	117,503627658629	9,303977434594	0,004859105584	0,000384745642	0,000356516541	0,000000015910	0,000000001260	0,000000000000	0,000000000000
397027	52,669303708961	4,085501551085	0,001710211144	0,000132659249	0,000123109757	0,000000004308	0,000000000334	0,000000000000	0,000000000000
516135	85,144402683852	6,472829739634	0,002169974910	0,000164965373	0,000153310439	0,000000004204	0,000000000320	0,000000000000	0,000000000000
670975	86,162643502352	6,422146310825	0,001722865979	0,000128414089	0,000119506643	0,000000002568	0,000000000191	0,000000000000	0,000000000000
872267	183,494946159270	13,414500105973	0,002877559279	0,000210365572	0,000196034370	0,000000003299	0,000000000241	0,000000000000	0,000000000000
1133947	147,046006845825	10,547574697221	0,001807844635	0,000129676261	0,000120997163	0,000000001594	0,000000000114	0,000000000000	0,000000000000
1474131	179,884236229371	12,664711719696	0,001733224523	0,000122027307	0,000114001085	0,000000001176	0,000000000083	0,000000000000	0,000000000000
1916370	324,970165644709	22,464498736258	0,002453075346	0,000169575899	0,000158611427	0,000000001280	0,000000000088	0,000000000000	0,000000000000

Amélioration :

dans les tables ci-dessus, on peut utiliser les propriétés des limites

--> (f' et g' à la place de f et g).

Ce qui permet d'aller plus loin dans les fonction polynomiales envisagées

--> P. ex, n^4 à la place de n^3.

--> On sera quand même limité !

XXI- Outils de calcul de la complexité

XXI.1- Introduction : Équation de récurrence

- Analyse de la complexité des fonctions non récursives est relativement simple.

  → cf. les règles simples de calcul vues plus haut.

- Cette analyse (& la preuve) devient plus simple avec les fonctions récursives.

Une définition nécessaire : on appelle une récurrence la forme :

$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} + c = 0$ où c, k, a_i sont des constantes

= **l'équation de récurrence linéaire et homogène à coefficients constants**.

Par exemple, dans le schéma récursif de la factorielle :

```
def factorielle(n) :
   if (n==1) :   return 1
   return n*factorielle(n-1)
```

L'équation de récurrence sera (on regarde le cas de récurrence) :

$$t_n - t_{n-1} - 1 = 0 \text{ <--> } T(n) = T(n-1) + 1$$

On retrouve :

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} + c = 0 \qquad \text{où } c, k, a_i \text{ sont des constantes}$$

- Le but sera d'exprimer T(n) sous forme de récurrence.
- Puis de résoudre cette équation.

XXI.2- Propriétés des solutions à l'équation de récurrence

- Pour "résoudre" une récurrence, on cherche une forme fermée (ou close).

→ Une forme fermée pour T(n) est une équation pour T(n) sans utiliser T.

- Par exemple, pour la spécification :

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

T(1)=1

→ une forme fermée finale obtenue est T(n)=n.

- Pour arriver à cette résolution :

Il faut (au moins) une condition initiale pour trouver une telle solution.

Remarque : si absence de condition initiale, on aura une **famille** de formes closes.

→ Peu exploitable, voire non-exploitable.

→ Avec une condition initiale, une seule de ces formes satisfait la récurrence.

Pour l'exemple factoriel, selon la condition initiale :

→ T(n)=T(n-1)+1 avec T(1)=0 donne T(n)=n-1

→ T(n)=T(n-1)+1 avec T(1)=1 donne T(n)=n

→ T(n)=T(n-1)+1 avec T(1)=2 donne T(n)=n+1

Résultats obtenus par **substitution** ; voir pages suivantes.

XXII- Calcul de la complexité

Il existe plusieurs méthodes et techniques pour "résoudre" une récurrence.

I) **Pour les cas simples** :

- A partir de la définition même de la fonction (voir TD)
- En posant l'équation de récurrence T(N), procéder à une :
  - *Substitution ascendante*
  - *Substitution descendante*
  - Observation

  Puis faire une hypothèse et la vérifier.

  --> Vérifier avec une preuve simple pour les schémas non complexes

II) **Pour les cas moins simples, utiliser** :

- Équation caractéristique
- Méthodes ad-hoc (Boîte à outils)
- Théorème ou **Méthode Principale** (ou MM : *master method*).

XXII.1- Exemples de calcul de complexité

XXII.1.1- Calcul à partir des ordres de croissances (0,Ω,Θ)

Vérifier les complexités suivantes :

- $f(n) = 3n^2 - 100n + 6 = O(n^2)$

  car pour $c = 3$, $3n^2 > f(n)$

- $f(n) = 3n^2 - 100n + 6 = O(n^3)$

  car pour $c = 1$, $n^3 > f(n)$ quand $n > 3$;

- $f(n) = 3n^2 - 100n + 6 \neq O(n)$

  car pour tout $c > 0$, $cn < f(n)$ quand $n > (c + 100)/3$,

  puisque $n > (c + 100)/3 \Rightarrow 3n > c + 100 \Rightarrow 3n^2 > cn + 100n > cn + 100n - 6$

  $\Rightarrow 3n^2 - 100n + 6 = f(n) > cn$;

- $f(n) = 3n^2 - 100n + 6 = \Omega(n^2)$,

  car pour $c = 2$, $2n^2 < f(n)$ quand $n > 100$;

- $f(n) = 3n^2 - 100n + 6 \neq \Omega(n^3)$,

  car pour tout $c > 0$, $f(n) < c \cdot n^3$ quand $n > 3/c + 3$;

- $f(n) = 3n^2 - 100n + 6 = \Omega(n)$,

  car pour tout $c > 0$, $f(n) < 3n^2 + 6n^2 = 9n^2$ qui est $< cn3$ pour $n > \max(9/c, 1)$;

- $f(n) = 3n^2 - 100n + 6 = \Theta(n^2)$,

  car les deux O and Ω s'appliquent;

- $f(n) = 3n^2 - 100n + 6 \neq \Theta(n^3)$,

  car seulement O s'applique;

- $f(n) = 3n^2 - 100n + 6 \neq \Theta(n)$,

  car seulement Ω s'applique.

- $2^{n+1} = O(2^n)$? --> Oui

  On sait que $f(n) = O(g(n))$ ssi il existe les constantes a, c telles que pour n tendant vers l'infini, $f(n) \leq c \cdot g(n)$.

--> Ce qui est le cas ici car $2^{n+1} = 2 \cdot 2^n$ et $2 \cdot 2^n \leq c \cdot 2^n$ pour tout $c \geq 2$.

- $2^{n+1} = \Omega(2^n)$? --> Oui

Selon la définition, $f(n) = \Omega(g(n))$ ssi il existe la constante $c > 0$ telle que pour n tendant vers l'infini, $f(n) \geq c \cdot g(n)$.

Ici, cette condition est satisfaite pour tout $0 < c \leq 2$.

--> Ayant les conditions Big-Oh et Ω, on a $2^{n+1} = \Theta(2^n)$.

- $(x + y)^2 = O(x^2 + y^2)$? --> oui

  Il nous faut trouver la constante c telle que $(x + y)^2 \leq c(x^2 + y^2)$ pour x,y à grande valeur.

Si on développe : $(x + y)^2 = x^2 + y^2 + 2xy$.

Sans le terme $2xy$, on aurait satisfaction pour $c > 1$.

--> On a donc besoin de lier $2xy$ à $x^2 + y^2$.

Pour $x \leq y$, $2xy \leq 2y^2 \leq 2(x^2 + y^2)$.

Pour $x \geq y$, $2xy \leq 2x^2 \leq 2(x^2 + y^2)$.

--> Dans les 2 cas, on a $2xy \leq 2(x^2 + y^2)$.

Ce qui veut dire que $(x + y)^2 \leq 3(x^2 + y^2)$.

Autres :

$O(f(n)) \cdot O(g(n)) \rightarrow O(f(n) \cdot g(n))$

$\Omega(f(n)) \cdot \Omega(g(n)) \rightarrow \Omega(f(n) \cdot g(n))$

$\Theta(f(n)) \cdot \Theta(g(n)) \rightarrow \Theta(f(n) \cdot g(n))$

XXIII- Cas simples : proposer et vérifier

Pour calculer la complexité dans les cas simples à partir l'équation de réc. T(N),

on peut faire une substitution **ascendante** ou **descendante**.

- Substitution **ascendante** :

Exemple factorielle où T(n)=T(n-1)+1 (avec T(1)=1)

→ T(1)=1 supposons cette valeur arbitraire !
→ T(2) = T(1) + 1 = 1+1 = 2
....
→ T(n-1) = … n-1
→ T(n) = T(n-1)+1 ….= (n-1)+1 = n --> Semble être O(N)

La forme fermée pour la Factorielle sera T(n)=n.

Exemple chiffré :

pour N=5 et la condition initiale T(1)=1

→ T(1)=1

→ T(2) = T(1) + 1 = 1+1 = 2

....

→ T(5) = T(4)+1 = 4+1=5

--> Cela suffit comme vérification

--> On peut au besoin faire une preuve par Induction

- Substitution **descendante** :

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

$= [T(n-2)+1]+1$

$= [[T(n-3)+1]+1]+1$

...

$= [...[[T(n-(n-1))+1]+1]....+1]+1$ avec (n-1) fois '1'

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

→ La forme fermée pour la Factorielle sera $T(n)=n$.

Exemple chiffré : pour N=5 et condition initiale $T(1)=1$

→ $T(5) = T(4)+1 = (T(3)+1)+1 = ... = (T(1)+1)+1...+1=1+1+1+1+1=5$ → $O(N)$

- La vérification des résultats pour ces cas simples peut être triviale.

Rappel : ces techniques basiques sont utilisées pour les cas simples.

Mais elles ne donnent pas de résultat pour tous les cas moins simples.

Exemple : pour

Fib(n) = Fib(n-1)+Fib(n-2) et

Fib(1..2) = 1.

Les recettes pour les cas simples ne fonctionnent pas.

- **Mais dans tous les cas, il faudra apporter la preuve de nos calculs.**

  --> La preuve de la solution proposée nous dira si la solution est juste.

XXIV- Preuve

On peut systématiser une preuve de l'hypothèse (une vérif.) de la complexité.

Comme toute preuve, celle de la complexité calculée peut être faite par différentes méthodes :

- *intuitivement (cas simples),* par l'absurde, directe, indirecte, ...
- "réécriture" (*algébriquement),*
- *induction mathématique*, etc...

XXIV.1- Exemple de preuve : Hanoi

```
Action hanoi(entier n, Dep, Aux, Arr ) :
        Si ( n =< 0 ) rien
        Sinon
          hanoi( n-1, Dep, Arr, Aux )
          déplacer le disque sur Dep vers Arr
          hanoi( n-1, Aux, Dep, Arr )
```

- $T(n)=2.T(n-1)+1 = 2.(2.T(n-2)+1)+1 = 2.(2.(2.T(n-3)+1)+1)+1 = ...$

  $= 2^k T(n-k) + 2^{k-1} + 2^{k-2} +... + 2^0$ substitutions ascendantes

- Hypothèse : on a la forme close $T(n)=2^n-1$ avec T(1)=1 (et t(0)=0) : ok ?

Vérification : $T(n)=2.T(n-1)+1$ posons $S(n)=T(n)+1$

$S(n)=T(n)+1= [2.T(n-1)+1]+1 = 2[T(n-1)+1]$ et donc $S(n)=2.S(n-1)$, n>0

On voit clairement que $S(n) = 2^n$ d'où $T(n)=2^n-1$.

XXIV.2- Exemple de preuve : recherche Dichotomique

Renvoie -1 si X ∉ T, sinon renvoie l'indice I tel que T[I]=X

```
fonction Recherche_binaire(indice Inf, indice Sup, val X, tableau T) =
  Si (Inf > Sup) Alors -1            // par convention, '-1' vaut X ∉ T
    milieu = (Inf+Sup)/2
    Si (X=T[milieu]) Alors milieu
    Sinon  Si (X < T[milieu]) Alors    Recherche_binaire(Inf, milieu-1, X, T)
           Sinon                       Recherche_binaire(milieu+1, Sup, X, T)
```

- On a (hyp. $n=2^k$) $T(1)=1$, $T(2)=T(1)+1 = 2$,
  $T(4)=T(2)+1=3$, ... , $T(8) = T(4)+1=4$, ...
  $T(n) = T(n/2) +1$
- On "devine" $T(n) = lg(n) + 1$ pour $n=2^k$

Est-ce vrai ?

- **Vérification** intuitive : si l'hypothèse lg(n) est fondée, on aura

  $T(n) = T(n/2)+1 = [lg(n/2)+1]+1=[lg(n)-lg(2)+1]+1=lg(n)+1$ ← Yes !

XXIV.3- Preuve par Induction

XXIV.3.1- Introduction à l'induction

XXIV.3.1.a- Exemple 1

Montrer que $\sum_{i=1}^{n} i = n(n+1)/2 \quad \forall n > 0$

Cas de base : n=1 --> OK

Hypothèse : $\sum_{i=1}^{n} i = n(n+1)/2 \; \forall n > 0$

Induction : à partir du cas de base et l'hypothèse, peut on prouver que :

$$\sum_{i=1}^{n+1} i = (n+1)(n+2)/2 \quad \forall n > 0 \; ?$$

Preuve : $\sum_{i=1}^{n+1} i = (n+1) + \sum_{i=1}^{n} i$ = (n+1)+[n.(n+1)]/2

= $[2(n+1) + n(n+1)]/2$ = (n+1)(n+2)/2 factorisation par (n+1) CQFD

XXIV.3.1.b- Exemple 2

Montrez que $\sum_{i=1}^{n} i.\, i! \;=\; (n+1)! \;-\; 1, \;\; \forall\, n > 0$

Cas de base : on a pour n=1 : $\sum_{i=1}^{1} i.\, i! \;=\; (1+1)! \;-\; 1 \;=\; 1$

Hypothèse : cas général $\sum_{i=1}^{n} i.\, i! \;=\; (n+1)! \;-\; 1 \;\; \forall\, n > 0$

Induction : peut on prouver que $\sum_{i=1}^{n+1} i.\, i! \;=\; (n+2)! \;-\; 1, \;\; \forall\, n > 0$

On a : $\sum_{i=1}^{n+1} i.\, i! \;=\; (n+1).\,(n+1)! \;.\; \sum_{i=1}^{n} i.\, i!$

Donc $\sum_{i=1}^{n+1} i.\, i! \;=\; (n+1).\,(n+1)! \;.\,(n+1)! \;-\; 1$

$= (n+1)! \;.\,((n+1) \;+1) \;-\; 1$

$= (n+1)! \;.\,(n+2) \;-\; 1$

$= (n+2)! \;-\; 1$ CQFD

XXIV.4- Utilisation dans la complexité

XXIV.4.1- Exemple 1

- A partir de l'équation de récurrence
- Exemple (on notera $T(n)$ par t_n):

```
def fact(n) :
  if (n == 0) : return 1
  return n * fact(n - 1)
```

Si t_n = nombre de multiplications nécessaires pour un *n* donné (= nbr d'appels),

Alors on a $t_n = t_{n-1} + 1$

t_{n-1} est la valeur de *t* dans les appels récursifs et 1 le coût de la multiplication

⇒ $t_n = t_{n-1} + 1$ est **l'équation de récurrence recherchée.**

La **condition initiale** (nécessaire à la résolution) est ici $t_0 = 0$

Suite : Factorielle

```
def fact(n) :
  if (n == 0) : return 1
  return n * fact(n - 1)
```

On décide $t_0=1$ mais $t_0=0$ ne modifiera pas la complexité (linéaire) de "fact".

→ cette constante sera négligeable devant la limite de *n*.

On peut alors calculer t_n pour différentes valeurs de n (par une subs. desc.) :

$t_1 = t_{1-1} + 1 = t_0 + 1 = 1,$
$t_2 = t_{2-1} + 1 = t_1 + 1 = 2,$
.....

On constate (hypothèse) :

$t_n = ... = n$ appelée **solution** de l'équation de récurrence

☞ Maintenant, prouvons que $t_n=n$ est bien une solution (?)

Vérification par Induction :

Base de l'induction : $t_0=0$

Hypothèse d'induction : supposons avoir $t_n=n$

Étape d'induction : démontrer, par induction que : $t_{n+1} = n+1$

Facile ! : il existe plusieurs autres "preuves" triviales

selon l'équation $t_n = t_{n-1} +1 \rightarrow T_{(n+1)} = T_{(n+1)-1} +1 = T_n +1$ CQFD !

Rappel : une induction ne peut pas trouver une solution mais permet de vérifier si une solution proposée en est une.

Notons cependant que **l'induction constructive** peut aider à trouver une solution.

XXIV.4.2- Exemple 2

$t_n = t_{n/2} + 1$ pour $n > 1$, $n=2^k$ (une puissance de 2)
$t_1 = 1$

Quelques valeurs de t_n :

$t_1 = 1$ $t_2 = t_{2/2} + 1 = t_1 + 1 = 2$ $t_4 = t_{4/2} + 1 = t_2 + 1 = 3$

$t_8 = 4$ $t_{16} = 5$ ...

On propose une solution (constatée, flairée !) : $t_n = \log n + 1$.

Vérifions :

Base : $t_1 = 1$

Hypothèse : $t_n = \log n + 1$

Étape d'induction : $t_{2n} = \log (2n) + 1$? si $n = 2^k$: la prochaine val. pour n est $2n$

--> De l'algorithme lui-même, on a : $t_{2n} = t_{(2n)/2} + 1 = t_n + 1$

--> et de l'hypothèse, on a : $t_n = \log (n) + 1$

D'où $t_{2n} = t_{n+1} = \log (n) + 1 + 1 = \log (n) + \log(2) + 1 = \log (2n) + 1$ CQFD.

XXIV.4.3- Exemple 3 (à démontrer)

Soit la récurrence :

$t_n = 7t_{n/2}$ pour n > 1, *n* une puissance de 2
$t_1 = 1$

Quelques valeurs de t_n :

$t_1 = 1$ $\quad t_2 = 7t_{2/2} = 7t_1 = 7$ $\quad t_4 = 7t_{4/2} = 7t_2 = 7^2$

$t_8 = 7^3$ $\quad t_{16} = 7^4$ ...

On constate une solution :

$t_n = 7^{\log n}$ à démontrer par l'induction Math.

Remarque : on sait $7^{\log n} = n^{\log 7}$

--> $t_n = n^{\log 7} \simeq n^{2.81}$

XXIV.5- Vers des outils plus puissants

Les outils élémentaires utilisés jusqu'ici ne suffisent pas parfois.

XXIV.5.1- Exemple 1 (cas défavorable)

$t_n = 2t_{n/2} + n - 1$ pour n > 1, *n* une puissance de 2
$t_1 = 0$

On a quelques valeurs de t_n :

$t_1 = 0$ $t_2 = 2t_{2/2} + 2 - 1 = 2t_1 + 1 = 1$ $t_4 = 2t_{4/2} + n - 1 = 2 + 4 - 1 = 5$
$t_8 = 17$ $t_{16} = 49$ ...

Il n'y a pas de solution candidate comme dans les exemples précédents.

--> L'induction ne nous servira pas **dans ce cas** puisque l'induction sert à vérifier si une solution candidate est juste.

NB : la complexité de cet exemple sera $t_n = n.\log(n) - (n-1) = n.\log(n)-n+1$ (voir + loin)

XXIV.5.2- Exemple 2 (cas défavorable) : Fib

Rappel Fib : $t_n = t_{n-1} + t_{n-2}$

$t_0 = 0$

$t_1 = 1$

On peut écrire la première équation par :

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

→ On est vite bloqué dans l'approche par équation de récurrence !

→ Au mieux, une forme exponentielle se dégage (si on fait des substitutions).

→ On peut faire mieux,

La suite Fibonacci est définie par une **équation linéaire homogène**.

Comment résoudre ce type d'équation ?

→ On va le voir après quelques exemples introductifs simples.

XXV- Résolution de l'équation caractéristique

Supposons l'équation suivante (l'exemple Fib est traité à la suite) :

$t_n - 5t_{n-1} + 6t_{n-2} = 0$ pour $n>1$

$t_0 = 0$

$t_1 = 1$

Si l'on note $t_n = r^n$

On aura alors $\mathbf{t_n - 5t_{n-1} + 6t_{n-2} = r^n - 5r^{n-1} + 6r^{n-2}}$

Dans ce cas, $\mathbf{t_n = r^n}$ est une solution à cette récurrence si **r** est la racine de l'équation

$$\mathbf{r^n - 5r^{n-1} + 6r^{n-2} = 0}$$

Résolution :

On a $r^n - 5r^{n-1} + 6r^{n-2} = r^{n-2}(r^2 - 5r + 6)$

les racines : **r=0** et celles de $r^2 - 5r + 6 = 0$

qui sont obtenues par $r^2 - 5r + 6 = (r-3)(r-2) = 0$

Les racines de l'équation : **r=0, r=3** et **r=2**

C'est à dire :

$t_n=0$, $t_n = 3^n$ et $t_n = 2^n$ sont tous trois solutions à l'équation Réc.

On note :

Si 0, 3^n et 2^n sont des solutions,

Alors toute solution de la forme générale (= une combinaison linéaire des t_n) :

$$t_n = c_1\ 3^n + c_2\ 2^n$$ (c_1 et c_2 sont des constantes arbitraires)

Est aussi une solution (voir le théorème-1 suivant).

N.B. : On peut démontrer que ce sont les seules solutions.

- On a une infinité de solutions (suivant c_1 et c_2) mais laquelle choisir ?

--> Ce choix est déterminé par les conditions initiales :

$t_0=0$ $t_1=1$. D'où

$$t_0 = c_1\ 3^0 + c_2\ 2^0 = 0$$

$$t_1 = c_1\ 3^1 + c_2\ 2^1 = 1$$

L'ensemble se simplifie par : $c_1 + c_2 = 0$

$3\ c_1 + 2\ c_2 = 1$

On obtient : $c_1 = 1$ et $c_2 = -1$

La solution à l'équation de récurrence sera :

$t_n = 1\ (3^n) - 1\ (2^n) = 3^n - 2^n$

N.B. : avec les conditions initiales $t_0 = 1$ et $t_1 = 2$, on aurait la solution $t_n = 2^n$

Cela veut dire que la récurrence représente une classe de fonctions.

NB : l'équation ($r^2 - 5r + 6 = 0$) est appelée **l'équation caractéristique** de la récurrence.

Cette équation est définie comme suit

XXV.1- Théorème 1

Théorème 1 (preuve dans le support "long" du cours):

- L'équation caractéristique de l'équation de récurrence linéaire et homogène à coefficients constants $a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = 0$ est définie par : $a_0 r^k + a_1 r^{k-1} + \ldots + a_k r^0 = 0$ avec k racines distinctes $r_1, r_2 \ldots r_k$
- L'unique solution à la récurrence sera alors (c_i constantes arbitraires) :

$$t_n = c_1 r_1^n + c_2 r_2^n + \ldots + c_k r_k^n$$

Par exemple :

L'équation caractéristique pour $\mathbf{5t_n - 7t_{n-1} + 6t_{n-2} = 0}$

est $\mathbf{5r^2 - 7r + 6 = 0}$ avec un ordre k=2.

XXV.1.1- Exemple 1

Résoudre l'équation de récurrence :

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

$t_0 = 0$

$t_1 = 1$

L'équation caractéristique $\mathbf{r^2 - 3r - 4 = 0 = (r - 4)(r + 1)}$

D'où $r = 4$ et $r = -1$

→ N.B. : la racine r=0 n'apporte aucune information.

La solution générale : $\mathbf{t_n = c_1\ 4^n + c_2\ (-1)^n}$

Trouver les constantes c1 et c2 en utilisant les conditions initiales :

--> D'où $c_1 = 1/5$ et $c_2 = -1/5$

On obtient la solution finale $\mathbf{t_n = 1/5\ 4^n - 1/5\ (-1)^n}$

XXV.1.2- Exemple 2 : une autre définition de Fib

Nous avons vu plus haut une autre définition récursive pour Fib :

$fib(2n) = fib(n)^2 + 2fib(n).fib(n-1)$ *pour tout entier n pair*

$fib(2n+1) = fib(n)^2 + fib(n+1)^2$ *pour tout entier n impair*

Pour Simplifier, on considère le premier cas. Il est représentatif.

Le 2e cas a un terme supplémentaire.

$T_{2n} = (T_n)^2 + 2T_n.T_{n-1}$

→ Hyp : le coût de $(T_n)^2$ est simplement le coût de $T_n + X$ (X disparaîtra !)

$T_{2n} = T_n + X + 2T_n.T_{n-1}$

D'où → $r^{2n} = r^n + X + 2r^n r^{n-1}$

→ $r^n \cdot r^n = r^n + X + 2r^n\, r^{n-1}$

→ $r^n \cdot r^n - r^n - X - 2r^n\, r^{n-1} = 0$

→ $r^n (r^n - X/r^n - 2\, r^{n-1}) = 0$

→ $r = 0$ est une racine et on a :

$r^n - X/r^n - 2\, r^{n-1} = 0$:

→ $r^{n-1} (r - X/r^{2n-1} - 2) = 0$

→ r = 0(on le sait déjà), la quantité X/r^{2n-1} est négligeable

→ r - 2 = 0

→ r = 2

Pour CETTE VERSION de Fib, on a une complexité $O(2^N)$.

Comparer r=2 à 3/2 < r < 5/3 de ci-dessus.

XXVI- Cas de racines multiples : théorème-2

- Le théorème-1 ci-dessus indique que les **k** racines de l'équation caractéristique sont **distinctes**.

Comment utiliser ce théorème dans le cas d'une équation caractéristique

de la forme $(r - 1)(r - 2)^3 = 0$

C-à-d. le terme (r-2) est à la puissance 3 ?

- Dans ce cas, la racine r=2 est appelée **la racine de multiplicité 3** de l'équation.

- Le théorème suivant permet alors à une racine d'avoir une multiplicité.

XXVI.1- Théorème 2

Si r est une racine de multiplicité m de l'équation caractéristique.

Alors :

$t_n=r^n$, $t_n=n\,r^n$, $t_n=n^2\,r^n$, $t_n=n^3\,r^n$, ..., $t_n=n^{m-1}\,r^n$

sont toutes des solutions à la récurrence.

C'est-à-dire, pour chacune de ces solutions, un terme est inclus dans la solution générale de la récurrence.

XXVI.1.1- Exemple 1

Soit la récurrence :

$t_n - 7\, t_{n-1} + 15\, t_{n-2} - 9\, t_{n-3} = 0$ pour $n>2$

$t_0 = 0$

$t_1 = 1$

$t_2 = 2$

- L'équation caractéristique $\mathbf{r^3 - 7\, r^2 + 15r - 9 = 0}$

→ $r^3 - 7\, r^2 + 15r - 9$

= $(r-1)(r-3)^2 = 0$ où la racine 3 est de multiplicité 2.

→ La solution générale sera alors :

$$\mathbf{t_n = c_1\ 1^n + c_2\ 3^n + c_3\ n\ 3^n}$$

On introduit les termes $\mathbf{3^n}$ et $\mathbf{n\ 3^n}$ car la racine 3 est de multiplicité 2.

Les conditions initiales (à l'aide de t_0, t_1 et t_2) donnent

$$c_1 = -1, \qquad c_2 = 1 \qquad \text{et} \qquad c_3 = -1/3$$

d'où :

$$\mathbf{t_n = (-1)\ 1^n + (1)\ 3^n + (-1/3)\ n\ 3^n}$$

$$\mathbf{= -1 + 3n - n3^{n-1}}$$

XXVI.1.2- Exemple 2

Soit la récurrence :

$t_n - 5\, t_{n-1} + 7\, t_{n-2} - 3\, t_{n-3} = 0$ pour $n>2$

$t_0 = 0$

$t_1 = 2$

$t_2 = 3$

On obtient l'équation caractéristique $r^3 - 5\, r^2 + 7r - 3 = (r-3)(r-1)^2 = 0$

→ La racine 1 est de multiplicité 2.

→ La solution générale sera alors: $\mathbf{t_n = c_1\, 3^n + c_2\, 1^n + c_3\, n\, 1^n}$

→ Conditions initiales : $c_1 = 0$, $c_2 = 1$ et $c_3 = 1$

→ $\mathbf{t_n = 0\, 3^n + 1\, 1^n + 1\, n\, 1^n}$

$\mathbf{= 1 + n}$

XXVII- Récurrence linéaire non homogène

Le cas où le terme à droite de l'équation n'est pas = 0 mais *f(n)*.

Une récurrence de le forme $a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = f(n) \neq 0$

où ai sont des constantes et f(n) une fonction non nulle appelée une

équation de récurrence linéaire non Homogène à coefficient constant.

- Il n'y a pas de méthode générale connue pour résoudre cette équation.

  Par contre, on propose une solution pour une forme particulière où :

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

Deux exemples :

$t_n - 3t_{n-1} = 4^n$

$t_n - 3t_{n-1} = 4^n (8n+7)$

XXVII.1- Théorème 3

Une équation de récurrence linéaire non Homogène à coefficient constant de la forme

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

peut être transformée en $(a_0 r^k + a_1 r^{k-1} + \ldots + a_k)(r-b)^{d+1} = 0$

où *d* = le degré de *p(n) et P(n) est un polynôme en* $\mathbb{N} \rightarrow \mathbb{R}$

→ N.B. : noter l'origine de k (en rouge), **b** sera une nouvelle racine.

Résolution : cette équation est alors composée de 2 parties :

1- l'équation caractéristique pour la partie homogène (premier terme à gauche)

2- un terme obtenu de la partie non homogène de la récurrence.

⇨ S'il y a plus d'un terme de la forme $\mathbf{b^n\ p(n)}$ à droite, chacun contribuera à un terme de l'équation caractéristique.

XXVII.1.1- Exemple 1

Soit

$$t_n - 3\,t_{n-1} = 4^n\,(2n+1) \qquad \text{pour } n>1$$

$$t_0 = 0$$

$$t_1 = 12$$

Démarche :

1- On obtient l'équation caractéristique pour la **partie homogène**

$$t_n - 3\,t_{n-1} = 0 \qquad \rightarrow \; r^1 - 3 = 0$$

2- On obtient de la partie **non homogène** de la forme $b^n\,P(n)$:

$$4^n\,(2n^1+1) \qquad \text{où} \qquad b = 4 \text{ et } d = 1 \text{ (degré de p(n))}$$

Le terme de la partie non homogène sera : $(r - b)^{d+1} = (r - 4)^{1+1}$

3- On applique le théorème 3 pour obtenir une équation caractéristique :

L'équation caractéristique sera $(r - 3)(r - 4)^2$

4- La résolution de l'équation $(r - 3)(r - 4)^2 = 0$

donne les racines r=3 et r=4 r=4 est de multiplicité 2

5- Par le théorème 2 : $t_n = c_1 3^n + c_2 4^n + c_3 n4^n$.

6- Calcul des constantes : ici, on n'a que deux cas basiques (t_0 et t_1)

→ Mais on peut calculer t_2 par $t_2 - 3 t_1 = 4^2 (2 * 2 + 1)$

Sachant $t_1 = 12$, on obtient $t_2 = 116$.

→ La forme générale se la solution sera : $t_n = 20(3^n) - 20(4^n) + 8 n 4^n$.

XXVII.1.2- Exemple 2

Soit la récurrence

$$t_n = 3\ t_{n-1} + 2^n$$

$$t_0 = 1$$

Rappel :

Une équation de récurrence linéaire non Homogène à coefficient constant de la forme

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

peut être transformée en $(a_0 r^k + a_1 r^{k-1} + \ldots + a_k)\ (r-b)^{d+1} = 0$

où *d* = le degré de *p(n) et P(n) est un polynôme en* $\mathbb{N} \rightarrow \mathbb{R}$

→ la racine de la partie homogène est r=3

→ la partie droite est de la forme $(r-2)^{0+1}$ et p(n)=1 d'où le degré d = 0.

La complexité sera de la forme $T(n)= c1\ 3^n + c2\ 2^n$

Et les calculs donneront c1=3 et c2 = -2

d'où $T(n)= 3.\ 3^n - 2\ .\ 2^n.$

Remarque :

Si l'équation de récurrence (non homogène) est de la forme

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = \text{Cste (une constante)}$$

Alors **Cste** représente **P(n)** et non b^n

Car si P(n) est constante, alors n=0.

Dans le cas contraire, on introduirait une racine supplémentaire r=b

→ incohérent !

Pour un exemple de ce cas : voir les tours de Hanoï.

XXVII.1.3- Exercice 1

Soit la somme $f(n) = \sum_{j=0}^{n} j^3 = 1^3 + 2^3 + 3^3 + \ldots + n^3$

dont l'équation de récurrence est donnée par :

$t_0=0,\ t_1=1,\ t_2=9,\ t_3=36,\quad t_4=100$

$t_n = t_{n-1} + n^3$

On est dans le cas d'une équation de récurrence non homogène car $t_n - t_{n-1} = n^3$

(I) Calculer les détails cette complexité.

Indication :

le résultat devrait être $t_n = c_1 + c_2\, n + c_3\, n^2 + c_4\, n^3 + c_5\, n^4$

(II) Calculer les coefficients et en déduire une relation entre les fonctions *f(n)* des deux exemples précédents.

XXVII.1.4- Exercice 2

Calculer la complexité de

$$t_n = 7\ t_{n-1} - 10\ t_{n-2} + (2n+5)\ 3^n$$

et déduire qu'elle est $\Theta(5^n)$.

Indication :

dans la partie non homogène, on a $b^n = 3^n$ et $p(n)=2n+5$

d'où la partie homogène sera multipliée par $(r-3)^{1+1}$.

XXVIII- Boite à outils (Cook Book)

- La généralisation de certains de ces calculs a donné lieu à une boite à outils.

  Il s'agit d'une méthode dite "principale" (*main method*) qui permet de calculer (rapidement) une complexité Θ (Théta).

  --> Le calcul de O(.) est également possible.

- Elles peuvent remplacer le calcul dans le cas d'équation non-homogènes.

XXVIII.1- Méthode principale (MM) et la famille Thêta

- Cook-Book des cas **fréquents** calculés par différentes techniques.
- Utilisé (souvent) pour trouver la complexité Θ (mais aussi O et Ω)
- Permet d'éviter des calculs détaillés et donne directement la solution.
- Le Cook-Book s'applique bien aux algorithmes de type "diviser pour régner"

  Rappel : découpent en sous-problèmes, résolvent ces sous-roblèmes de taille n/b puis combinent ces réponses en un temps $O(n^d)$.
- Pour a,b,d > 0, leur temps d'exécution T(n) peut s'écrire :

$$T(n) = aT(n/b) + O(n^d)$$

  --> "n/b" : $\lceil n/2 \rceil$ et $\lfloor n/2 \rfloor$ (par excès /défaut pour équilibrer le découpage).
- Le "Théorème Maître" (MM) permet de déterminer la complexité de la plupart des algorithmes de type "diviser pour régner".

- Le but (de MM) : résoudre la récurrence de la forme générique :
  - $T(n) = a\,T(n/b) + cn^k$ avec $n>1$, $n=b^m$ (n est une puissance de b, $m>=0$)
  - $T(1) = d$ (condition initiale)
  - $b >= 2$, $k >= 0$ sont des constantes entières,
  - $a > 0$, $c > 0$, $d >= 0$ des constantes

Suivant l'algo. étudié, on pioche dans le cook-book et on choisit un des cas :

I	$T(n) = \Theta(n^k)$	si $a < b^k$
II	$T(n) = \Theta(n^k . lg(n))$	si $a = b^k$
III	$T(n) = \Theta(n^{lg\_b\,(a)})$	si $a > b^k$

→ Remarque : fournit une complexité Theta

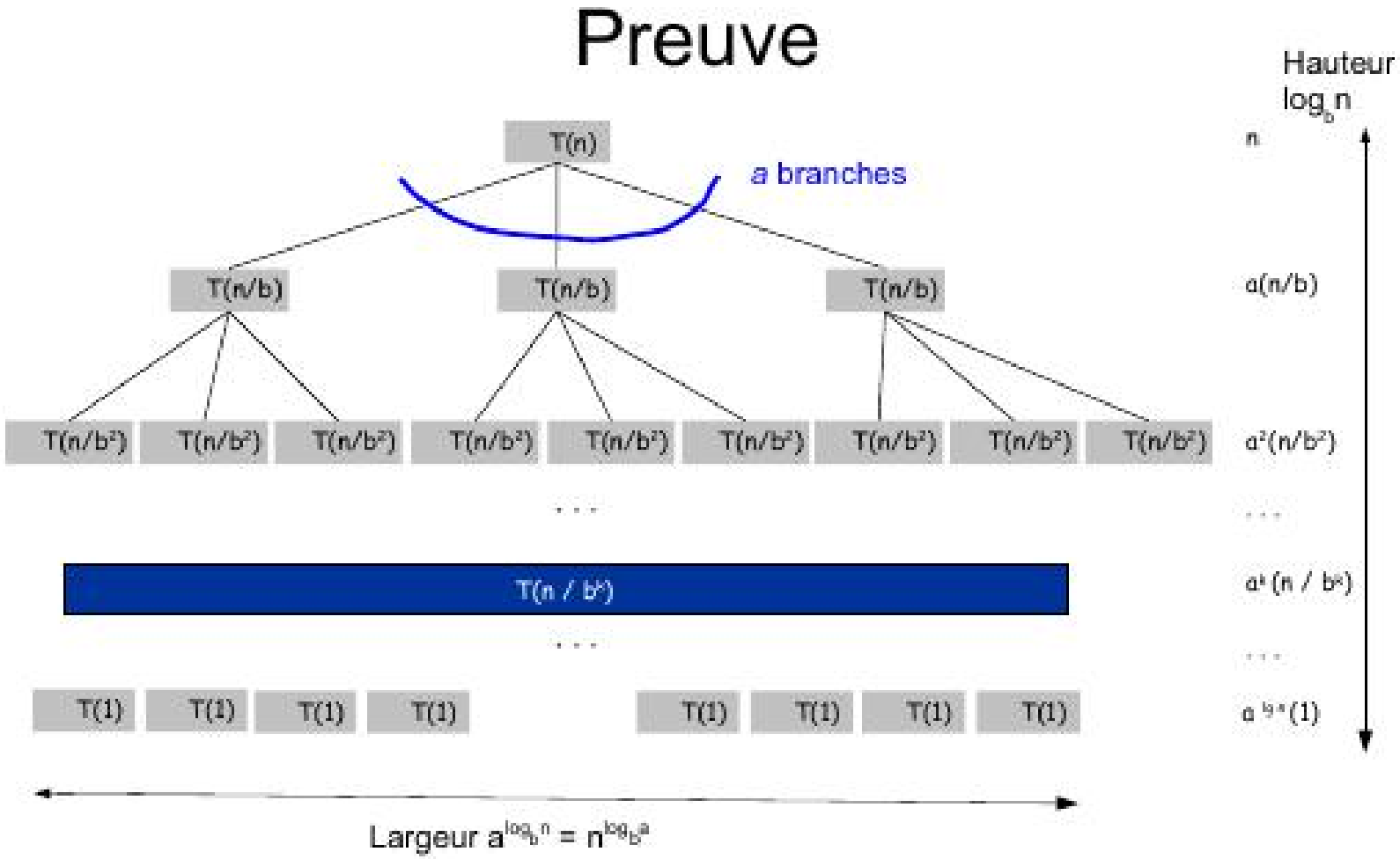
→ Pour *big-Oh* / Ω, remplacer "T(n)= " par "T(n) <=" ou "T(n) ≥"

XXVIII.1.1- Exemples

Il suffit d'identifier simplement chaque cas de figure.

- $T(n) = 16\ T(n/4) + 5n^3$
  - $a=16, b=4, k=3$ et $16 < 4^3$ donc **cas (I)** : $T(n) = \Theta(n^k)$ si $a < b^k$
  - $T(n) = \Theta(n^k) = \Theta(n^3)$
- $T(n) = 2\ T(n/2) + n$
  - $a=2, b=2, k=1$ et $2=2^1$ donc **cas (II)** : $T(n) = \Theta(n^k . lg(n))$ si $a = b^k$
  - $T(n) = \Theta(n^k . lg(n)) = \Theta(n^1 . lg(n))$
- $T(n) = 8\ T(n/2) + n$
  - $a=8, b=2, k=1$ et $8 > 2^1$ donc **cas (III)** : $T(n) = \Theta(n^{lg\_b\ (a)})$ si $a > b^k$
  - $T(n) = \Theta(n^{lg\_b\ (a)}) = \Theta(n^{lg(8)}) = \Theta(n^3)$

XXVIII.2- Illustration de MM



Le niveau k est composé de a^k sous-problèmes, chacun de taille n/b^k

XXVIII.3- Méthode principale généralisée (MMG)

- Pour résoudre la récurrence : si on a les conditions suivantes
  - $T(n) = a\,T(n/b) + f(n)$ avec $n>1$, $n=b^m$ (n est une puissance de b)
  - $T(1) = d$ (cond. initiale)
  - $b >= 2$ une constante entière et $a > 0$, $d >= 0$ des constantes

Suivant l'algorithme que l'on étudie, on choisit un des cas :

I')	$T(n) = \Theta(f(n))$, si $f(n) = \Omega(n^{\lg_b (a+\varepsilon)})$	et $a.f(n/b) <= cf(n)$ pour $c <= 1$ cste et $n \rightarrow \infty$
II')	$T(n) = \Theta(n^{\lg_b (a)}. \lg(n))$, si $f(n) = \Theta(n^{\lg_b (a)})$	
III')	$T(n) = \Theta(n^{\lg_b (a)})$, si $f(n) = O(n^{\lg_b (a-\varepsilon)})$	$\lg_b$: log en base b

Avec $\varepsilon > 0$, $\lg_b$ est important car le paramètre b est présent dans $T(n)$

Rappel :

I') $T(n) = \Theta(f(n))$, si $f(n) = \Omega(n^{\lg_b (a+\varepsilon)})$ et $a.f(n/b) <= cf(n)$ pour $c <= 1$ cste et $n \rightarrow \infty$

II') $T(n) = \Theta(n^{\lg_b (a)}. \lg(n))$, si $f(n) = \Theta(n^{\lg_b (a)})$

III') $T(n) = \Theta(n^{\lg_b (a)})$, si $f(n) = O(n^{\lg_b (a-\varepsilon)})$ lg_b : log en base b

Avec $\varepsilon > 0$, lg_b est important car le paramètre b est présent dans T(n)

- Le cas (I') peut être ré-écrite en $n^{\lg_b (a+\varepsilon)} = O(f(n))$.

  Car si $f(n)=\Omega(g(n)) \rightarrow g(n)=O(f(n))$

- Ce cas (I') est souvent ignoré à cause de sa condition complexe !

XXVIII.3.1- Exemples

A partir de la récurrence T(n), identifier f(n) et extraire la complexité.

- $T(n) = 2\ T(n/2) + n$

  Nous avons : *a = 2, b=2, f(n) = n, lg_b (a) = lg(2)*

  → **cas (II')** : $f(n) = n = n^1 = \Theta(n^{lg\_b\ (a)})$

  II') → $T(n) = \Theta(n^{lg\_b\ (a)}.\ lg(n)) = \Theta(n^{lg(2)}.\ lg(n)) = \Theta(n\ .\ lg(n))$

- $T(n) = 8\ T(n/2) + n$

  On a a = 8, b = 2, f(n) = n, lg_b (a) = lg(8) = 3 et $2=2^1$

  → **cas (III')** : $f(n) = n = O(n^{3-1})$ avec $\varepsilon = 1$

  III') → $T(n) = \Theta(n^{lg\_b\ (a)}) = \Theta(n^{lg\ (8)}) = \Theta(n^3)$

Remarques :

- Les 3 cas de MM (I,II,III) sont des cas particuliers de MMG (I',II', III').
- P. ex. : Si $T(n) = a\,T(n/b) + cn^k$ satisfait la condition de (II) : $a=b^k$

  Alors on peut écrire (en utilisant le cas II') :

  $a=b^k$

  $lg\_b(a) = lg\_b(b^k) = k$

  $\rightarrow f(n) = cn^k = cn^{lg\_b(a)} \in \Theta(n^{lg\_b\,(a)})$

  cf. la condition du cas général (II')

XXVIII.4- Exemples de calcul et Master Theorem

Les algorithmes comparés ci-dessous sont développés ensuite.

Rappels Pour $T(n) = aT(n/b) + O(n^d)$, le terme général est :

I	$T(n) = \Theta(n^k)$	si $k > \log_b a$	(était $\equiv a < b^k$)
II	$T(n) = \Theta(n^k . lg(n))$	si $k = \log_b a$	(était $\equiv a = b^k$)
III	$T(n) = \Theta(n^{lg\_b\ (a)})$	si $k < \log_b a$	(était $\equiv a > b^k$)

- Algorithme **mult** : $T(n) = 4T(n/2) + O(n)$

  $a=4, b=2, k=1$ et donc $1<2=\log_2 4 \rightarrow O(n^{\log 4})=O(n^2)$

- Algorithme **mult2** : $T(n) = 3T(n/2) + O(n)$

  $a=3, b=2, k=1$ et donc $1<1{,}58 \approx \log_2 3 \rightarrow O(n^{\log 4})=O(n^{1,58})$

- Algorithme **Merge-sort** : $T(n) = 2T(n/2) + O(n)$

  $a=2, b=2, k=1$ et donc $1=1=\log_2 2 \rightarrow O(n \log n)$

XXVIII.4.1- Algorithme mult

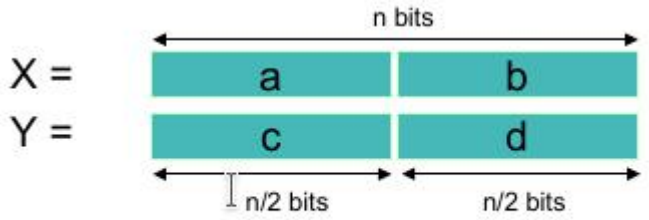
def mult(x,y) : # x et y sur n chiffres / bits

si n = 1 :

renvoyer x * y

partitionner x en a,b et y en c,d

renvoyer mult(a,c)*2^n + (mult(a,d)+ mult(b,c))*$2^{n/2}$ + mult(b,d)



$X \times Y = ac\ 2^n + (ad + bc)\ 2^{n/2} + bd$

Ex. décimal :

12345678 * 21394276

1234*2139 1234*4276 5678*2139 5678*4276

12*21 12*39 34*21 34*39 <-- *On développe seulement le 1er couple.*

1*2 1*1 2*2 2*1

2 1 4 2

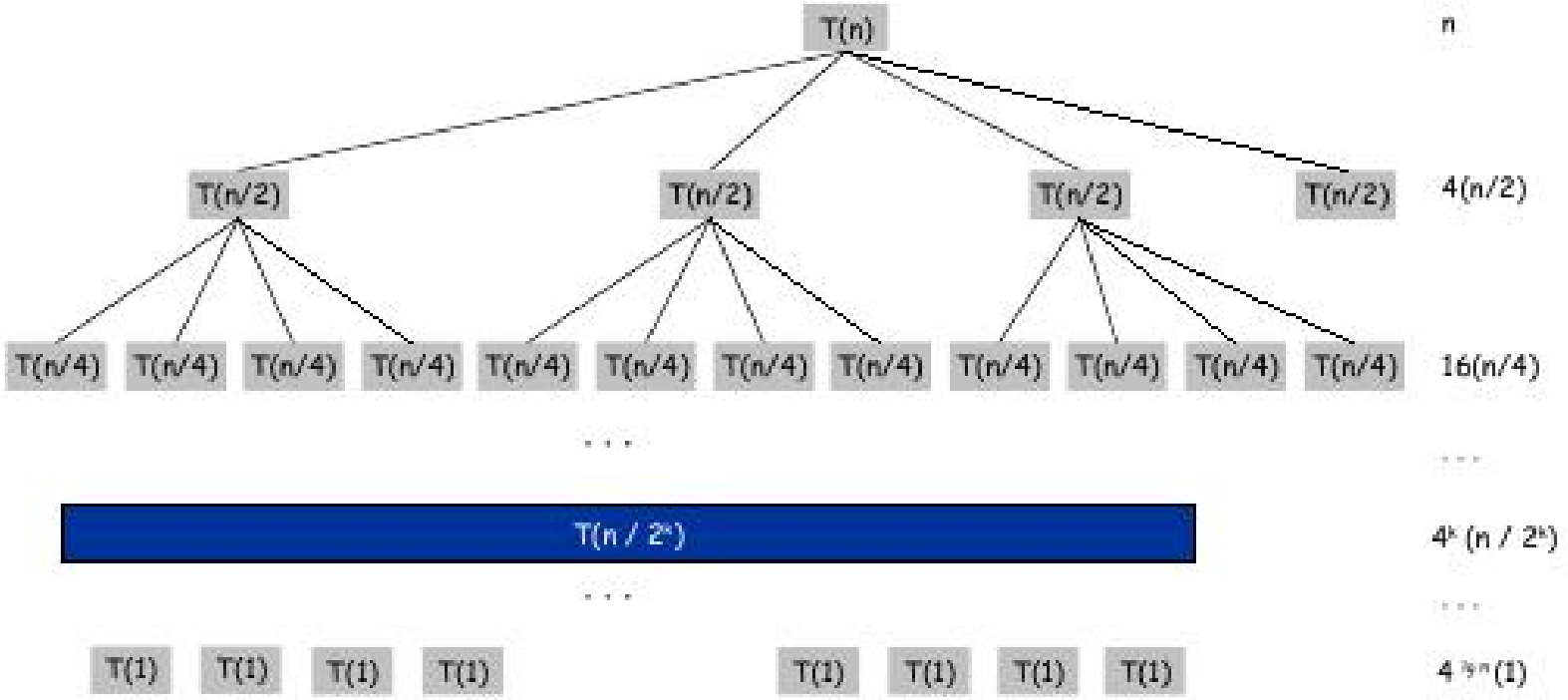
--> 12*21 = 2*102 + (1 + 4)101 + 2 = 252

--> Pour les 4 pairs initiaux, on aura : 252 468 714 1326

(chaque terme levé à la puissance) *10^4 + *10^2 + *10^2 + *1 = 2639526

La complexité de mult :

L'arbre des appels récursifs de mult :



A chaque niveau k de l'arbre, il y a 4^k nœuds.

La complexité de cet algorithme est : $T(n) = 4T(n/2) + O(n)$

XXVIII.4.2- Mult2 : une modification de mult

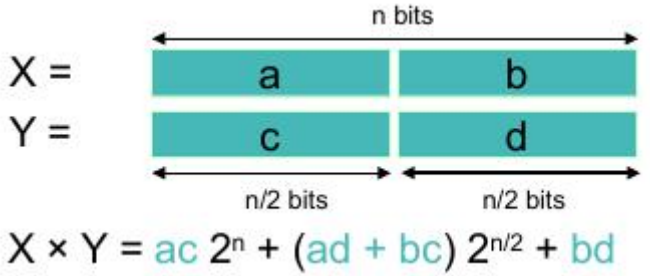
def mult2(x,y): # deux entiers x et y sur n chiffres/bits

si n = 1 : renvoyer x * y

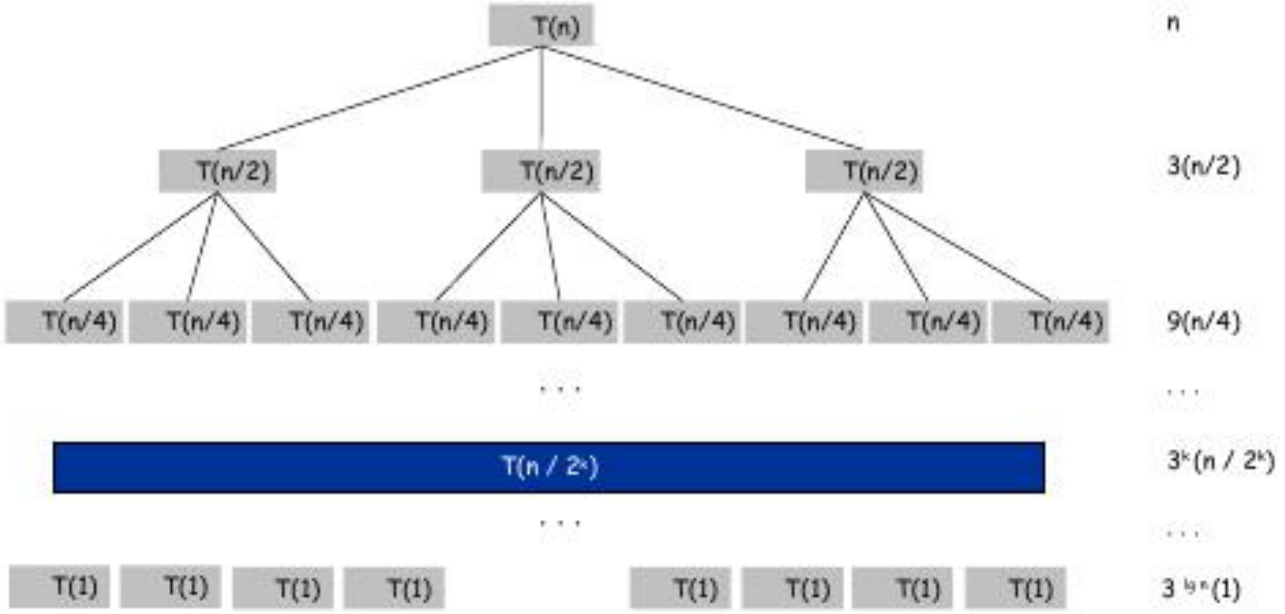
partitionner x en a,b et y en c,d

P3= mult(a+b,c+d); P4= mult(a,c); P5= mult(b,d)

renvoyer $P4*2^n + (P3-P4-P5)*2^{n/2} + P5$



Avec le dernier terme qui est en fait équivalent de celui de mult, on a :

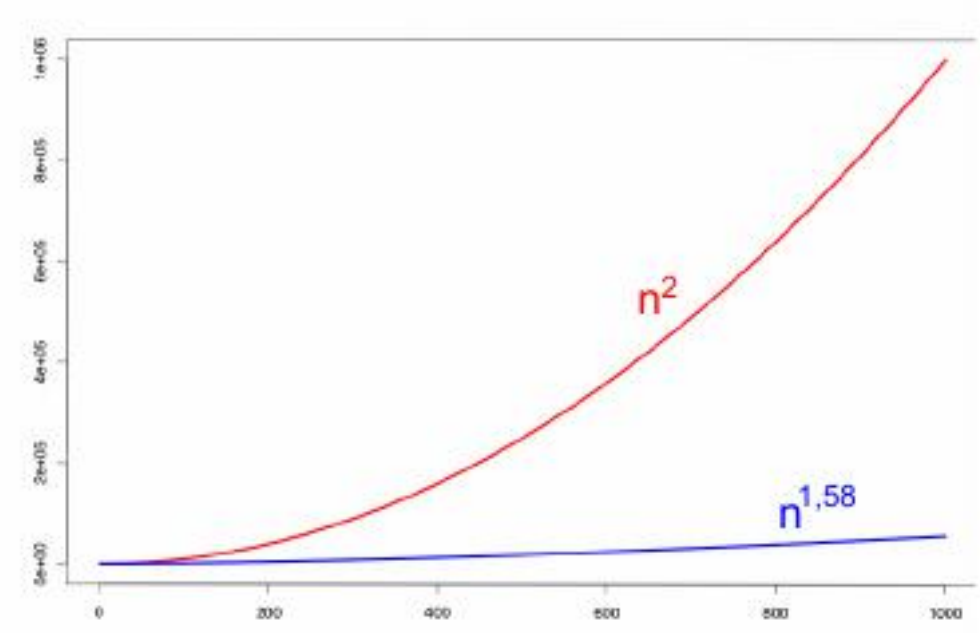


La complexité de **mult2** : à l'aide de cet arbre, on obtient:

$$T(n)=\sum_{k=0}^{\log_2 n} n\left(\frac{3}{2}\right)^k = n\,\frac{\left(\frac{3}{2}\right)^{1+\log_2 n}-1}{\frac{3}{2}-1} = 2n\left(\left(\frac{3}{2}\right)^{\log_2 n}\frac{3}{2}-1\right) = 2n\left(n^{\log_2\frac{3}{2}}\frac{3}{2}-1\right) = 3n^{\log_2 3}-2n$$

--> On obtient une complexité : $\boxed{3n^{\log_2 3} - 2n \in O(n^{1,58})}$ car $\log_2 3 = 1{,}58$

Le gain est appréciable : on compare **mult** et **mult2** : $O(n^{1,58}) << O(n^2)$



Résumé : principe de Diviser pour Régner :

DIVISER le problème en a sous-pbs de taille n/b

RESOUDRE les sous-problèmes récursivement

FUSIONNER les réponses aux n/b sous-pbs en $O(n^k)$ afin d'obtenir la réponse au problème de départ.

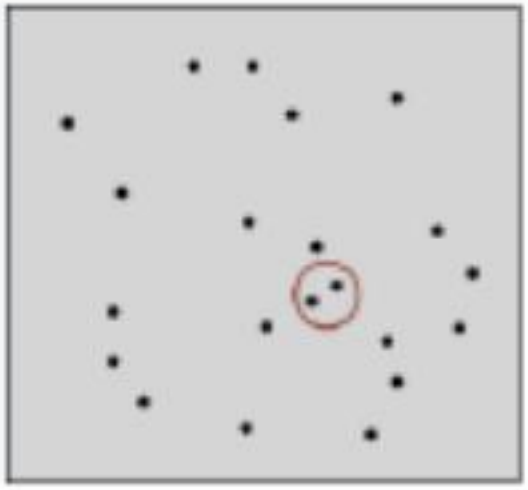
Avec les valeurs des paramètres a, b et k, le théorème maître permet de déterminer "automatiquement" la complexité d'une méthode de type diviser pour régner.

XXVIII.5- Exercice : paire de points les plus proches

Étant donné n points dans un plan, trouver une paire de points les plus proches au sens de la distance euclidienne.

Applications : vision, systèmes d'informations géographiques, contrôle aérien, modélisation moléculaire...

1- Algorithme naïf : tester toutes les paires de points en $\Theta(n^2)$



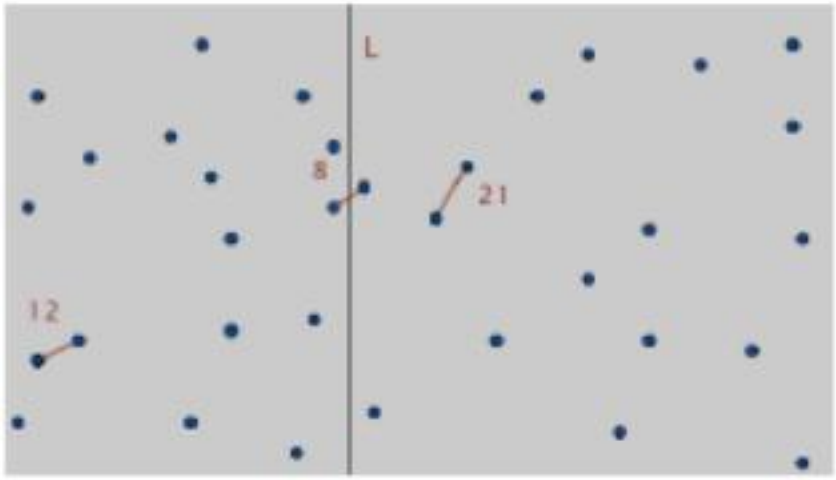
2- Approche Diviser pour régner :

Diviser : tracer une droite verticale L de façon à obtenir n/2 points dans chaque sous-région (médiane); nécessite un tri;

Régner : trouver la paire de points les plus proches dans chaque sous-région

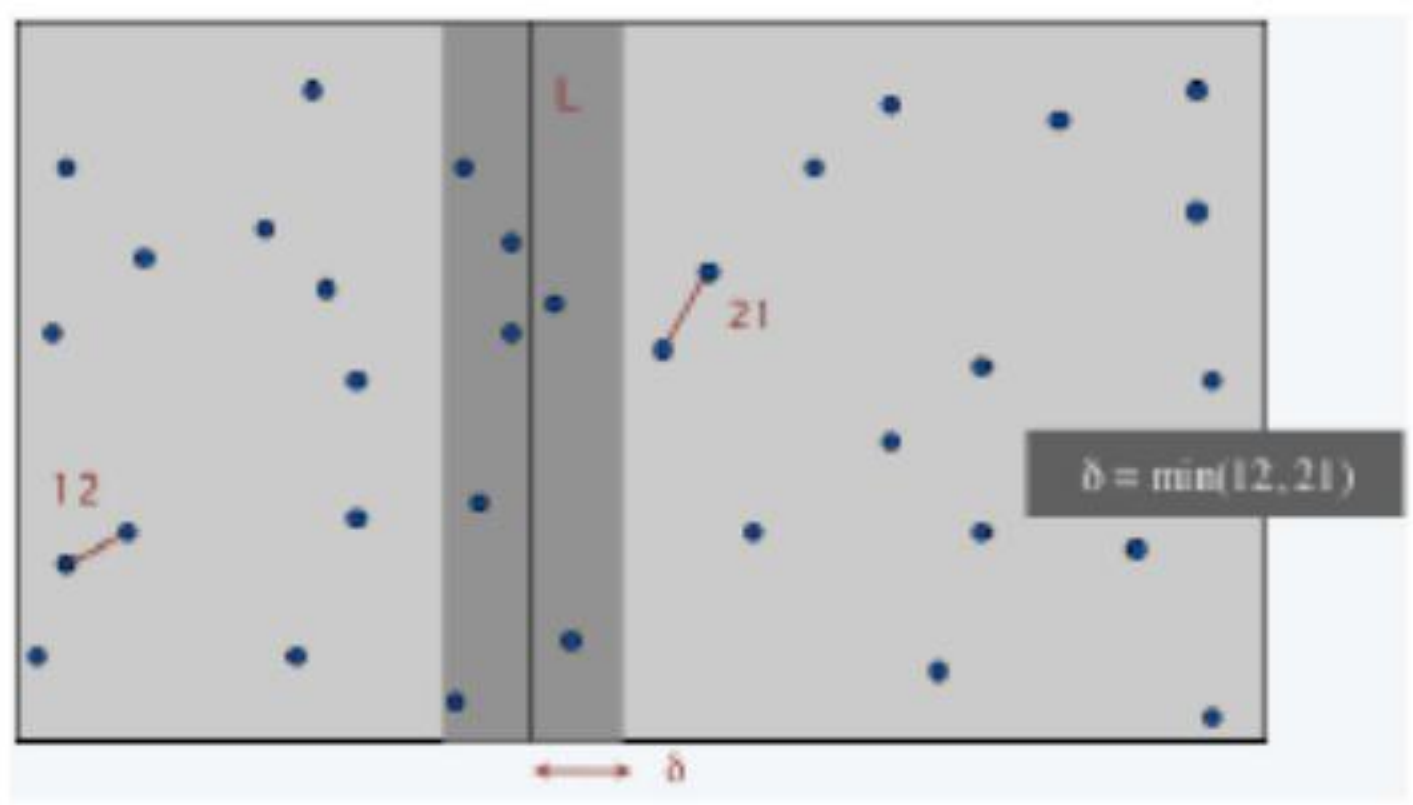
trouver la paire de points les plus proches avec un point dans chaque sous-région

Combiner : comparer et renvoyer la meilleure des trois solutions trouvées

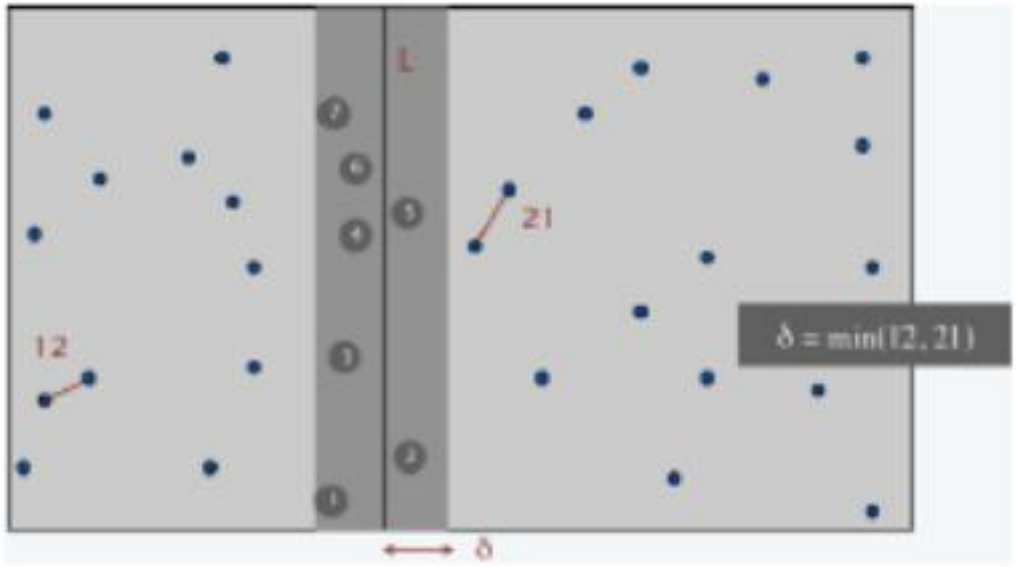


Comment trouver la paire de points les plus proches avec un point dans chaque sous-région ?

Idée : on peut se contenter d'examiner seulement les points de distance ≤ δ de la droite L, où $\delta = \min(d_{min}(regG), d_{min}(regD))$



Donc, on peut trier les points dans la bande de largeur **2**δ selon leurs ordonnées.



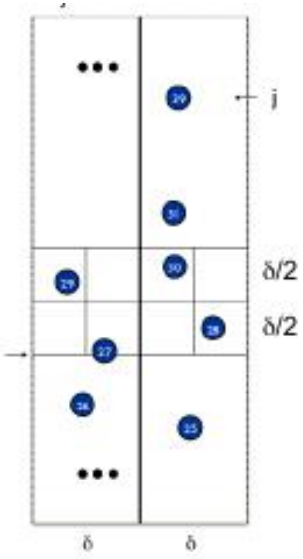
Soit s_i le point de la bande de largeur 2δ de ième plus petite ordonnée

Propriété Si $|i-j| \geq \delta$, alors la distance entre s_i et s_j est $\geq \delta$.

Preuve : 1 seul point dans chaque carré

Deux points séparés par au moins deux bandes

horizontales sont distants d'au moins $2(\delta/2)$



PairePlusProche($p_1, \ldots, p_n$) :

Si n <= 3 alors renvoyer le min(distances) entre les points (∞ si n=1)
Sinon calculer la ligne de séparation L

1. d_{min} (regG) ← PairePlusProche(points regG)
2. d_{min} (regD) ← PairePlusProche(points regD)
3. $\delta = \min(d_{min}(regG), d_{min}(regD))$
4. Supprimer dans L tous les points à distance supérieure à δ
5. Examiner les points dans l'ordre de leurs ordonnées croissantes et
   Comparer la distance entre chacun avecles 8 suivants.
   Si une de ces distances est ≤ δ, mettre à jour δ
6. Renvoyer(δ)

Étapes 1 et 2 : division de l'ordre 2T(n/2)

Étapes 4,5 et 6 : recomposition de l'ordre O(n)

On dispose de deux listes des points triés par ordre d'abscisses croissantes et d'ordonnées croissantes (calculées une fois pour toute en O(n log n))

Calcul de la complexité :

Soit T(n) la fonction complexité (big-oh) de **PairePlusProche(p 1,...,pn)**

On a :

$$T(1)=O(1)$$

$$T(n)= T(\lfloor n/2 \rfloor)+ T(\lceil n/2 \rceil)+O(n) = 2\ T(n/2) + O(n)$$

La complexité de l'algorithme **diviser pour régner** pour le calcul d'une paire de points les plus proches est $O(n \log n)$.

A comparer oà $O(n^2)$

N.B. : il existe une solution à cet exercice par l'utilisation des quad-trees.

XXIX- Complexité : complément technique

Ci-dessous, quelques exemples qui montrent différentes autres techniques de réécriture algébriques employées pour le calcul de la complexité.

XXIX.1- Changement de variable (cas homogène)

Nous avions vu un premier exemple de changement de variable :

Preuve par induction de la complexité de Hanoi.

D'autres exemples ci-après.

XXIX.1.1- Exemple : Recherche dichotomique

Pour un cas dichotomique tel que :

$T(n) = T(n/2) + 1$ $\quad n>1$ et $n = k^2$

$T(1) = 1.$

- On pose $n=2^k \rightarrow \quad k = \log n$ et donc $T(2^k) = T(2^k/2) + 1 = T(2^{k-1}) + 1$
- Poser ensuite $T(2^k) = t_k$ d'où $t_k = t_{k-1} + 1$

  --> On sait résoudre cette équation par le théorème 3 et obtenir

  $t_k = c_1 + c_2\, k$ et donc $T(2^k) = c_1 + c_2\, k$
- Ensuite, on substitue *n* pour 2^k et *log(n)* pour *k* $\rightarrow$ $T(n) = c_1 + c_2 \log(n)$
- On obtiendra $c_1=c_2=1$ par ailleurs et au final :

  $T(n) = 1 + \log(n)$

XXIX.1.2- Exemple : Tri Fusion

On a $T(1)=T(0)=0$

$T(n) = 2\ T(n/2) + n$

- Soit $n=2^k$ d'où $k=\log(n)$
  --> On aura $T(2^k) = 2\ T(2^k/2)+ 2^k$. -> $T(2^k) = 2\ T(2^{k-1})+ 2^k$.
- Soit $t_k = T(2^k)$ pour tout entier k.
  --> D'où l'équation de récurrence $t_k= 2t_{k-1} + 2^k$
  → l'équation caractéristique $r^k = 2\ r^{k-1} + 2^k$.
- En accord avec le théorème 3 (non homogène), on aura $(r^k - 2\ r^{k-1})(r- 2)=0$.
  → qui donne (r-2)(r-2) : une racine double r=2.

- La forme générale de la solution $tk= c_0+ c_1\ 2^k+ c_2\ k\ 2^k$.

→ Et donc $t_n= c1\ n + c2\ n \log(n)$.

- Les calculs des coefficients donnent c1=0 et c2=1/2.

→ D'où $t(n) = 1/2\ n \log(n) = O(n. \log(n))$

N.B. : une autre technique ad-hoc pour ce même exemple :

On peut utiliser une soustraction pour aboutir aux mêmes résultats :

- On avait (de la solution précédente) :

$$(1) \quad t_k = 2t_{k-1} + 2^k.$$

Et on pose $$(2) \quad t_{k-1} = 2t_{k-2} + 2^{k-1}.$$

On multiplie (2) par 2 : $$(3) \quad 2t_{k-1} = 4t_{k-2} + 2^k.$$

On soustrait (1) et (3) : $$t_k = 4t_{k-1} - 4t_{k-2}\ .$$

$$\rightarrow \quad r^k - 4\, r^{k-1} + 4\, r^{k-2} = 0$$

$$\rightarrow \quad r^{k-2}\,(r^2 - 4r + 4) = 0 \quad \rightarrow \quad (r^2 - 4r + 4) = (r-2)(r-2).$$

Les racines : r=0, r=2 (racine double)

et la forme générale : $t_k = c0 + c1\ 2^k + c2\ k\ 2^k$.

Le reste est identique à la solution précédente.

XXIX.2- Changement de variable dans une cas non homogène

La récurrence de la forme

$$t_n = a\, t_{n/b} + f(n)$$

est résolu pour obtenir T(n) en posant

$S(n) = T(b^k)$ et donc

$s_k = t_{b^k} = a\, t_{b^k/b} + f(b^k) = a\, t_{b^{k-1}} + f(b^k) = a\, s_{k-1} + f(b^k)$.

Ce qui nous ramène aux cas précédents.

Exemple : ⮒.

XXIX.2.1- Exemple

Soit $t_n = 3\ t_{n/2} + n\ lg(n)$ pour n>1 et $t_1=2$

On pose $S(n) = T(2^k)$ avec $n=2^k \rightarrow k=lg(n)$ et donc

$$s_k = t_2{}^k = 3\ t_2{}^{k}{}_{/2} + 2^k\ .\ lg(2^k) = 3\ t_2{}^{k-1} + 2^k\ .\ k = 3\ s_{k-1} + k.2^k$$

La résolution de ce cas non homogène donnera :

$(r-3)(r-2)^{1+1}=0$ et donc $s_k = c1\ 3^k + c2\ .\ 2^k + c3\ k.\ 2^k$.

- Pour les conditions initiales sur S :

  $S(0) = T(2^0) = T(1) = 2$

  $S(1) = T(2^1) = T(2) = 8$ que l'on calcule à l'aide du système initial

  $S(2) = T(2^2) = T(4) = 32$

- On peut maintenant calculer les constantes et obtenir :

  $S(k) = 8.\ 3^k - 6\ .\ 2^k - 2.\ k.\ 2^k$.

On s'occupe maintenant de calculer T(n) :

Avec $S(n) = T(2^k)$ et $k=\lg(n)$, on a :

$T(n) = S(\lg(n))$

$$T(n) = 8.\ 3^{\lg(n)} - 6\ .\ 2^{\lg(n)} - 2.\ \lg(n).\ 2^{\lg(n)}.$$

N.B. : $3^{\lg(n)} = (2^{\lg(3)})^{\lg(n)} = (2^{\lg(n)})^{\lg(3)} = n^{\lg(3)}$.

- La forme finale de la complexité sera donc $T(n)=8n^{\lg(3)} - 6n - 2n\lg(n)$

→ Sachant que $1 < \lg(3) < 2$ (car $2^1 < 3 < 2^2$)

on a une complexité $O(n^2)$ pour $n=2^k$.

Une remarque générale :

Une règle appelée la ***règle de lissage*** permet d'affirmer ceci.

si T(n) = Θ(f(n)) pour **n** une puissance de b >= 2

et si f(n) est nice polynomiale

--> f(n) est dite *nice si* elle n'est ni *exponentielle* ni *factorielle*

alors T(n) est réellement Θ(f(n)) pour toute valeur de n

Dans le cas de l'exemple présent, n > 1.

XXX- Cas particulier : racines imaginaires !

Il arrive (rarement) que l'on ait besoin de racines imaginaires.

XXX.1- Exemple 1

Soit $U_n = U_{n-1} - 2\,U_{n-2}$

$U_0 = \ldots$

Pas très simple : les racines seront complexes.

On obtient (avec l'aide de Maple) : $Un = \left(\frac{U_0}{2} - i\left(\frac{2u_1-u_0}{2\sqrt{7}}\right)\right) r_1^n + \left(\frac{U_0}{2} + i\left(\frac{2u_1-u_0}{2\sqrt{7}}\right)\right) r_2^n$

La résolution donnera $r_1, r_2 = \frac{1 \pm i\sqrt{7}}{2}$

XXX.2- Exemple 2

- Soit

  $t_n = 2\ t_{n-1} - 2\ t_{n-2}$,

  $t_0=0$,

  $t_1 = 2$

Pour le cas $t_n = 2\ t_{n-1} - 2\ t_{n-2}$, dont l'équation caractéristique est $r^2-2r+2 = 0$

→ les racines seront $r=1+i$ et $r=1-i$ (et r=0 par ailleurs).

→ dans ce cas, on aura $T(n) = c1\ (1+i)^n + c2\ (1-i)^n$.

- On obtient des conditions initiales $c1+c2=0$ et $c1-c2=-2i$

→ et donc $T(n) = i.[(1-i)^n - (1+i)^n]$

→ Cette fonction oscille entre $\sqrt{2^n}$ et $-\sqrt{2^n}$ avec une période de 4 ;

en particulier, on a $T(4n) = 0$ car $(1-i)^4=(1+i)^4 = -4$ ⮒

N.B. : la présence de racine imaginaire laisse supposer un terme (ici la complexité) négatif, mais il se peut que $T(n) > 0$.

P. Ex. : si les racines d'un équation sont 2, 1+i et 1-i on aura une complexité oscillant autour de $c2^n$ (supposons des conditions initiales ad-hoc) qui sera toujours positif.

XXXI- Compléments

Cette partie contient des complément sur les points abordés dans ce cours.

- Classe NP
- Ex de problèmes indécidables
- Définition Little-oh et little-oméga
- Complexité et vitesse de calcul
- Idée intuitive Big-Oh et big-Oméga
- Sur les algorithmes et les ressources de calcul
- Stratégies et complexité (recherche de "vis")
- Complexité et 'bon' et 'mauvais' algorithme
- Résumé et remarques sur les ordres
- Complexité et stratégies : n-reines
- Complexité et stratégies : table des moyennes
- Complexité et stratégies : : anagramme
- Complexité et stratégies : : égalité de 2 tableaux

- Détails algorithme mariage stable
- Algo médiane

- La complexité de Fib est dans un intervalle
- Remarques sur Fib
- Une autre solution (matricielle) à Fib
- Fib et les lapins

- Exercice : coloration de graphes
- Exercice de complexité : Tris, Fib (autres)

- Exemples de complexité (homogène) : Hanoi
- Exemples de complexité (non homogène) : 4 exemples

XXXI.1- Classe d'algorithmes NP

On dit qu'un problème est **NP-complet** (par exemple, le *TSP*) quand :

- On ne lui connaît pas d'algorithme polynomial (c-à-d. : $O(n^k)$, k constante).
- Mais on ne connaît pas non plus la preuve de l'**inexistence** d'un tel algo.

Exemples NP :

- La méthode RSA et la difficulté de la factorisation des nbrs. entiers
- Autres exemples NP :

  Sac à dos, Bin Packing, Flux, Coloration de graphes, etc

Pas de panique : on traitera des cas plus simples (*Cavalier, N-reines, SENDMOREY, Dijkstra, MST, ..)*

En plus de la classe NP, certains problème sont réputés indécidables

--> (pas d'algorithme du tout !).

XXXI.2- Exemples de problèmes indécidables

1- "The Halting problem" : étant donné un programme P et des données D en entrée, existe-t-il un algorithme finissant en un temps fini capable de décider si P termine en un temps fini.

2- Un(e) variant(e) de ce problème :

étant donné un programme P et des données D en entrée, existe-t-il un algorithme finissant en un temps fini capable de décider si P boucle (à l'infini).

3- Carrelage / remplissage de n'importe quelle surface avec un ens. quelconque de petits carreaux (e.g. surface 20 x 20 cm et petits carreaux de 3 x 3 cm).

4- Un variant : la forme de la surface / petits carreaux est libre !

5- Déterminer en un temps fini si l'on peut résoudre une équation "Diophantienne" en un nombre fini d'opérations est indécidable.

Une équation "Diophantienne" :

C'est une équation polynomiale avec plusieurs inconnues et à solutions dans / et avec des coefficients dans Z

(dont l'une des plus célèbres est l'identité de *Bezout* ax+by=c, utilisé pour la preuve de l'algorithme d'Euclide).

6- Toute expression en logique des prédicat est-elle démontrable (prouvable) est indécidable.

XXXI.3- Définition du Little-oh (et de little-oméga)

- Dans les déf. de O, Ω et Θ, on a les relations $\leq$ et $\geq$.

--> En passant à < et > (stricte), on définit **o** (little-oh) et ω (little-omeag).

- P. Ex., certains logiciels Temps Réels exigent little-o.

Définition : soit f et g deux fonctions de $\mathbb{R}$ dans $\mathbb{R}$.
On dit que **f** est d'ordre de **o(g)** si l'on peut trouver un réel x_0 et **pour tout réel** positif **c** tel que $\forall x \geq x_0 > 0,\ f(x) \leq c.\ g(x)$.

- Ici, la relation se vérifie pour **toute constante c**.
- La définition de O(.) utilise "$x_0 \geq 1$" et le terme "*il existe un réel positif c > 0* " tandis que la définition de (little-oh), on a "$x_0 > 0$" et "*pour tout c*".

N.B. : si on a **T(N)= o(f(N))**, on a **T(N) = O(f(N))** mais **T(N) ≠ Θ(N)**.

→ En d'autres termes, o(N) = O(N) avec '<' strict.

O(N) peut être Θ(N) mais pas o(N).

Définition : Si f(n) = o(g(n)) Alors f(n) ∈ O(g(n)) implique f(n) ∈ Ω(g(n))

C-à-d : f(n) est O(g(n)) mais pas dans Ω(g(n)).

De manière analogue, on définit ω (little-oméga) par rapport à Ω.

Les "littles" apportent une précision accrue :

- ***Little-oh*** est Pessimiste mais **plus précise** que *big-oh*
- ***little-oméga*** (ω) est Optimiste mais **plus précise** que *big-Oméga*

--> On connaît les limites que l'on n'atteindra pas !

Exemple : $f(n) = 12n^2 + 6n$ est $\mathbf{o(n^3)}$ et $\omega(\mathbf{n})$.

f(n) sera toujours au-dessous $\mathbf{n^3}$ et toujours au-dessus de **n**,

sans jamais être "="

XXXI.4- Remarque sur la Complexité et vitesse de calcul

Un algorithme rapide contient au pire une complexité polynomiale.

Un algorithme lent (qui ne peut pas garantir une réponse rapide) contient un terme qui croît plus rapidement que tout polynôme (de la forme e.g. eL ou) .

La notion de rapidité ou lenteur dépend en général du nombre de bits nécessaires pour coder les entrées (dépendance aux structures de données).

On dira qu'un algorithme est traçable (tractable) s'il est rapide (il n'est pas lent).

Un exemple : soit l'algorithme *A* qui produit, pour une donnée en entrée de taille L, une réponse en au plus $7L^3$ minutes.

L : longueur d'un tableau (longueur en nombre de bits d'un entier)

Si pour le même pb. , un algo. *A'* garantie une réponse en au + $0.57L^{23}$min.

→ Alors *A est plus rapide que A'*

XXXI.5- Idée intuitive: big-Oh et Oméga

Exemple :

soit à estimer la croissance de la somme

$$f(n) = \sum_0^n x^2 = 1^2 + 2^2 + 3^2 + \ldots + n^2.$$

1- On sait que :

$$\sum_1^{n-1} x^2 \;\leq\; \int_1^n x^2\,dx \;=\; \left[\frac{x^3}{3}\right]_1^n \;=\; \frac{(n^3-1)}{3}$$

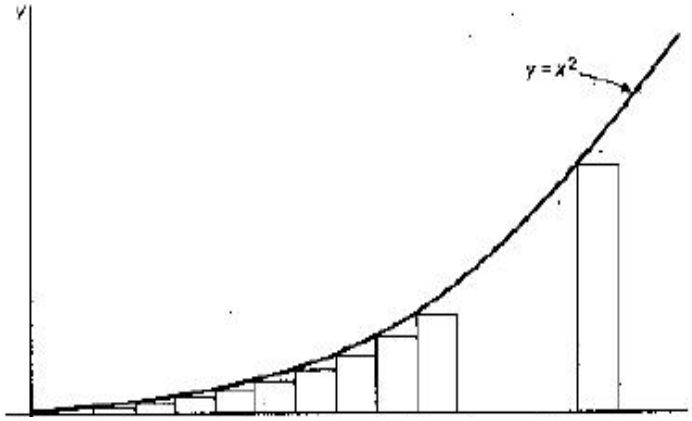
Si on passe de n à n+1 (big-O):

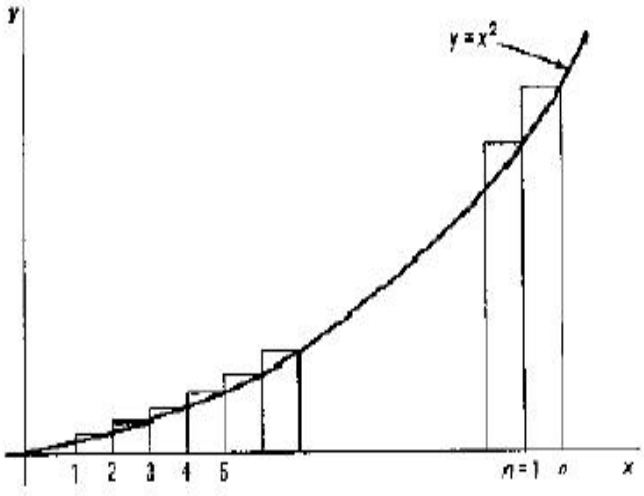
$$f(n) \;\leq \int_1^{n+1} x^2\,dx \;\leq\; \frac{\left((n+1)^3-1\right)}{3}$$

2- De même (Oméga) :

$$f(n) = 1^2 + 2^2 + 3^2 + \ldots + n^2 \;\geq\; \int_0^n x^2\,dx \;=\; \frac{n^3}{3}$$

d'où $\int_1^{n+1} x^2\,dx \;\geq\; f(n) \;\geq\; \int_0^n x^2\,dx$





XXXI.6- Algorithmique et les ressources

Algorithmique : l'étude de la conception et l'analyse des algorithmes.

On a vu : une démarche (méthode) pour aboutir, d'un état donné (initial) à un état final en décrivant les transitions successives d'états (séquentiel vs. Parallèle).
→ La question de ressource est prise en compte dans l'algorithmique.

Question de ressources disponibles :

- Calculer sur un ordi., un boulier, une machine "FACIT", ...

- Les ressources en temps de calcul et en espace mémoire?

  - **Espace** : l'unité *bit* (ou *octet*); universellement admise.

  - **Temps** : une sec. sur un processeur récent n'est pas la même que sur un 6802.

- **Les premiers ordinateurs :** dans les années 40 (Turing "Bombe" puis "Colossus")
- **Les premiers algorithmes :** l'antiquité (Euclide, PGCD).

XXXI.7- Complexité : 'bons' et 'mauvais' algorithmes

- Classification grossière :

  Algorithmes **polynomiaux** (de complexité de l'ordre d'un polynôme) et les autres dits **exponentiels**.

--> Exemple de complexité **polynomiale** : log n, $n^{0.5}$, n log n , n^2, ...

--> Exemple de complexité **exponentielle** : e^n, 2^n, $n^{\log n}$, n! , n^n , ..

(n $^{\log n}$ est sous exponentielle)

- **Un bon algorithme est polynomial.**

Ce critère d'efficacité est confirmé par la pratique :

Une exponentielle dépasse tout polynôme pour **n** assez grand.

Par exemple, $\mathbf{1.1^n}$ croit d'abord lentement mais finit par dépasser $\mathbf{n^{100}}$.

- En pratique, il est rare de trouver / utiliser des algorithmes polynomiaux d'ordre supérieur à 5 (n^5).
- Les polynômes ont des propriétés de fermeture intéressantes.

--> L'addition, la multiplication et la composition des polynômes des polynômes.

On peut construire de grands algos. polynomiaux à partir des petits.

Remarque :

- Une complexité polynomiale d'ordre **n** est meilleure si pas de terme d'ordre < **n**

- De même : n^2 est mieux que $n^2 + 15n$ même si les deux sont $O(n^2)$.

- Cette même complexité est encore meilleure si le coefficient de n^2 diminue :

--> n^2 est mieux que $2n^2$ même si les deux sont $O(n^2)$.

XXXI.8- Résumé et remarques sur les ordres

- L'ordre de croissance de T(N) ≤ f(N) --> **T(N) = O(f(N))** big-oh
- L'ordre de croissance de T(N) ≥ f(N) --> **T(N) = Ω(f(N))** Oméga
- L'ordre de croissance de T(N) ≃ f(N) --> **T(N) = Θ(f(N))** théta
- L'ordre de croissance de T(N) < f(N) --> **T(N) = o(f(N))** little-oh

--> *little-oh* n'accepte pas l'égalité (=) comme le fait big-oh

- **T(N) = O(f(N))** : T(N) ne va pas croître à une vitesse plus grande que f(N).

--> f(N) est une **borne supérieur pour / de T(N).**

- **T(N) = O(f(N))** --> f(N) = Ω(T(N)) --> T(N) est une borne **inférieure** de f(N).

Par exemple, N^3 croît plus vite que N^2, on peut donc dire que :

$N^2=O(N^3)$ ou $N^3=\Omega(N^2)$.

- Avec $f(N)=N^2$ et $g(N)=2.N^2$, les deux croient à la même vitesse,

→ donc : $f(N)= O(g(N))$ et $g(N) = \Omega(f(N))$.

- Quand deux fonctions croissent au **même taux**, on pourra **utiliser** Θ selon les contextes.

--> Par exemple, si $g(N)=2.N^2$, on a $g(N) = O(N^4) = O(N^3) = O(N^2)$.

--> Cependant, $O(N^2)$ est la meilleure estimation (la facteur 2 de $2.N^2$ peut dépendre de l'ordinateur).

--> Si l'on écrit **$g(N) = \Theta(N^2)$**, on dira que non seulement $g(N)=O(N^2)$ mais aussi que le résultat est **très proche de l'estimation**.

Notons : avec les catégories de complexité (avec k > j > 2 et b > a > 1) :

$\Theta(\log n) < \Theta(n) < \Theta(n \log n) < \Theta(n^2) < \Theta(n^j) < \Theta(n^k) < \Theta(a^n) < \Theta(b^n) < \Theta(n!)$

--> Si une fonction de complexité g(n) est dans une catégorie qui est à gauche (ordre ci-dessus) de celle contenant f(n), alors g(n) o(f(n)) (strictement inférieur)

--> On constate que toute complexité logarithmique est meilleure que les polynomiales, et les polynomiales sont meilleures que les exponentielles, et les exponentielles sont éventuellement meilleures que les factorielles.

Par exemple : $\log n < o(n), < n^{10} < o(2^n), < 2^n < o(n!)$

- Pour c >= 0, d > 0,

Si $g(n) \in O(f(n))$ et $h(n) \in \Theta(f(n))$

alors $c.\, g(n) + d\, .\, h(n) \in \Theta(f(n))$ (et bien sur $O(f(n))$)

- $g(n) \in O(f(n))$ ssi $f(n) \in \Omega(g(n))$

  $g(n) \in \Theta(f(n))$ ssi $f(n) \in \Theta(g(n))$.

Autres remarques :

- Si **a,b>1**, $\log_a n \in \Theta(\log_b n)$

Les complexités logarithmiques sont toutes dans la même catégorie.

Exemple : $\Theta(\log_4 n) = \Theta(\log n)$

--> la relation entre $\log_4 (n)$ et $\log(n)$ est du même ordre (Θ) que $7n^2+5n$ vs. n^2

- De même : si **a,b>0**, $a^n \in o(b^n)$

Les complexités exponentielles NE sont PAS toutes dans la même catégorie.

- Pour tout a>0, $a^n \in o(n!)$ voir ex. limite (un peu + loin).

n! est pire que toute fonction de complexité exponentielle.

XXXI.9- Complexité et Stratégies / Heuristiques dans les algorithmes

Rappel : la stratégie, la logique et le contrôle, les structures de données et les heuristiques utilisées peuvent jouer un rôle important dans la complexité.

- **Exemple 1** : rechercher un nom dans l'annuaire de 20 millions de références.

  Une recherche linéaire ≈ **2×10^7 comparaisons** possibles (cas pire).

  Une recherche dichotomique : $\log (2 \times 10^7) \simeq$ **17** comparaisons possibles (cas pire).

- **Exemple similaire** : rechercher un mot dans un dictionnaire ?

- On développe plusieurs exemples ci-dessous.

XXXI.9.1- Stratégies et complexité (ex. "Vis")

- Il y a souvent plusieurs méthodes (algorithmes) pour résoudre un problème.

Dans **Logique + Contrôle** : plusieurs Contrôles possibles pour une même logique.

Exemple de TRI de S : on peut permuter S jusqu'à tomber juste !!

Un Exemple (trivial):

On a 15 boîtes de vis de différentes longueurs rangées dans un bloc de rangement avec 15 tiroirs.

→ **Comment faut-il** ranger ces vis afin de les retrouver facilement ?

1- Naïve : ranger n'importe comment, rechercher de gauche à droite.

--> En moyenne : 8 (longueurs équiprobables et toutes les vis représentées) :

→ La probabilité pour que le $K^{ième}$ tiroir soit le bon, k=1..n est 1/15 ,

→ Pour arriver à l'indice k=1..15, on aura fait les comparaisons :

$\sum \left(\frac{1}{15} * k\right) = \frac{1}{15} \sum k = \frac{1}{15} * \frac{16*15}{2} = 8$ consultations en moyenne.

1'- Variante : ranger les vis selon la fréquence des demandes.

→ tjs 8 en moyenne.

2- Naïve 'Las Vegas' : Naïve mais plus *démocratique*

- Recherche aléatoire (au lieu de gauche-droite) :

--> Consulter un tiroir : si non-marqué et c'est le bon alors trouvé,

sinon marquer celui-ci et choisir un autre...

- L'espérance du nombre de consultations : 8

$$\frac{1}{15} * 1 + \frac{14}{15} * \frac{1}{14} * 2 + \frac{14}{15} * \frac{13}{14} * \frac{1}{13} * 3 + \frac{14}{15} * \frac{13}{14} * \frac{12}{13} * \frac{1}{12} * 4 + \dots$$
$$= \frac{1}{15} * \frac{2}{15} * \frac{3}{15} * \dots * \frac{15}{15} = \frac{1}{15} \sum i = 8$$

--> Le nombre de consultations à chaque recherche varie et dépend du choix init.

--> Ce même calcul peut aussi être fait pour le cas naïf.

3- Tri : on range les vis selon la longueur de la plus petite à la plus grande.

- On cherche par la méthode Dichotomique.

  --> Au pire, on fera 4 consultations ($\approx \log_2 15$).

  --> En moyenne 3,26 comparaisons : supposons trouver à l'indice 15 (cas défavorable)

- trouver en 1e tentative sur le tiroir du *milieu* (soit T_8) : 1/15 T=Tiroir
- en 2e tentative : ne pas trouver en T_8 et trouver en T_{12} : 14/15 * 1/7 = 2/15
- trouver en 3e : ne pas trouver en T_8 ni en T_{12} et trouver en T_{14}:

  14/15 * 6/7 * 1/3 = 4/15 , .. et enfin, trouver en T_{15}.

  La somme : $\frac{1}{15} * 1 + \frac{2}{15} * 2 + \frac{4}{15} * 3 + \frac{8}{15} * 4 = \frac{49}{15} = 3{,}26$

☞ **On a supposé que la vis recherchée était (sûrement) présente.**

Dans le cas contraire : la probabilité 1/15 des cases devient 1/30 (voir + loin)

XXXI.9.2- Complexité et stratégies : le problème de N-reines

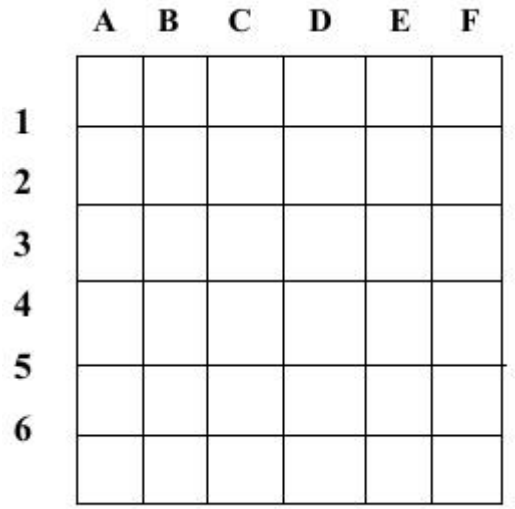
--> Problème général d'**affectation sous contraintes.**

But : Placer N pions sur un échiquier N x N de sorte que :

C1 : 2 pions ne soient pas sur la même colonne

C2 : 2 pions ne soient pas sur la même ligne

C3 : 2 pions ne soient pas sur la même diagonale.



Idées de la combinatoire (cas pire)

1- N variables (pions) et pour chacune, une case (1..N^2) possibles

--> **$N^2!$** possibilités (si l'on ne mettra pas 2 pions dans la même case);

--> sinon, il y a **$(N^2)^N$** possibilités.

2- Considérer chaque case de l'échiquier comme une variable (N^2 variables) dont le domaine est {vrai, faux}.

⇨ Une case=*vraie* si elle contient un pion dans la solution; fausse sinon.

⇨ Combinatoire = $(2^N)^2$.

3- Associer un pion par ligne (ou colonne).

⇨ N^N possibilités mais plus efficace.

Aperçu de quelques méthodes de résolution de N-reines :

1. **Naïve** : on jette les pions au hasard et on vérifie les contraintes ($O(N^2)^N$)
2. **Constructive** : réserver une colonne par pion (supprime la contrainte C1).

   - Puis, placer un pion sur une ligne et vérifier les contraintes C2 et C3;

   - Utiliser le mécanisme de **retour arrière** pour les autres tentatives :

   - Pour tout état partiel E ainsi crée :

     Si E = état final alors fini

     Trouver un nouvel état partiel E = successeur(E) respectant C2 et C3,

     Recommencer.

   --> **Combinatoire** : N^N possibilités

3. **Centrée Contraintes** : vérifier en permanence la validité d'un ensemble de contraintes --> **Complexité** théorique : N^N, mais méthode très efficace.

XXXI.9.3- Complexité et stratégies : égalité de 2 tableaux

- Soit 2 tableaux T1 et T2 d'entiers de taille N dont les éléments sont **uniques**.

--> Vérifier que les deux tableaux contiennent les mêmes éléments.

XXXI.9.3.a- Première version (en C++)

- On vérifie que chaque élément de T1 figure dans T2.
- On sait que les deux tableaux ont la même taille.

```
bool  egaux(vector<int> & T1, vector<int> & T2)
{int N=T1.size();                    // la même taille pour T2
 for (int i=0; i<N; i++)
        {for (int j=0; j<N; j++)      if (T1[i] == T2[j]) break;          // on a trouvé, sortie de la boucle interne
         if (j == N)
        return false;
  }             // on a jamais trouvé T1[i] dans T2
 return true;
}
```

- **Analyse de la première version :**

La boucle externe est exécutée N fois.

La boucle interne est exécutée (au pire) N fois --> $O(n^2)$

Remarque : une étude plus approfondie permet de constater :

La boucle interne conduit à un nombre de comparaisons : 1+2+ ... N

--> On a une complexité = $\frac{n.(n+1)}{2}$= $\Theta(n^2)$ car **à la fois $O(n^2)$ et $\Omega(n^2)$**

XXXI.9.3.b- Deuxième version (en C++)

- Utiliser un bon algorithme de tri (*O*(N. log N)) puis faire une comparaison *O*(N).

```
bool  egaux(vector<int> & T1, vector<int> & T2)
{int N=T1.size();          // c'est la même taille pour T2
 vector<int> Temp1(N), Temp2(N);
  sort(T1, Temp1);
  sort(T2, Temp2);

  // en cas d'égalité des deux, Temp1[i] = Temp2[i], pour i=0..N-1
 for (int i=0; i<N; i++)
         if (T11[i] != T21[i])   return false;
 return true;
}
```

Analyse de la deuxième version :

Le tri des deux tableaux : 2. *O*(N.log N) , La boucle : *O*(N)

La complexité = 2. *O*(N.log N) + *O*(N) = *O*(N.log N)

Remarques sur cet exemple :

- On peut faire pire : trier les deux tableaux mais appliquer le premier algorithme !
- Encore pire : pour le tri, utiliser un algorithme $O(N^2)$ mais appliquer première version de l'algorithme !

- **Exercice** : étudier le cas où les éléments des deux tableaux ne sont pas forcément uniques.

XXXI.9.4- Complexité et stratégies : anagrammes

- Soient deux chaînes de caractères (2 phrases) données dans deux tableaux T1 et T2 de taille N1 et N2 (on vérifiera si les 2 tailles sont identiques).
- Vérifier si on a une **anagramme** (mots différents utilisant les mêmes lettres).

Solution : Traitement commune à toutes les solutions proposées :

– Supprimer les espaces dans chaque phrase : O(N) avec N=max (N1,N2).

– Si les deux tableaux restants n'ont pas la même taille N alors *échec*

– Transformer tous les caractères de T1 et de T2 en minuscule (ou en majuscule) (complexité O(N))

→ Complexité de la partie commune : 4. O(N) = O(N)

- **1- Première méthode :**

– Trier T1 et T2 par une méthode : O(N.log N)

– Comparer l'égalité des deux tableaux triés : O(N)

→ Complexité globale = **O(N.log N)**

- **2- Deuxième méthode :**

- Créer un tableau Occ[1..26] de lettres pour chacun des tableaux T1 et T2

Occ1[1] = nombre d'occurrences de la lettre 'a' dans T1 ... ⇒ **O(N)**

- Comparer les tableaux Occ1 et Occ2 : **O(N)**

→ Complexité globale = **O(N)**

- **3- Une variante :** Créer un seul tableau d'entiers Occ[1..26];

– Le remplir avec les occurrences de chaque lettre dans T1 : O(N)

– Parcourir T2 et décrémenter Occ[i] pour chaque lettre rencontrée dans T2 : O(N)

– Arrêter avec échec dès que Occ[j] < 0, j = 1..26. teste fait à chaque décrémentation

A la fin du traitement, on doit avoir Occ[j]=0, pour j=1..26 : O(N) → Complexité globale = **O(N)**

→ Écrire les algorithmes correspondants.

XXXI.9.5- *Complexité et stratégies : tableau des moyennes cumulatives*

Soit T un tableau de N entiers.

Créer le tableau A tel que A[i] = moyenne de T[1] .. T[i]

$$A[i] = \frac{\sum_{j=1}^{i} T[j]}{i}$$

→ On peut avoir un algorithme $O(N^2)$ ou un $O(N)$ ../..

XXXI.9.5.a- Première version

Pour chaque élément de *A*, on calcule la moyenne (depuis T[1]) .

```
def moyennes1(T): # T est de la classe vector_d_entiers
  N=T.size()
  A = vector_d_entiers(N)
  for i in range(N):
        Somme_jsq_i=0
        for j in range(i+1):          # On recalcule la somme à chaque fois
          Somme_jsq_i += T[j]
        A[i] = Somme_jsq_i / (i+1)
 return  A
```

Analyse de la première version :

La boucle externe est exécutée N fois. Celle interne au pire N fois

✍ 1+2+... N = $\frac{n.(n+1)}{2}$ ✍ La complexité = $O(n^2)$.

Y a-t-il une version d'une meilleure complexité ?

XXXI.9.5.b- Deuxième version

→ Conserver en permanence la somme des éléments T[0] .. T[i].

```
def moyennes2(T): # T est de la classe vector_d_entiers
  N=T.size()
  A = vector_d_entiers(N)
  Somme_partielle=0
  for i in range(N):
        Somme_partielle += T[i]
        A[i] = partielle / (i+1)
 return  A
```

- **Analyse de la deuxième version :**

la complexité = *O*(n). calcul trivial.

XXXI.10- Mariages stables (stable matching)

Construction / vérification d'un couplage stable.

Un couplage est instable s'il contient deux personnes A et B non appareillées (non mariées) ensemble qui se préfèrent mutuellement à leur conjoints.

P. Exemple, on a 'fêté' ces mariages (f,F,g,G): F est mariée avec g

G est mariée avec f

Mais 'ça' ne tiendra pas (instable) car : F préfère G à g

G préfère F à f

Questions :

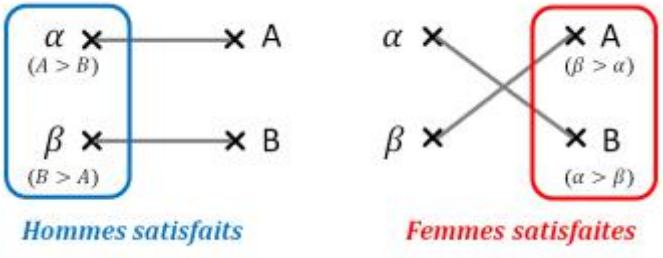
- Commet vérifier qu'un couplage est stable?
- Un couplage stable existe-il toujours? Si oui, peut-on le trouver (par un algo?)

Ce problème n'a pas toujours une solution.

P. Ex : on a deux hommes α et β et deux femmes A et B tels que

α préfère A, Mais A préfère β

β préfère B, Mais B préfère α



Exemple de données :

Des femmes : Alice, Bénédicte, Camille
Des hommes : Élie, François, Gondran

Préférences des femmes :

A: G, E, F B: F, E, G C: G, E, F

Préférences des hommes :

E: A, B, C F: B, C, A G: A, C, B

==> Comment constituer des couples (stables) ?

Applications:

Répartition des biens rares (lorsque les mécanismes habituels et simples ne fonctionnent pas).

Affectation des candidats aux postes / places

Élève vs. École

Travailleur vs. Poste de travail

Internes vs. Services hôpitaux

Étudiants vs. Universités

Élevés 1A vs. PEs 1A (ou stages), etc

Dons d'organes (reins), jurys TIPE !

Etc.

XXXI.11- Algorithmes de mariage stable

XXXI.11.1- Solution par un matching (couplage) maximum

Le couplage maximum dans les graphes bipartis peut être appliquée à n'importe quel problème d'affectation.

Remplacez les gars par des employés et les filles par des tâches.

Les arêtes sont établies si l'employé est qualifié pour accomplir la tâche.

Si les employés ne peuvent gérer qu'une seule tâche à la fois, alors, en tant qu'employeur, votre meilleure ligne de conduite est de choisir un couplage maximum.

Important : dans la suite, nous n'allons pas utiliser la théorie des graphes.

--> Le terme 'matching' utilisé veut dire 'mettre en correspondance' (ne veut pas dire 'couplage' comme en théorie des graphes).

Variant :

Tenir aussi compte des capacités des employés à bien accomplir les tâches.

Dans le problème du mariage, cela nous amène à considérer les préférences des garçons et des filles.

Une façon simple de le faire est d'associer les niveaux d'attraction aux arêtes.

Ces niveaux pourraient être plus élevés pour les couples qui sont plus disposés à aller au bal ensemble.

Le problème sera alors de trouver une mise en correspondance **maximum pondérée**.

Pour créer une instance de ce problème, ajoutons les arêtes manquantes entre les garçons et les filles et donnons-leur un niveau d'attraction nul.

Commençons par trouver un matching maximum (au sens 'mise en correspondance' et non 'matching' de la théorie des graphes).

Ensuite, on peut améliorer les résultats si et seulement si le réseau résiduel dans lequel les arcs inversés s'opposent au niveau d'attraction de l'arc contient un cycle avec un niveau d'attraction positif cumulé. Répéter cela conduira à une correspondance avec le niveau d'attraction maximum.

Une instance du problème des mariages :

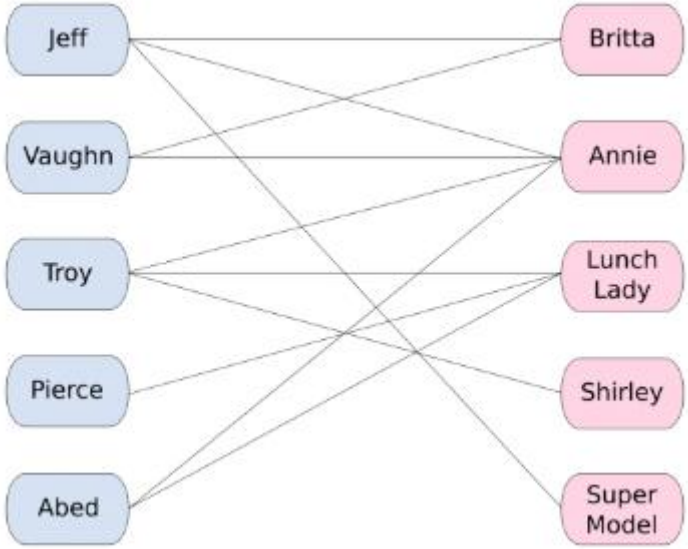
On part de cette instance pour

trouver une solution.

N.B. : Il est également possible d'appliquer

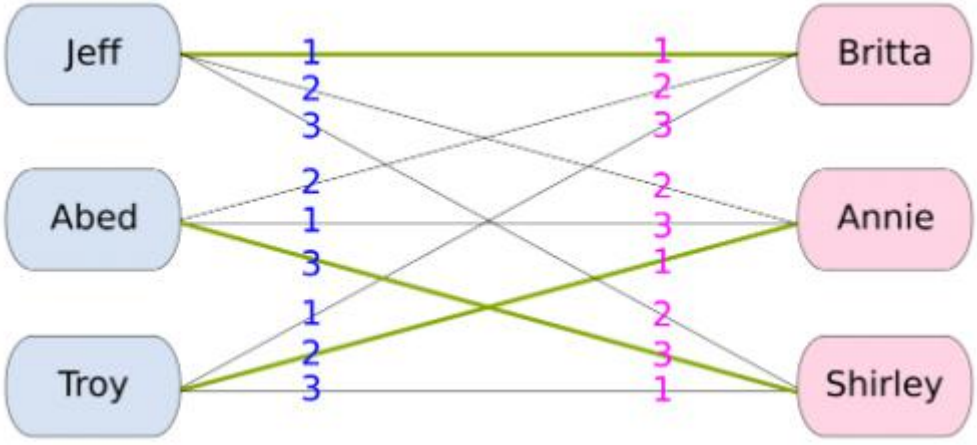
la méthode du flow Maximum et trouver

une solution optimale



(cf. http://www.science4all.org/article/marriage-problem-and-variants/)

Simplifions et ajoutons les préférences (1, 2 et 3 : niveau de préférences) :



La méthode utilisée (algorithme de Gale-Shapley : 1962 qui a valu à Shapley un récent prix Nobel d'économie) :

A chaque itération, les hommes non fiancés se proposent à la fille qu'ils préfèrent qui ne les a pas encore refusés, même s'ils sont fiancés à quelqu'un d'autre.

Dans notre exemple, à la première itération,

Jeff et Troy se proposent à Britta,

Abed se propose à Annie.

Ensuite, les filles choisissent le garçon qu'elles préfèrent parmi ceux avec qui elles sont déjà fiancées et ceux qui viennent de se proposer (elles n'ont pas besoin d'être fidèles !).

Chacun est maintenant fiancé à une personne.

Dans notre cas, à la première itération,

Britta se fiance à Jeff,

Annie est fiancée à Abed.

Dans le cas général, nous continuons à itérer jusqu'à ce que tout le monde soit fiancé.

À la deuxième itération,

Troy est le seul à ne pas être engagé.

Britta l'a refusé, alors il se proposera à sa prochaine fille préférée : Annie.

Annie peut désormais choisir entre son fiancé actuel, Abed, et le challenger Troy.

Elle préfère Troy.

==> Ainsi, elle se fiance à Troy tout en rejetant c dans le processus.

À la troisième itération,

Abed se propose à sa deuxième fille préférée, Britta.

Britta a le choix entre Jeff et Abed, mais elle préfère Jeff, c'est pourquoi elle décide de rester avec son fiancé.

Abed se fait à nouveau larguer et n'est toujours pas fiancé.

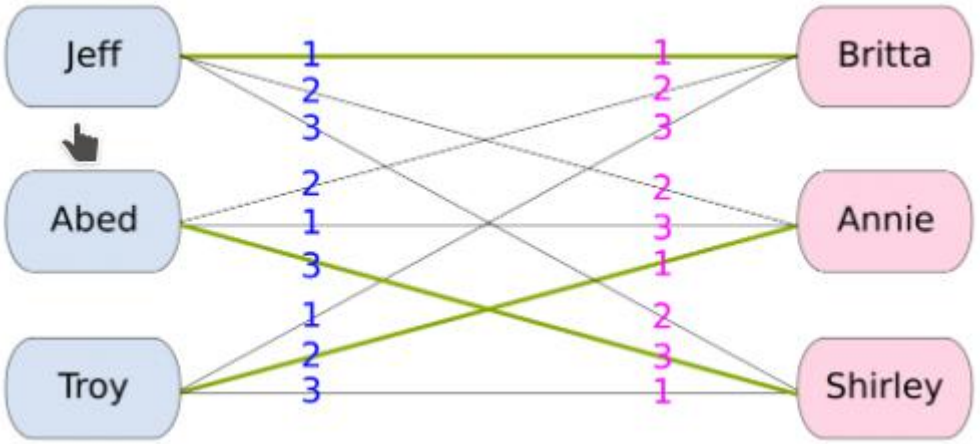
À la quatrième itération,

Abed se propose à sa troisième fille préférée, Shirley.

Shirley n'est pas encore fiancée, alors elle accepte.

Tout le monde est désormais assorti.

L'algorithme se termine par les mariages stables suivant.



Voir http://www.science4all.org/article/marriage-problem-and-variants/ pour la suite des discussions.

Les détails de l'algorithme sont donnés ci-dessous.

XXXI.11.2- Algorithme de mariage stable Chez MitHic

Soit un ensemble de garçons {g1, …, gn}

et un ensemble de filles {f1, …, fn} qui cherchent âme-sœurs.

MitHic demande à chacun (e) d'établir une liste de préférences.

But : mettre les âme-sœurs ensemble.

Notions utilisées : instabilité , engagement.

Au lieu d'appliquer un algorithme exhaustif et naïf qui se perd à vérifier 2 à 2

les compatibilités (voyez p.ex. dans le cas de stages ?)

MitHic applique un algorithme à base de graphe bipartie

→ Dans son principe, cet algorithme cherche à renforcer le "statu quo"

Algorithme d'appariement stable (stable matching) de Gale-Shapley :

```
Initialement, tous les g (garçons) et f (filles) sont libres
Tant que il y a un garçon g libre et non proposé aux filles
        Choisir un garçon g
        Soit f la fille en tête des préférences de g à qui g ne s'est pas proposé
        Si f est libre (non engagée) alors <g, f> deviennent engagés
        Sinon : f est actuellement engagé auprès de g'
            Si f préfère g' à g alors g reste libre
            Sinon : f préfère g à g'
              <g, f> deviennent engagés
              g' devient libre
            Fin si
        Fin si
Fin Tant que
Renvoyer l'ensemble S des couples engagés.
```

Cet algorithme (**appariement stable** =**stable matching**) est à la base de la solution à plusieurs types de problèmes similaires.

XXXI.12- Pseudo-algorithme de calcul de la médiane d'une séquence

```
fonction  médiane(Séquence , Seq_Prec, K, K_prec) :
  Si taille(Séquence) =0 : return -1
  Si taille(Séquence) = taille(Seq_Prec) & K != K_prec : renvoyer Séquence[0]
  Si K=1 & taille(Séquence) = 1 renvoyer Séquence[0]
  pivot=Séquence[0]
  Left = les éléments de Séquence < pivot
  Right = les éléments de Séquence > pivot
  Eq_pivot = les éléments de Séquence = pivot

  Si Left = vide and Right = vide : renvoyer Eq_pivot[K-1]          // tout est dans Eq_pivo
  Choix=[]; Val=0
  Si taille(Left) >= K :
          Choix= Left
          Val=K
  Sinon Si taille(Left) + taille(Eq_pivot) >= K :
                    Choix= Eq_pivot
                    Val= K-len(Left)
                  Sinon :
                    Choix = Right
                    Val = K - taille(Left) - taille(Eq_pivot)
                    renvoyer médiane(Choix, Séquence, Val, K)
```

Exemple : chercher le 5e plus petit élément d'une séquence L

Soit L = une séquence de 100 nombres aléatoires
med = médiane(L, L , Seq_Prec, 5, K_prec)

Pour avoir la médiane, fixer k = taille(L) / 2.

XXXI.13- Fib : pourquoi les lapins remercient Fibonacci ?

Un jour, un éleveur de lapins a soumis à Fibonacci le problème suivant.

Soit: on ouvre un élevage avec un couple de lapins.

Combien de couples de lapins aurais-je en 12 mois si chaque couple engendre tous les mois un nouveau couple à compter du second mois de son existence ?

Janvier : 1 couple	→	Février : 1 couple
Mars : 1 + 1 = 2 couples ,	→	Avril : 2 + 1 = 3 couples
Mai : 3 + 2 = 5 couples ,	→	Juin : 5 + 3 = 8 couples
Juillet : 8 + 5 = 13 couples ,	→	Août : 13 + 8 = 21 couples
Septembre : 21 + 13 = 34 couples,		Octobre : 34 + 21 = 55 couples
Novembre : 55 + 34 = 89 couples,		Décembre : 89 + 55 = 144 couples
Janvier : 144 + 89 = 233 couples		

Ce qui donne :

Soit Fn est le nombre de couples de lapins au mois n.

F(n) = nombre de couples au mois (n-1) + nombre de couples nés au mois n

= nbr. de couples au mois (n-1) + nombre de couples productifs au mois (n-1)

= nbr. de couples au mois (n-1) + nombre de couples nés au mois (n-2)

D'où

F(n) = F(n-1) + F(n-2)

F(1) = 1

F(12) = 233

XXXI.14- Pourquoi Fib(N) est Ω(3 /2)N et O(5/3)N

Selon la formule de *fib(n) = fib(n-1) + fib(n-2) :*

(1) $t_n = t_{n-1} + t_{n-2}$

(2,3) $t_0 = 0,\ t_1 = 1$

- De (1), on obtient $t_n - t_{n-1} - t_{n-2} = 0$.
- En posant $t_n = r^n$: $r^n - r^{n-1} - r^{n-2} = 0$

$$r^{n-2}(r^2 - r - 1) = 0$$ ***équation caractéristique***

- Ce qui donne les racines r = 0, r = $\frac{(1+\sqrt{5})}{2}$ et $\frac{(1-\sqrt{5})}{2}$

sachant que (cas pire) $3/2 < r = \frac{(1+\sqrt{5})}{2} < 5/3$.

⇉

N.B. : le nombre $\frac{(1+\sqrt{5})}{2} \cong$ 1,6180339887 est le **nombre d'or**.

Ce nombre est le ratio entre le terme Fib d'indice (n+1) et le terme d'indice n, lorsque n tend vers l'infini.

En appliquant le théorème ci-dessus, on a :

$$t_n = c_1 \left(r = \frac{1+\sqrt{5}}{2}\right)^n + c_2 \left(r = \frac{1-\sqrt{5}}{2}\right)^n$$

L'application des conditions initiales donne $c_1 = \frac{1}{\sqrt{5}}$ et $c_2 = -\frac{1}{\sqrt{5}}$

La solution générale (qui **donne la valeur du n^{ième} terme** de la suite Fibonacci) :

$$\frac{\left[\frac{(1+\sqrt{5})}{2}\right]^n - \left[\frac{(1-\sqrt{5})}{2}\right]^n}{\sqrt{5}}$$

XXXI.15- Remarques sur Fib

Remarque-1 :

Sur la base de la définition de Fib(n), on devrait écrire

(1') $t_n = t_{n-1} + t_{n-2} + 1$ (+1 pour l'addition des 2 termes)
(2,3) $t_0 = 0,\ t_1 = 1$

- De (1'), on obtient $t_n - t_{n-1} - t_{n-2\ -1} = 0$.

- En posant $t_n = r^n$: $r^n - r^{n-1} - r^{n-2} = 0$

$$r^{n-2}\,(r^2 - r - 1 - 1/r^{n-2}) = 0$$ ***équation caractéristique***

Mais le terme $1/r^{n-2}$ est très négligeable.

Remarque-2 :

Finement calculé, Fib(n) demande en fait $2^{0.694n}$ additions.

--> Le calcul de Fib(200) requiert de l'ordre de 2^{140} additions.

Sur le super-calculateur Tianhe-2 capable de 33.86 pétaflops ($33.86 \times 10^{15} \approx 33.86 \times 2^{40} \approx 2^{45}$ opérations/sec.), cela prendra 2^{95} secondes > 126×10^{20} ans)

Tianhe-2 : U. nat. de technologie de la défense, Chine qui occupait (en 2018) la 4e place des super calculateurs dans le monde

Le supercalculateur Frontier (HP, USA) avec 1102 pétaflops mettra 2^{90} secondes

diminuera le temps total seulement d'un facteur de 2^5 (=32×10^{20}ans)

XXXI.16- Une autre solution à Fib

Une autre façon de calculer F(n) (merci à F. Pascual & O. Spanjaard, Sorbonne U.) :

On écrit $F_1=F_1$ et $F_2=F_0+F_1$ en matriciel :

$$\begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

où $F_0=0$ et $F_1=1$. De même,

$$\begin{pmatrix} F_2 \\ F_3 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2 \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

Et plus généralement

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

Calcul de F_n : élever la matrice à la puissance n.

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

On sait que pour une matrice M, le calcul de M^n nécessite **O(log n)** multiplications matricielles (besoin de **O(log n)** mises au carrée si $n=2^k$).

Ex. de calculs jsq M^3 : F(2) ... $F(2^3)=34$: $\begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}$ $\begin{pmatrix} 2 & 3 \\ 3 & 5 \end{pmatrix}$ $\begin{pmatrix} 13 & 21 \\ 21 & 34 \end{pmatrix}$

XXXI.16.1- Exercice : Coloration de graphes

But : Affecter une couleur à chaque nœud d'un graphe d'incompatibilité.

Minimiser le nombre de couleurs utilisées.

Contraintes : deux nœuds voisins ne doivent pas avoir la même couleur.

Combinatoire théorique (cas pire) : N nœuds et N couleurs --> N^N (classe NP)

--> Ici, 5 nœuds, un ensemble de K couleurs --> Combinatoire (théorique) : K^5.

N.B. : Applications de la coloration

Stratégies :

Méthode **Gloutonne** : 3 couleurs,

Méthode **Heuristique (et contraintes)** : 2 couleurs.

Calculer la complexité de chaque méthode

XXXI.16.2- Exercice 1 : Tris

- Étudier les algorithmes de tri les plus efficaces (O(N log N))
- Remarque importante sur le choix d'un algorithme de TRI

XXXI.16.3- Exercice 2 : autres complexités de Fib

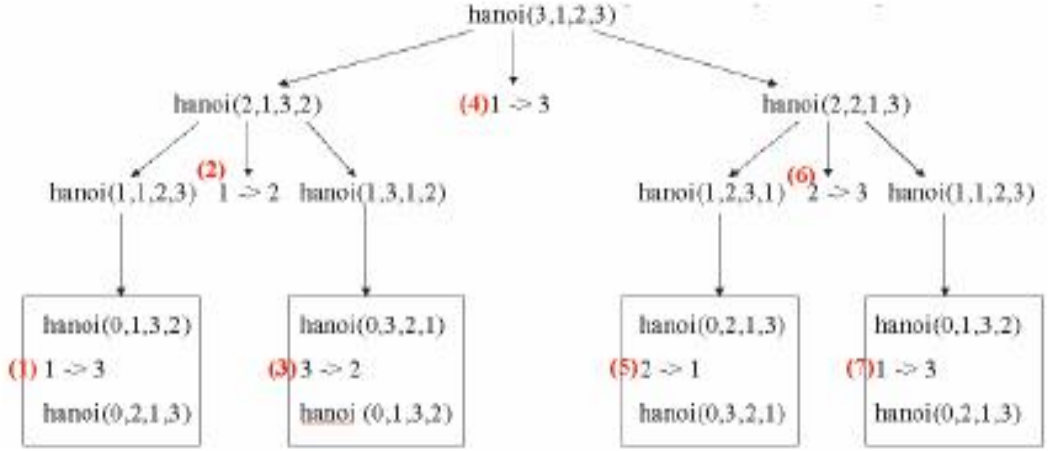
- Démontrer que

  - Fib(N) est de complexité $O(5/3)^N$
  - Fib(N) est $\omega(2^{N/2})$
  - Fib(N) : calcul de complexité par induction $T(N) > 2^{N/2}$

XXXI.17- Exemple de complexité Homogène : Hanoï

On a vu un calcul intuitif de la complexité de Hanoï.

Trace pour n=3



Détails (intuitifs) :

pour n=0 0 appels

pour n=1 2 appels

n=2 6

n=3 15

n=4 31

n=5 63 .....

l'*ordre de la fonction* Hanoï est **$O(2^n)$**, n>=1, … (voir plus loin)

- Si H_n = le temps pour exécuter la fonction (n>0 nombre de disques), on a :

  $H_n = 2\ H_{n-1} + C$ C : le coût du déplacement d'un disque

Posons $H_n = r^n$ $\rightarrow r^n - 2r^{n-1} - C = 0$.

--> d'où l'équation caractéristique $r^{n-1}(r-2- C/r^{n-1}) = 0$ à résoudre.

La quantité C/r^{n-1} (pour r^{n-1} non nul) négligeable → la racine r=2 (et r=0).

D'où $Hn = r^n = 2^n$ --> $T(N) = O(2^n)$.

- Plus exactement, $T(N) = c.\ 2^n$ et $T(1)=c.2^1=1 \rightarrow c=1/2$

→ Et donc : $T(N)=1/2.\ 2^n$ ✍ $T(N) = O(2^n)$.

Nota Bene :

1- Après quelques termes de Hanoi, on aura deviné une complexité :

l'*ordre de la fonction* Hanoï est **$O(2^n)$**, n>=1, …

On peut démontrer cette complexité par Induction.

2- Une (autre) analyse triviale des tours de Hanoi (cas particulier) :

Si le coût d'un déplacement = 1, l'approche descendante nous donne :

→ $T(n) = 2\ T(n-1) + 1$

→ $T(n) + 1 = 2\ T(n-1) + 1 + 1$

→ $[T(n) + 1] = 2\ [T(n-1) + 1]$

La multiplication par 2 apparaît clairement d'où la complexité $O(2^n)$.

XXXI.18- Exemples complexité non Homogène

XXXI.18.1- Exemple 1

$t_n = 5\ t_{n-1} + -6\ t_{n-2} + 3.\ 2^n$

$t_0=t_1=n+1$

- On obtient $(r2 - 5r + 6)(r-2)^{0+1} = 0$

  → $(r-3)(r-2)^2=0.$

- Les constantes seront calculées et c1=12, c2=-11 et c2=-6

  d'où $T(n) = 12.3^n -11 . 2^n -6 . n . 2^n.$

N.B. : on aurait pu conclure sur une complexité $\Theta(3^n)$ sans même calculer les constantes.

XXXI.18.2- Exemple 2

Soit la somme $f(n) = \sum_{j=0}^{n} j^2 = 1^2 + 2^2 + 3^2 + \ldots + n^2$

dont l'équation de récurrence est donnée par :

$t_0=0$, $t_1=1$, $t_2=5$, $t_3=14$

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

On est dans le cas d'une équation de récurrence non homogène : $t_n - t_{n-1} = n^2$

Résolution :

- la partie homogène : $t_n - t_{n-1} = 0$ → $r^n - r^{n-1} = 0$ et les racine sont $r=0$ et $r=1$.
- la partie **non homogène** à comparer avec $\mathbf{b^n\ p(n)}$ est $\mathbf{n^2}$

→ $n^2 = 1^1(n^2)$ d'où b=1, d=2 et $(r-b)^{d+1} = (r-1)^3$

On a donc les racines r=0 et r=1 de multiplicité 4 (3+1).

↵.

La complexité : $t_n = c_1 0^n + c_2 1^n + c_3 n.1^n + c_4 n^2.1^n + c_5 n^3.1^n$

→ $t_n = c_1 + c_2 n + c_3 n^2 + c_4 n^3$ (les constantes indicées à partir de 1)

Les conditions initiales permettent de trouver $c_1 = 0$, $c_2 = 1/6$, $c_3 = 1/2$, $c_4 = 1/3$

→ $t_n = n/6 + n^2/2 + n^3/3$

Rappel :

on sait par ailleurs que la somme des carrées $f(n) = 1/6(n.(n+1).(2n+1))$

XXXI.18.3- Exemple 3

Résoudre l'équation de récurrence suivante :

$$u_n = 3u_{n-1} - 4u_{n-3} + 3n$$

avec $u_0 = 27/4$, $u_1 = 81/4$, $u_2 = 34$.

Solution :

- La partie Homogène : l'équation linéaire homogène associée est

$$u_n = 3u_{n-1} - 4u_{n-3}$$

Et son polynôme caractéristique $r^3 - 3r^2 + 4 = (r+1)(r-2)^2$

Il admet une racine simple (r= -1) et r= 2 sera une racine double.

Les solutions de l'équation linéaire homogène associée sont donc de la forme

$$u_n = c1(-1)^n + 2^n(c2 + n.\ c3)$$

- La partie non homogène à comparer avec $\mathbf{b^n\ p(n)}$ est $\mathbf{n}$

  $\rightarrow$ $3n = 3^1(n)$ d'où b=3, d=1 et $(r-b)^{d+1} = (r-3)^2$

  On a donc les racines r=-1, r=2 et r=3 de multiplicité 2

- En reportant dans l'équation de récurrence, on obtient $\mathbf{c = 27/4}$.

L'ensemble des solutions de l'équation non homogène est donc de la forme

$$\mathbf{u_n = c1(-1)^n + 2^n(c2 + n.\ c3) + 3^n\ (c4+n.\ c5)}$$

- Compléter les calculs pour trouver les constates.

  Indication : de U_0, on obtient c4=27/4.

XXXI.18.4- Exemple 4

- Une banque propose une épargne rémunérée de la manière suivante :

- Si un client laisse une somme s sur son compte épargne durant une année complète, alors à la fin de l'année, le compte épargne du client est crédité de 10% d'intérêt, plus une somme fixe de 1000 euros.

- Soit u0, la somme déposée par un client à la fin de l'année A, et soit un l'argent disponible sur le compte épargne à la fin de l'année A+n en supposant qu'il n'a jamais retiré d'argent de son compte épargne.

- Déterminer et résoudre l'équation de récurrence vérifiée par la suite un .

Solution : l'équation de récurrence est $u_n = 1.1u_{n-1} + 1000$.

On a une *équation de récurrence linéaire non homogène d'ordre 1.*

- L'équation homogène associée admet pour solutions les suites de la forme

$$u_n = c1(1.1)^n$$

- L'équation non homogène admet une solution particulière constante **c2** qui vérifie

$$c2=1.1c2+1000, \quad \text{d'où} \quad c2=-10\,000$$

- L'ensemble des solutions est donc formé des suites de la forme

$$u_n = c1(1.1)^n - 10000$$

D'après la condition initiale, on a $c1 = u_0 + 10000$, et donc finalement

$$u_n = (u_0 + 10000) * (1.1)^n - 10000.$$

N.B. : comme pour Hanoï, on pourrait ignorer la quantité $(1000/r^n)$ et donc la constante 10 000 (-c2) disparaît de la solution.

XXXII- Table des matières

I- Quelques références bibliographiques 2
II- Objectifs du cours et éléments abordés 3
III- Pourquoi faut-il des algorithmes ? 5
IV- Qu'est-ce qu'un algorithme ? 6
IV.1- Exemple 2 :Tours de HANOI 7
IV.2- Exemple 3 : mariages stables (stable matching) 8
V- Ce que l'on peut attendre des algorithmes 9
VI- Propriétés 11
VII- Idée de la complexité 12
VIII- Complexité et Contrôle 14
VIII.1- Exemple 1 : Comparatif recherche d'une "vis" 16
VIII.2- Exemple 2 : séquence de Fibonacci 17
VIII.2.1- Comparatif Fibonacci 18
IX- Tableau des croissances relatives 20
IX.1- Comparaisons des courbes des croissances usuelles 21
IX.2- Comparaison de qq. ordres de de complexité 22
X- Éléments d'analyse de la complexité 23
XI- Sensibilité à la puissance des machines 25
XII- Déf. de big-oh (upper bound, pessimiste) 29
XII.1- Exemples de calcul de big-oh 31
XIII- La fonction Ω (lower bound : optimiste) 35
XIII.1- Exemples de calcul de Ω 36
XIV- La fonction *Θ* (égalité entre O et Ω) 37
XIV.1- Exemples de calcul de Θ 39
XV- *Illustration des 3 fonctions* O, Ω, Θ 41
XVI- Calculs : le Modèle (de la machine) 42

XVII- Règles basiques et empiriques de calcul 43
XVIII- A propos de la complexité moyenne A(n) 47
XIX- Complexité et Stratégies 49
XIX.1- Exemple 1 : calcul de la médiane d'une suite 50
XIX.1.1- Meilleure Solution : un stratégie Diviser pour régner 51
XIX.1.2- Détails du déroulement pour cet exemple 53
XIX.1.3- L'algorithme Aho & al. 54
XIX.2- Exemple 2 : séquence de somme maximale 56
XX- Propriétés des limites de fonctions 66
XX.1- Exemple trivial d'utilisation des limites 68
XX.2- Application : approximation de la complexité par programme 69
XX.3- Exemple 1 71
XX.3.1- Estimation empirique de la complexité de l'exemple 72
XX.4- Exemple 2 74
XXI- Outils de calcul de la complexité 77
XXI.1- Introduction : Équation de récurrence 77
XXI.2- Propriétés des solutions à l'équation de récurrence 79
XXII- Calcul de la complexité 81
XXII.1- Exemples de calcul de complexité 82
XXII.1.1- Calcul à partir des ordres de croissances (0,Ω,Θ) 82
XXIII- Cas simples : proposer et vérifier 86
XXIV- Preuve 90
XXIV.1- Exemple de preuve : Hanoi 91
XXIV.2- Exemple de preuve : recherche Dichotomique 92
XXIV.3- Preuve par Induction 93
XXIV.3.1- Introduction à l'induction 93
XXIV.4- Utilisation dans la complexité 95
XXIV.4.1- Exemple 1 95
XXIV.4.2- Exemple 2 98
XXIV.4.3- Exemple 3 (à démontrer) 99
XXIV.5- Vers des outils plus puissants 100
XXIV.5.1- Exemple 1 (cas défavorable) 100
XXIV.5.2- Exemple 2 (cas défavorable) : Fib 101
XXV- Résolution de l'équation caractéristique 102
XXV.1- Théorème 1 106

XXV.1.1- Exemple 1 107
XXV.1.2- Exemple 2 : une autre définition de Fib 108
XXVI- Cas de racines multiples : théorème-2 110
XXVI.1- Théorème 2 111
XXVI.1.1- Exemple 1 112
XXVI.1.2- Exemple 2 114
XXVII- Récurrence linéaire non homogène 115
XXVII.1- Théorème 3 116
XXVII.1.1- Exemple 1 117
XXVII.1.2- Exemple 2 119
XXVII.1.3- Exercice 1 121
XXVII.1.4- Exercice 2 122
XXVIII- Boite à outils (Cook Book) 123
XXVIII.1- Méthode principale (MM) et la famille Thêta 124
XXVIII.1.1- Exemples 126
XXVIII.2- Illustration de MM 127
XXVIII.3- *Méthode principale généralisée (MMG)* 128
XXVIII.3.1- Exemples 130
XXVIII.4- Exemples de calcul et Master Theorem 132
XXVIII.4.1- Algorithme mult 133
XXVIII.4.2- Mult2 : une modification de mult 135
XXVIII.5- Exercice : paire de points les plus proches 138
XXIX- Complexité : complément technique 144
XXIX.1- Changement de variable (cas homogène) 144
XXIX.1.1- Exemple : Recherche dichotomique 145
XXIX.1.2- Exemple : Tri Fusion 146
XXIX.2- Changement de variable dans une cas non homogène 148
XXIX.2.1- Exemple 149
XXX- Cas particulier : racines imaginaires ! 152
XXX.1- Exemple 1 152
XXX.2- Exemple 2 153
XXXI- Compléments 155
XXXI.1- Classe d'algorithmes NP 157
XXXI.2- Exemples de problèmes indécidables 158
XXXI.3- *Définition du Little-oh (et de little-oméga)* 160
XXXI.4- Remarque sur la Complexité et vitesse de calcul 162
XXXI.5- Idée intuitive: big-Oh et Oméga 163

XXXI.6- Algorithmique et les ressources 164
XXXI.7- *Complexité : 'bons' et 'mauvais' algorithmes* 165
XXXI.8- Résumé et remarques sur les ordres 168
XXXI.9- Complexité et Stratégies / Heuristiques dans les algorithmes 172
XXXI.9.1- Stratégies et complexité (ex. "Vis") 173
XXXI.9.2- Complexité et stratégies : le problème de N-reines 177
XXXI.9.3- Complexité et stratégies : égalité de 2 tableaux 180
XXXI.9.4- Complexité et stratégies : anagrammes 184
XXXI.9.5- Complexité et stratégies : tableau des moyennes cumulatives 187
XXXI.10- Mariages stables (stable matching) 190
XXXI.11- Algorithmes de mariage stable 193
XXXI.11.1- Solution par un matching (couplage) maximum 193
XXXI.11.2- Algorithme de mariage stable Chez MitHic 201
XXXI.12- *Pseudo-algorithme de calcul de la médiane d'une séquence* 203
XXXI.13- Fib : pourquoi les lapins remercient Fibonacci ? 205
XXXI.14- Pourquoi Fib(N) est Ω(3 /2)N et O(5/3)N 207
XXXI.15- Remarques sur Fib 209
XXXI.16- Une autre solution à Fib 211
XXXI.16.1- Exercice : Coloration de graphes 212
XXXI.16.2- Exercice 1 : Tris 213
XXXI.16.3- Exercice 2 : autres complexités de Fib 213
XXXI.17- Exemple de complexité Homogène : Hanoï 214
XXXI.18- Exemples complexité non Homogène 217
XXXI.18.1- Exemple 1 217
XXXI.18.2- Exemple 2 218
XXXI.18.3- Exemple 3 220
XXXI.18.4- Exemple 4 222
XXXII- Table des matières 224