Stratégies de Résolution de problèmes

ECL – 2A – MI

2017-2018

Chapitre 1 (transparents)

(Éléments de calcul de la complexité)

Alexandre Saidi (ECL-LIRIS)

Quelques références bibliographiques :

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

Objectifs du cours et éléments abordés :

En particulier : - ***rigueur*** et coût (***complexité***) des algorithmes :

" Votre programme marche " ? Prouvez-le !

- Connaissance de quelques méthodes importantes

Éléments abordés dans ce module :

- Analyse et Complexité des algorithmes, outil de mesure de complexité
- Différents outils et stratégies de résolution de problèmes :
  - Graphes et algorithmes remarquables, TAS & Arbres, Dijkstra, MST, etc
  - Récursivité (sa transformation vue en 1A)
  - Stratégies de résolution : AES, Programmation Dynamique, B&B, A*, etc.
- A travers l'Étude de quelques problèmes combinatoires connus.

I- Pourquoi un algorithme ?

Certains problèmes importants (p. ex. TSP, tournée de véhicules, …) n'ont pas de spécification (en Programmation) Mathématique.

→ Pas de formule pour calculer une sortie / décision / etc.

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

Ex. trivial *- trier une séquence d'entiers de taille N dans l'ordre asc.*

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

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

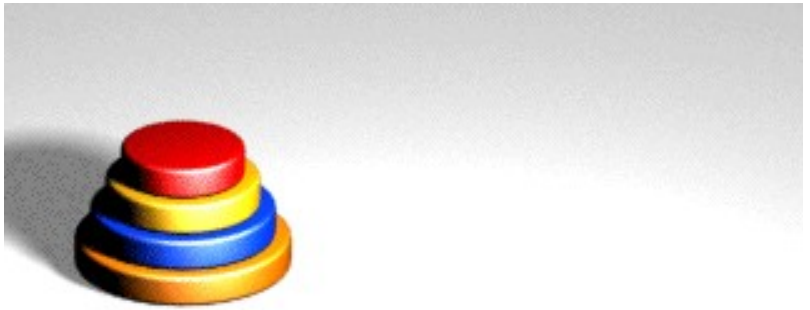
Un **algorithme** de TRI décrit la méthode sous forme d'une suite d'instructions précises exécutée pour S de taille N.

Parmi une pléthore de méthodes, on dira par exemple :

- *Commencer à partir du 1er élément de S (soit $S_{i=1,N-1}$),*
- *Le comparer aux éléments suivants dans S (soit $Sj_{=i+1, N}$);*
- *Permuter Si et Sj si Si > Sj.*

II.1- Exemples moins triviaux

II.1.1- Tours de HANOI



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

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

Appel initial avec le tuple ***<N, dép = A, arr = B, aux =C>*** *:*

- *Commencer par <N-1, dép = A, arr = C, aux =B>*
- *Placer le Nième disque sur B;*
- *Recommencer avec <N-1, dép = C, arr = B, aux =A>...*

II.2- Mariages stables

Un couplage est **instable** s'il contient deux personnes A et B non appareillées (non mariées) ensemble qui se préfèrent mutuellement à leur conjoints.

F est mariée avec g

G est mariée avec f

F préfère G à g

G préfère F à f

Questions :

- Commet vérifier qu'un couplage est stable ?
- Un couplage stable existe-il toujours ?
- Est-ce qu'on sait trouver un couplage stable s'il existe ?

Exemples :

Des femmes : Alice, Bénédicte, Camille

Des hommes : Elie, 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 ?

Applications :

- Répartition des biens rares (où les mécanismes habituels et simples ne fonctionnent pas).
- Affectation des candidats sur des places :
  - Elève vs. Ecole
  - Travailleur vs. Poste
  - Internes vs Services hôpitaux
  - Etudiants vs. Universités
  - Elèves 1A vs. Pes 1A (ou stages)
  - etc
- Dons d'organes (reins)
- Etc.

II.2.1- Mariages stables Chez MitHic (stable matching)

- Soit un ensemble de garçons $\{g_1, \ldots, gn\}$

  et un ensemble de filles $\{f_1, \ldots, f_n\}$ qui cherchent âme-sœurs.

- *MitHic* demande à chacun (e) d'établir une liste de préférences.

  But : mettre les âme-sœurs ensemble.

Notions : **instabilité** et **engagement**.

Les stratégies (algorithmes) que MitHic pourrait appliquer

- Algorithme fourre-tout qui se perd à vérifier 2 à 2 les compatibilités (stage ?)
- **Mieux** : un algorithme *à base de* graphe bipartie (qui *renforce 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 Tant que
Renvoyer l'ensemble S des couples engagés.

☞ L'algorithme ***appariement stable*** (*stable matching*) est à la base de la solution à plusieurs types de problèmes similaires.

II.2.2- Application : trouver un stage

- Les élèves/entreprises expriment des préférences (compétences / salaire /...)
- Hypothèse : il y a autant de stages que d'élèves.

Initialement, tous les **E** (élèves) et **S** (stages) sont libres
Tant que il y a un élève libre et non proposé à un stage
  Choisir un élève **E**
  Soit **S** le stage le plus préféré de **E** à qui **S** n'est pas encore proposé
  Si **S** est libre (non attribué) alors **<E, S>** établissent une convention
  Sinon : // **S** est actuellement engagé auprès de **E'**
    Si **S** préfère (correspond mieux à) **E'** à **E** Alors **E** reste libre
    Sinon : // **S** préfère **E** à **E'**
      <E,S> établissent une convention
      E' devient libre
Fin Tant que
Renvoyer l'ensemble des couples <E,S> objets d'une convention de stage.

- Variante : pour les PEs à ECL, il y a plusieurs élèves pour un PE.

III- Propriétés des algorithmes

- Un problème a une solution équationnelle et/ou algorithmique (ou pas).
- Un algorithme = une méthode de calcul = une suite d'instructions
- Un algorithme a un *domaine de définition* qui est :

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

- Un algorithme doit être **juste** (et **complet** si possible ou exigé)
- Un algorithme **juste** doit fonctionner sur toutes les **instances** d'un problème.

  ➥ Par Ex. : un algorithme de recherche d'un élément dans un tableau doit savoir répondre pour tout tableau et tout élément recherché (un *bug* courant).
- Au besoin, un algorithme ***complet*** doit trouver toutes les solutions justes.
- Un algorithme doit " finir " (terminaison). (+ existence+unicité+preuve+...)

III.1- Quelques remarques

☞ N.B. : ***Algorithme = Logique + Contrôle*** (une logique + une stratégie)

Programme = Algorithme + Structures de données

- Les algorithmes et les Structures de données sont dépendants
- Un algorithme décrit une solution dans un espace d'états (initiaux / finaux).
  - ☞ Un état final = solution & Algorithme = comment aller d'initial au final.
  - ☞ Toute recherche dans cet espace ne donne pas forcément un algorithme :
    - Même si l'analyse est juste : $(n+1)!=(n+1).n!$ $d'où$ $n!=\frac{(n+1)!}{n+1}$ (contrôle ?)
- Notion de *calculabilité* (traçabilité, en anglais : *tractable*)
  - ☞ Ex. : un programme pour détecter si un programme boucle ?
- Itératif / Récursif : indépendant de la complexité, transformable
  - ☞ Parfois, on a le choix récursif/itératif, parfois non (e.g. Hanoï, …).

IV- Idée de la complexité

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

  → Dans *Logique + Contrôle*, plusieurs *Contrôles* pour une même logique.

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

Un Exemple : 15 boîtes de vis de longueurs différentes 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 :

$$\sum_{k=1}^{15} \frac{1}{15} * k = \frac{1}{15} * \frac{16*15}{2} = 8$$ Consultations en moyenne.

1'-Variante : ranger les vis selon la fréquence des demandes. → tjs 8 en moyenne.

2- Naïve 'Las Vegas' : Naïve 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

$$\frac{1}{15}*1+\frac{14}{15}*\frac{1}{14}*2+\frac{14}{15}*\frac{13}{14}*\frac{1}{13}*3+\frac{14}{15}*\frac{13}{14}*\frac{12}{13}*\frac{1}{12}*4+\ldots=\frac{1}{15}*\frac{2}{15}*\frac{3}{15}*\ldots*\frac{15}{15}=\frac{1}{15}\sum_{i=1}^{15} i=8$$

➥ 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 (≈ log 15).

➥ En moyenne 3,26 comp. : supposons trouver à l'indice 15 (cas défavorable)

- trouver en 1e tentative au milieu (T8) : 1/15
- trouver en 2e : ne pas trouver en T8 et trouver en T12 : 14/15 * 1/7 = 2/15
- trouver en 3e : ne pas trouver en T8 ni en T12 et trouver en T14:

14/15 * 6/7 * 1/3 = 4/15 , etc.

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 présente.**

Sinon, la probabilité des cases devient 1/30 (voir plus loin)

V- Algorithmie et Questions de ressources

Algorithmique : l'étude de la conception et l'analyse des algorithmes.

→ Démarche (méthode) pour aboutir, d'un état donné (initial) à un état final en décrivant les transitions successives d'états (séquentiel vs. Parallèle).

Question des ressources disponibles :

- Calcule sur un ordinateur, un boulier, avec les doigts, une machine "FACIT", ...
- Les ressources en temps de calcul et en espace mémoire.
  - Espace : l'entité *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,
- **Les premiers algorithmes** : l'antiquité (Euclide, PGCD).

VI- Stratégie et estimation de complexité temporelle

VI.1- Exemple 1 : recherche d'un élément

- Recherche d'un élément dans une séquence ordonnée S :

→ on préfère une méthode de recherche binaire (dichotomique) à une recherche séquentielle. Pourquoi ?

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.

VI.2- Exemple 2 : Tours de Hanoï

Pour N disques, il y a 2^N mouvements justes (sans se tromper).

Hypothèse : un déplacement est une opération de base !

```
Procédure HANOI(N, Départ, Aux, Arrivée )
    Si ( N == 0 ) Fin;
    HANOI( N-1, Départ, Arrivée, Aux )
    Afficher : "On déplace le disque"  N   " du "   Départ      " à "   Arrivée
    HANOI( N-1,  Aux, Départ, Arrivée )
```

☞ Complexité d'ordre $O(2^n)$, n>=1, …

☞ Une solution itérative directe est très difficile à trouver.

VI.3- Choix d'algorithme : la séquence de Fibonacci

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

- **Choix d'algorithmes** :

1 - La solution <u>récursive</u> naïve (cf. la déf.) avec beaucoup de calculs redondants.

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

2- La solution utilisant un tableau pour stocker (PrD) : Le coût **T(n)** = **n+1**

3- Il existe une solution utilisant la définition récursive (sera O($\mathbf{2^n}$), v. + loin) :

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

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

Critères de choix d'un de ces algorithmes :

Tableau comparatif du **temps d'exécution** sur une machine qui calcule chaque terme en une nanoseconde (10^{-9} sec.) pour les deux premières méthodes :

n	$2^{n/2}$	*Temps Algorithme Linéaire (n+1 termes)*	*Temps Algorithme Récursif (naïf)*
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

☞ Remarque : on ne traitera pas la complexité en espace.

➥ Exemples où la capacité (espace) des machines est mise à mal :

fib, ***fact***, le test ***prime*** (2^{p-1} %p =1, pour p>2), ...

VI.3.1- Fibonacci et la reproduction des lapins

Soit au départ de notre élevage un couple de lapins.

Combien de couples de lapins aura-t-on 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

Soit **Fn** le nombre de couples de lapins au mois n. ../..

Donc

F(n) = nombre de couples au mois (n-1) + nombre de couples nés au mois n

= nombre de couples au mois (n-1) + nombre de couples productifs au mois (n-1)

= nombre de couples au mois (n-1) + nombre de couples nés au mois (n-2)

F(n) = F(n-1) + F(n-2)

VII- Qu'est-ce que la complexité

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, 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(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.**

- Certains algorithmes peuvent utiliser des conditions optimistes.

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

  ➥ On doit considérer quand-même le **cas pire** (e.g. si le tableau est déjà trié).

  N.B. : pour éviter le cas pire de Q-Sort, peut-on " brasser" le tableau avant ?

  Pratique : on fait appel à *HeapSort* si déséquilibre des partitions (voir chap 2).

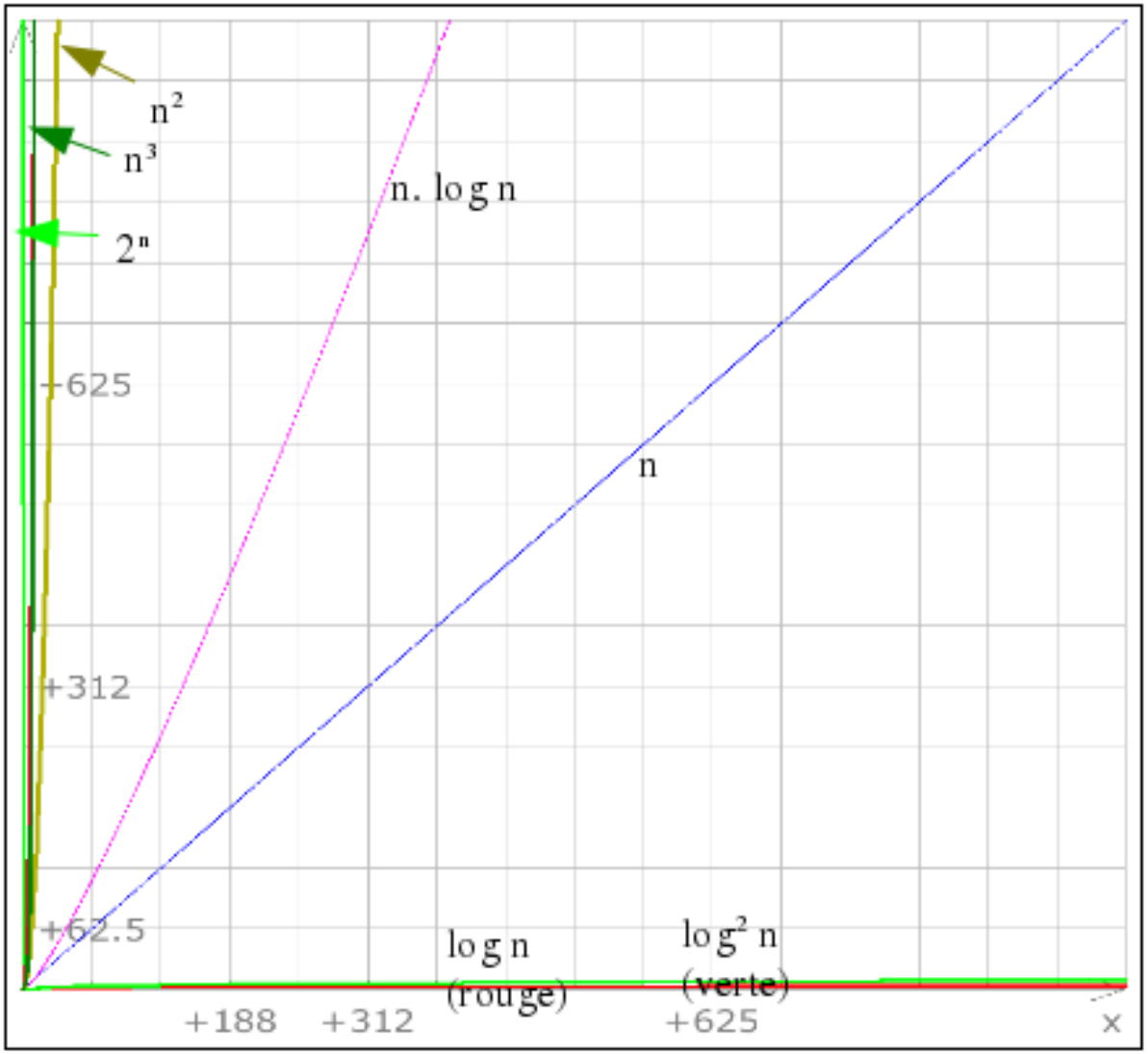
N.B. : accompagné de la complexité du cas pire, la complexité moyenne (≠ la moyenne des complexités) donne une indication intéressante.

VII.1- Tableau des croissances relatives

Dans l'ordre croissant 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 !)

VII.2- Comparaisons des courbes des croissances usuelles



VII.3- Éléments d'analyse de la complexité d'un algorithme

- Indique comment un algorithme se comporte et quelles ressources (temps et espace) il demande.
- Exprime : sous quelle forme le temps d'exécution augmente si la taille des données en entrée augmente.
- Doit être **indépendante** de tous les 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 :

  - 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 matrices* : $T(n) = n^3$

- 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*).

VII.4- Sensibilité à la puissance des machines

Principe d'invariance : malgré les différences technologiques, la complexité du même algorithme sur deux machines différentes ne varie que par un facteur constant.

Exemple : 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.

➥ De quel facteur (p/r à M1) peut-on augmenter la taille des 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**

  ➥ On peut augmenter seulement par 3 données !!

Constat : les progrès en puissance des machines sont négligeables face aux progrès en algorithmique (qui sont plus rares).

➥ Quand il y en a : c'est la révolution ! (e.g. Fourrier)

Résumé rapport Temps / Taille des données de l'exemple

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

Détails : la valeur de ε est négligeable devant la croissance de la fonction.

- dans la complexité **n log(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+ε].

- 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...

VII.5- 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 $\mathbf{n}^{100}$.

Remarque :

- une complexité polynomiale d'ordre **n** est meilleure si pas de terme d'ordre <**n**

  ➟ 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 $O(n^2)$.

- 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 donc construire de grands algos. polynomiaux à partir des petits.

VII.5.1- Classe d'algorithmes NP

On dit qu'un problème est **NP-complet** (par exemple, le *problème de voyageur de commerce*) 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.

Un exemple de problème NP (NP-hard) :

Carrelage d'une pièce : soit une surface rectangulaire à couvrir avec des petits carreaux (polygones) réguliers sans laisser de surface non couverte.

Question : y a-t-il un algo qui répond par oui/non à la faisabilité de ce problème ?

→ NON ! Il n'existe pas d'algorithme qui pourra répondre en un temps fini à la question pour toute forme de carreau (a été prouvé).

VII.5.2- Comparaison des complexités polynomiales et Exp-lle.

➟ 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 \ taille	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)** ➟ $O(5/3)^{100}$

Exemples de problèmes (combinatoires) : Sac à dos, Bin Packing, TSP, Flux, etc.

Plus simples : Cavalier, N-reines, SENDMOREY, Dijkstra, MST, Coloration, Chèvre et Chou, ...

VIII- Le Modèle (de la machine)

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

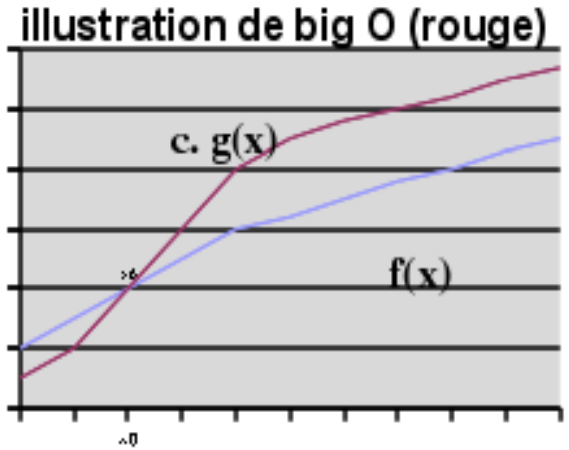
- **ordinateur normal** (séquentiel) avec des instructions simples (+,-,*,/).
- **chaque opération prend une unité de temps.**
- les entiers sont de taille fixe (32 / 64 bits)
- pas d'opération de base compliquée comme la multiplication ou inversion de matrices ou le tri (qui demandent > 1 unité). ➟ Elles ne sont pas (trop) basiques !
- 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.

IX- Définition 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)** ≤ **c.g(n)**, pour tout $n \geq n_0$ et **c** une constante.



Définition : soit f et g deux fonctions de ℝ dans 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 **∀x ≥ x0, f(x) ≤ c. g(x).**

➡ **g** devient plus 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 $\mathbb{N} \to \mathbb{R}$.

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

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

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

✍ 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. $n \geq n_0$).

IX.1- Exemples de big-oh

- **Recherche du maximum dans un tableau :**

  La recherche du max. d'un tableau peut prendre $(2n-1)$ opérations de base.

  ➟ On dira que $2n-1$ est $O(N)$ car on peut trouver une constante réelle $c>0$ et n_0 $(n_0 \geq 1)$ tels que $2n-1 < c.n$, 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.

- $20\ n^3 + 10\ n \log n + 5$ est **$O(n^3)$**

  ➟ car $20\ n^3 + 10\ n \log n + 5 < n^3$ pour $n \geq 1$

  ➟ Ici, $c=35$ et $n_0=1$

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

X- La fonction Oméga (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 **∀x ≥ 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.**

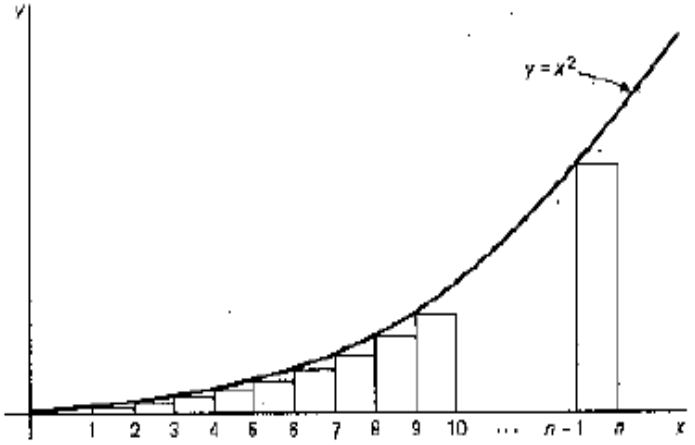
➡ Ω cherche à donner une meilleure estimation de la complexité.

X.1- Idée et exemple d'encadrement (par big-Oh et Oméga)

Exemple : soit à estimer la croissance de la somme $f(n)=\sum_{j=0}^{n} j^2=1^2+2^2+3^2+...+n^2$.

- La figure en face montre :

$$\sum_{j=1}^{n-1} j^2 \le \int_1^n x^2\,dx = \left[\frac{x^3}{3}\right]_1^n = \frac{(n^3-1)}{3}$$

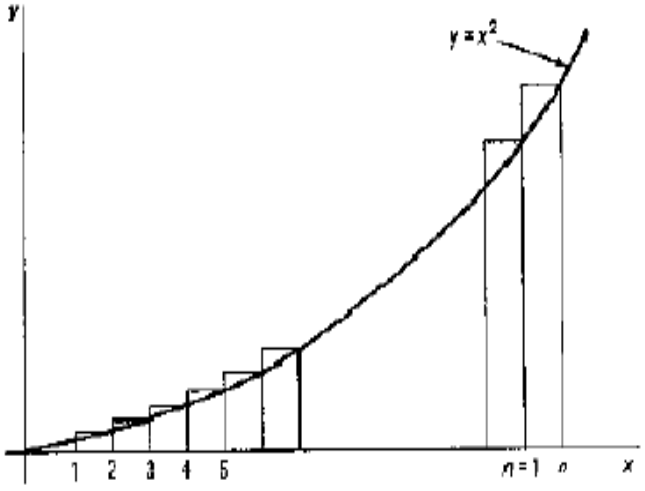


et donc (passer de n à n+1)

$$f(n) \le \int_1^{n+1} x^2\,dx \le \frac{((n+1)^3-1)}{3}$$

- De même

$$f(n)=1^2+2^2+3^2+...+n^2 \ge \int_0^n x^2\,dx = \frac{n^3}{3}$$



d'où $\int_0^{n+1} x^2\,dx \ge \sum_1^n i^2 \ge \int_0^n x^2\,dx$

X.2- Définition de Théta (égalité d'un ordre, Θ entre O et Ω)

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

L'ordre Θ permet de dire que deux fonctions sont asymptotiquement égales (modulo un facteur constant).

Définition : Pour une fonction de complexité f(x), $\Theta(f(x)) = O(f(x)) \cap \Omega(f(x))$.

Ce qui veut dire que $\Theta(f(x))$ 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 x_0 tels que :

$$\forall x \geq x_0, c.\, g(x) \leq f(x) \leq d.\, g(x).$$

Remarques :

- Si $f(n) = \Theta(g(n))$, alors $f(n)$ est à la fois $O(g(n))$ et $\Omega(g(n))$.
- si $f(x) \in \Theta(g(n))$, on dira que $g(x)$ est l'**ordre** de $f(x)$.
- Si l'on a $O(N)$ et $N = \Theta(x)$, alors on a $O(x)$.
- Si $g(n) \in \Theta(f(n))$ ssi $f(n) \in \Theta(g(n))$ (Θ porte l'égalité)

Exemples de Θ (Théta)

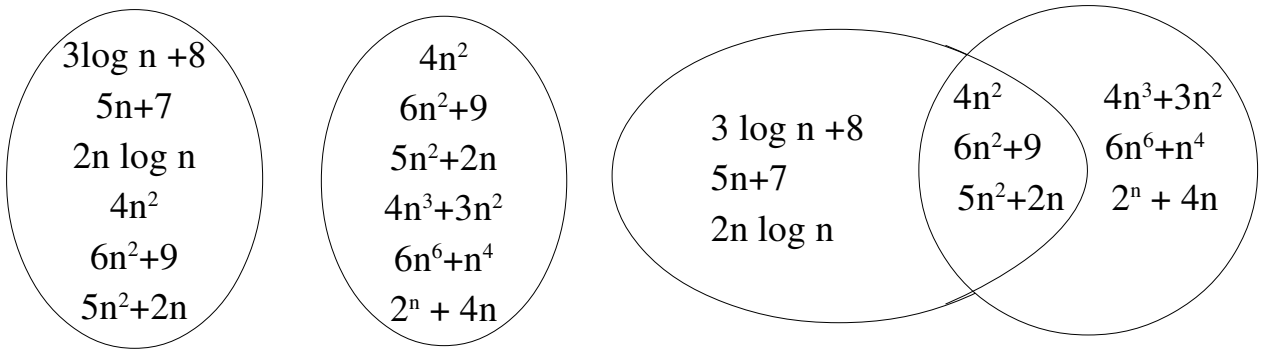
- $f(n) = n(n-1)/2$ est à la fois $O(n^2)$ et $\Omega(n^2)$ (cf. sections précédentes)
  - dans ce cas, $f(n) = \Theta(n^2)$
- $n^2 + 10n \in \Theta(n^2)$ car $n^2 + 10n \in \boldsymbol{O}(n^2)$ et $n^2 + 10n \in \Omega(n^2)$ pour n grand.

  On a également $n^2 \in \Theta(n^2 + 10n)$ par la propriété (3) ci-dessus.

X.3- Illustration des 3 fonctions **O**, **Ω**, **Θ**

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

Rappel : les termes d'ordre inférieurs ne sont pas significatifs dans les 3 fonctions.



(a) $O(n^2)$ (b) $\Omega(n^2)$ (c) $\Theta(n^2) = O(n^2) \cap \Omega(n^2)$

X.4- Définition du Little-oh (et symétriquement little-oméga)

- Dans les 3 fonctions O, Ω et Θ, on a l'ordre ≤ et ≥.
- On peut définir des relations avec < et > (**strictes**) en définissant **o** (little-oh) et ω (little-omeag).

  ➟ Certains logiciels Temps Réels exigent little-o.

Définition : soit f et g deux fonctions de ℝ dans 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 $\forall x \geq x_0, f(x) \leq c.\, g(x)$.

➟ Ici, contrairement à big-Oh, la relation se vérifie pour **toute constante c**.

➟ L'inégalité tiendra pour **tout réel c**, en particulier pour une petite constante c (e.g. C=0,00001).

- Avec O (big-oh), la constante $x_0 \geq 1$ alors que pour little-oh, on a $x_0 > 0$ et pour la constante c, le terme "*il existe c > 0*" de big-O devient "*pour tout c*".
- Idem pour ω (little-oméga) par rapport à Ω.

- N.B. : si on a **T(N)= o(f(N))**, on a **T(N) = O(f(N))** et **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)) - Ω(g(n))

C-à-d : f(n) est O(g(n)) mais pas dans Ω(g(n)).

X.4.1- Exemple de little-oh

- $f(n) = 12n^2 + 6n$ 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 =

Intérêt des "littles" : 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 !

XI- Règles basiques empiriques de calcul

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

XI.1- Règles basiques

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=0; i < n; i++)

        for (j=0; j < n; j++)      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
- Parfois, la fonction récursive est une itération « cachée »

  ➟ cas d'analyse simple.

Exemple O(N) :

```
long fact(int n)
  {  if (n <= 1) return 1;
     else     return  n * fact(n-1);
  }
```

Remarque : le calcul de complexité formelle des algorithmes récursifs est souvent plus simple !

XI.2- Quelques exemples de calculs simples

XI.2.1- Recherche d'un entier x dans T de taille N

```
// Recherche de x dans T[1 .. N]

   i=0;

   while (i < N && T[i] != x)  i = i+1;
```

- Les instructions à compter :

  le test **T[i] != x** est exécuté N+1 fois (au pire).

- On a une complexité en O(N).

- Les N incrémentations (O(N)) ne changent rien à la complexité.

XI.2.2- Somme des cubes d'un entier

Un programme qui calcule $\sum_1^N i^3$

```
int sum(int n)
{ int S=0;                          /* 1 */
 for (int i=1; i <= n; i++)         /* 2 */
      S = S + i*i*i;                /* 3 */
 return S;                          /* 4 */
}
```

Analyse :

La somme de tout ceci : 6.N + 4 ➟ O(N).

→ faites les détails !

XII- Et 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) = $\sum_{k=1}^{n} (k \cdot \frac{1}{n}) = \frac{1}{n} \cdot \sum_{k=1}^{n} k = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}$ 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 ϵ 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,

  Ce qui donne

La complexité moyenne (dépendra de p) :

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

→ si **p=1**, A(n) = (n+1)/2

→ si p= ½, A(n) =3n/4 + ¼ ➠ seul ¾ de S est recherché en moyenne

→ si p= 0, A(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 <u>taille/ valeurs</u> :

↳ telle valeur est plus fréquente que telle autre...

☞ **Le calcul de la complexité moyenne est souvent difficile (et donc rare car la proba. Pas toujours disponible) mais** donne des indications intéressantes.

XIII- Remarques sur tous les ordres

- Soit les catégories de complexité suivantes (avec k>j>2 et b>a>1) :

$\Theta(\log n) \nearrow \Theta(n) \nearrow \Theta(n \log n) \nearrow \Theta(n^2) \nearrow \Theta(n^j) \nearrow \Theta(n^k) \nearrow \Theta(a^n) \nearrow \Theta(b^n) \nearrow \Theta(n!)$

➟ 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 : $\log n \in o(n)$, $n^{10} \in o(2^n)$, $2^n \in o(n!)$

- 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 $f(n) \in \Theta(g(n))$.

- 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, **$a^n \in o(n!)$**

n! est pire que toute fonction de complexité exponentielle.

XIV- Efficacité et complexité : quelques exemples

Quelques exemples de complexité pour des algorithmes (simples et moins simples).

→ Bien remarquer que le choix d'un algorithme implique une complexité.

→ Idem pour le choix des structures de données.

XIV.1- Exemple : 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]

$$A[i] = \frac{\sum_{j=1}^{i} T[j]}{i}$$

XIV.1.1- Première version

Pour chaque élément de A, on calcule la moyenne (depuis T[1]) .

```
Fonction moyennes1(vector_d_entiers T)
  N=T.size();
  vector_d_entiers A(N);
  pour (i=0; i<N; i++)
    Somme_jsq_i=0
    pour (j=0; j<= i; j++)          – On recalcule la somme à chaque fois
      Somme += T[j]
    A[i] = Somme / (i+1);
 renvoyer  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 ?

XIV.1.2- Deuxième version

→ Conserver en permanence la somme des éléments T[0] .. T[i].

```
Fonction moyennes2(vector_d_entiers T)
   N=T.size();
   vector_d_entiers A(N);
   Somme_partielle=0
   pour (i=0; i<N; i++)
      Somme_partielle += T[i]
      A[i] = Somme / (i+1);

 renvoyer  A
```

- **Analyse de la deuxième version :**

  la complexité = $O(n)$. calcul trivial.

XIV.2- Exemple : 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)**

XIV.3- Exercice : égalité de 2 tableaux

1- **Égalité de 2 tableaux**

- Soit deux tableaux T1 et T2 d'entiers de taille N dont les éléments sont

  cas 1 : **uniques**

  → O(N. log(N)) si tri+comparaison, $O(N^2)$ si pas trié

  cas 2 : **non uniques** → Idem

  Vérifier que les deux tableaux contiennent les mêmes éléments.

  Une version pour le cas des éléments uniques est en annexes.

2- **Étudier** la complexité de l'algorithme **palindrome** dans ses différentes versions.

Itératif, récursif, à l'aide de l'inversion, ...

XIV.4- Exemple : calcul de la médiane d'une suite

Calculer la médiane M dans une suite d'entiers S de taille N (on note |S| = N) telle que :

La moitié des nombres de S soient plus petits que M et l'autre moitié plus grande.

Sol 1- Pour calculer la médiane, il suffit de 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 (voir + loin, min_heap, O(N)) puis retirer k éléments (k= $\lfloor$ N /2 $\rfloor$).

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

Sol 3- pour S de taille |S|, on peut trouver le $k^{ième}$ plus petit élément (ici k= $\lfloor$ |S| /2 $\rfloor$).

→ Le point fort de cette méthode est de ne trier que la plus petite sous séquence de S contenant le kième plus petit élément.

→ Voyons quelques détails ../..

XIV.4.1- Solution : un algorithme Diviser-régner

Soit la séquence S. On <u>généralise</u> pour k = 1..|S| et pour la médiane : k=1/2 |S|

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**

Exemple : S = <2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1> et **v** = 5 :

→ Sg=<2, 4, 1>, Sv = <5, 5>, Sd=<36, 21, 8, 13, 11, 20>

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

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

On notera : *selection(S,8) = selection(Sd,3) sachant |Sg| = 3 et |Sv|=2.*

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

On répétera ce traitement (récursif) sur le sous-tableau concerné jusqu'à arriver à un singleton qui est le résultat recherché (voir l'algorithme en Annexes).

☞ Le découpage déséquilibré des Sd et Sg peut donner une complexité défavorable

- V ∈ S sera choisi **aléatoirement**.

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

→ Voir [Aho & al][Wirth] pour 2 algorithmes de complexité moyenne O(n)

pour obtenir le kième plus petit élément de S.

Déroulement :

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

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

Il y a bien 5 éléments < 8 et 5 éléments >= 8

XIV.5- Un exemple de référence : séquence de somme maximum

- Soit une suite d'entiers *A1 … An* (peuvent 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 :

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

  → Pour **4, -3, 5, -2, -1, 2, 6, -2** ➟ La réponse = 11 (*A1..A7*)

- Ce problème a de multiples applications (e.g. Appariement *de sous-intervalles*)

  P. Ex : estimer le max de vraisemblance d'un motif dans une image (numérique).

- Il est intéressant car il y a plusieurs algorithmes (4 cités ici) possibles :

  ➟ **$O(N^3)$** à **$O(N)$**. (Voir les détails dans la version longue de ce support)

- Le tableau suivant donne une idée comparative des temps en **secondes** :

↓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 optimisation (aucun résultat/somme intermédiaire conservé)
- $O(n^2)$: optimisation dans les sommes partielles
- O(n log n) : approche dichotomique

la somme recherchée est soit dans la moitié gauche, soit droite, soit au milieu des 2.

- O(n) : approche récursive (petit inconvénient : il manquera les indices de l'intervalle)

Prenons le cas Dichotomique (O(n log n)) : Diviser-pour-régner

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

- Les 2 premiers cas (soit à gauche, soit à droite) ont une solution récursive directe ;
- La 3e peut être obtenue en trouvant **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 (continuité de la sous-séquence).
- Ces deux sommes pourront ensuite être additionnées (étape Régner).

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) et 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) et pour l'autre moitié, c'est 7 (A5..A7)

  ➠ **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 (B2) et celle de la 2e moitié = 13 (B4).
- 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).

➠ 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).

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

Pour le cas linéaire (en Python) :

```
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
```

→ 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.

A modifier (légèrement) 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 le meilleur somme max jusqu'à la position précédente, on est en présence de la PrD (voir le chapitre 2).

Notes sur les applications de max_sum :

* 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 du DNA).

→ On utilise également *min_sum* pour supprimer des séquences alignées de Protéine.

Dixit (un papier recherche en bio-info)

XV- Résumé des ordres les plus importantes

- 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 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 $\mathbf{g(N) = \Theta(N^2)}$, on dira que non seulement $g(N)=O(N^2)$ mais aussi que le résultat est **très proche de l'estimation**.

XVI- Complexité et vitesse de calcul : lent ou rapide ?

- 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. e^L ou $2^{\sqrt{L}}$) .
- 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. , *l'algo. A' garantie une réponse en au+* $0.57L^{23}$ *min.*

→ Aors A est plus rapide que A'

XVII- Calculs : 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 "**Hôpital**" :

si f et g sont dérivables et lim f(x) = lim g(x) = ∞

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

XVII.1- Exemples d'utilisation des limites

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

- **Pour a > 0, $a^n \in o(n!)$**

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

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

- Exemple d'utilisation de la règle Hôpital : on peut montrer que **$\log n \in o(n)$** :

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

XVIII- Calcul : approximation par programme

- La complexité empirique estimée peut être vérifiée par programme.

- **1- Une façon** : faire à la main des mesures et des calculs !

- **2- <u>Une autre façon</u>** de montrer qu'un algorithme est O(g(N)) est de calculer les valeurs de T(N)/f(N) pour un intervalle de N habituellement espacé par un **facteur de 2**, puis d'étudier la limite de T(N)/g(N) .

⇨ est une application de la limite.

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)**
- Si $\lim_{n \to \infty} \frac{f}{g}$ = + ∞, f est d'ordre supérieur à g.

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

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

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

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

  → Si **f(N)** est <u>sous estimée</u> **(**donc **erreur),** les valeurs **divergent** (tendent vers l'infini).

- Les taux de convergence et de la divergence signalent aussi le degré de justesse / erreur / efficacité.

XVIII.1- Exemple1

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

aille	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)

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

XVIII.2- Exemple 2

Montrer que la probabilité pour que deux entiers distincts et aléatoires I, J $\leq$ N soient premiers entre eux est approche $6/\Pi^2$ = .608 (pour N grand).

→ En estimer la complexité.

```
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
```

XVIII.2.1- Analyse de la complexité de l'exemple 2

- Deux boucles et pgcd ($O(\log N)$) dans la boucle interne → $N^2 \log N$ ➡ $O(N^2)$.

- 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 valeurs représentent les rapports $T(N)/g(N)$ et les constantes ('*c* ' de la définition de big-Oh) sont négligées.

- Chaque colonne représente **T(N) / g(N) et on étudie la limite**.

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 sont respectivement :

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

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

XIX- Analyse formelle de la complexité

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

  cf. les règles de calcul vues plus haut.

- Faire davantage attention aux erreurs dans l'analyse des fonctions récursives.

- Par exemple, dans le schéma :

```
fonction factorielle(n) =
  Si (n=1) alors   1
  Sinon    n*factorielle(n-1)
```

But : savoir combien de '*' ou combien de fois la fonction sera appelée (ou …)

→ T(n) = T(n-1) + 1

- Différentes méthodes pour trouver une réponse à notre question.

- On a besoin des *conditions initiales* (conditions d'arrêt).

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

1 → Deviner (faire une hypothèse) et vérifier ...

… avec une preuve simple pour les schémas non complexes via une :

- Substitution ascendante
- Substitution descendante

2 → Équation caractéristique

3 → Théorème ou Méthode Principale (ou MM : *master method*).

4 → Méthodes ad-hoc (Boîte à outils)

XIX.1- Deviner et vérifier : cas simples

Si on veut savoir la complexité pour une certaine valeur de n,

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

- Substitution **ascendante** :

  Exemple *factorielle* pour *n=5* et *condition initiale* : T(1)=2 (hypothétique)

  → T(1)=2 supposons cette valeur arbitraire !

  → T(2) = T(1) + 1 = 2+1 = 3

  ….

  → T(5) = T(4)+1 = 5+1=6 Semble être O(N)

- Substitution **descendante** :

  → T(5) = T(4)+1 = (T(3)+1)+1 = … = (T(1)+1)+1…+1=2+1+1+1+1=6 → O(N)

Conditions de résolution :

- Il faut (au moins) une condition initiale pour trouver une telle solution.
- T(n) est exprimée sous forme de **récurrence**.
- Pour "résoudre" une récurrence, on cherche une **forme fermée** (close).

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

- Par exemple :

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

et $T(1)=1$:

→ une forme fermée est $T(n)=n$.

- Forme fermée pour *Factorielle* (avec la 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+1

- De même avec la substitution a*scendante.*
- La vérification peut être triviale.

- Si pas de condition initiale, on aura une famille de formes closes.

  → Peu (voire Non-) exploitable.

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

  Pour l'exemple précédent :

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

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

- Ces techniques basiques sont utilisées pour les cas simples.

  Mais elles ne marchent pas pour tous les cas (ou sont fastidieuses)

  P. Ex. pour Fib(n) = Fib(n-1)+Fib(n-2) et Fib(1..2) = 1.

  → Si on fait une substitution, on entre-aperçoit un motif exponentiel (faux)

Une vérification de l'hypothèse (devinette) de complexité est nécessaire.

XIX.1.1- Exemple Hanoi

Action hanoi(entier n, pieux 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 $T(n)=2^n-1$.

XIX.1.2- Exemple de 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 (hyp.)** $T(n) = lg(n) + 1$ pour $n=2^k$ Est-ce vrai ?

- **Vérification** intuitive :

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

XIX.2- Méthode principale (MM) et Thêta

- Cook-Book des cas **fréquents** calculés par différentes techniques (v. + loin).
- Pour trouver la complexité Θ (O et Ω)
- On appelle une récurrence de la forme :

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

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

Exemple d'équations :

- $7t_n - 3t_{n-1} = 0$
- $6t_n - 5t_{n-1} + 8t_{n-2} = 0$
- $8t_n - 4t_{n-3} = 0$

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

Suivant l'algorithme étudié, on pioche dans le cook-book et 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 $a > b^k$

→ Remplacer "T(n)= " par "T(n) <=" ou "T(n) >=" pour avoir big-Oh ou Ω.

Exemples :

- $T(n) = 16\ T(n/4) + 5n^3$
  - $a=16, b=4, k=3$ 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 $a > b^k$
  - $T(n) = \Theta(n^{lg\_b\ (a)}) = \Theta(n^{lg(8)}) = \Theta(n^3)$

XIX.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)
  - $T(1) = d$
  - $b >= 2$ une constante entière et $a > 0$, $d >= 0$ des constantes

Suivant l'algorithme étudié, 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 $n \to \infty$

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 $n^{lg\_b\ (a+\varepsilon)} = O(f(n))$.

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

Exemples :

Proposer la récurrence T(n), identifier f(n) et extraire la complexité.

- $T(n) = 2\ T(n/2) + n$
  - Chercher les cond. de la récurrence : a = 2, b=2, f(n) = n, lg_b (a) = lg(2) = 1

    → **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 MM généralisée (I'..).
- P. ex. : Si $T(n) = a\,T(n/b) + cn^k$ satisfait la condition de (II) : $a=b^k$

  Alors on peut écrire :

  $a=b^k$

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

  → $f(n) = cn^k = cn^{lg\_b(a)} \in \Theta(n^{lg\_b\,(a)})$ cf. la condition du cas général (II')

- De même pour les deux autres cas.

- D'où viennent ces *recettes* ? Comment faire si aucune recette ne colle ?

  → **Analyse par l'équation de récurrence. ../..**

XIX.4- Analyse par équation de récurrence

- Vers une formalisation ….
- Solutions d'une large classe de récurrence (pour algorithmes récursifs).

XIX.4.1- Techniques d'exploitation de l'équation de récurrence

- Il n'y a pas de technique simple pour toute équation de récurrence
- Le cook-book (et le Master Method) ne couvre pas tous les cas.
- Dans les cas simples, utiliser la substitution (asc, desc) pour deviner l'hypothèse (une forme close) et utiliser **l'induction** pour la prouver.
- Plus généralement : résolution de l'équation linéaire homogène (ou non)

  → D'où vient le cook-book ?

On aborde en premier l'approche par récurrence + Induction

XIX.5- Approche par récurrence

XIX.5.1- Exemple 1

```
fonction fact ( n) =
  Si (n == 0) renvoyer 1
  renvoyer n * fact (n - 1)
```

Rappel : si t_n = nombre de multiplications nécessaires pour un *n* donné → $t_n = t_{n-1} + 1$

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

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

Remarque : dans la notation, t_n remplace T(n).

Une récurrence possède une **condition initiale** (ou ***condition d'arrêt***, pour une *forme close*)

➥ Ici, $t_0 = 0$ (aucune multiplication n'est faite pour n=0)

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

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

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

Pour le problème ci-dessus, $\mathbf{t_n=n}$ est bien une solution (?)

Vérifions la par Induction

Base de l'induction : $t_0=0$

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

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

$t_{n+1} = t_{(n+1)-1} + 1 = t_n + 1 = n+1$ CQFD.

Rappel : une induction ne peut pas trouver une solution mais peut vérifier si une solution proposée en est une (l'induction constructive peut, aider à trouver une solution).

XIX.5.2- 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) : $t_n = \log n + 1$

Base : $t_1 = 1$

Hypothèse : $t_n = \log n + 1$

Étape : puisque *n* est une puissance de 2, la prochaine valeur pour *n* est *2n* :

➥ $t_{2n} = \log (2n) + 1$

➥ $t_{2n} = t_{(2n)/2} + 1 = t_n + 1 = \log n + 1 + 1 = \log n + \mathbf{\log\ 2} + 1 = \log (2n) + 1$ CQFD.

XIX.5.3- 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 :

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

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

On constate une solution : $t_n = 7^{\log n}$ à démontrer par induction Math.

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

➥ $t_n = n^{\log 7} \simeq n^{2.81}$

XIX.5.4- Exemple 4 (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.

➥ On ne peut donc pas vérifier celle-ci puisque l'induction sert à vérifier si une solution candidate est juste.

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

XIX.5.5- Un autre 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 bloqué !

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

XIX.6- Équation de récurrence linéaire Homogène

XIX.6.1- Exemple de la séquence Fibonacci

Définie par :

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

$t_0 = 0$

$t_1 = 1$

On peut écrire la première équation par :

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

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

Comment résoudre ce type d'équation....(voir après des ex. + simples) ../..

XIX.7- Résolution de l'équation caractéristique

Supposons l'équation suivante :

$t_n - 5t_{n-1} + 6t_{n-2} = 0$ *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 $\mathbf{r^n - 5r^{n-1} + 6r^{n-2} = r^{n-2}(r^2 - 5r + 6)}$

➥ les racines : **r=0** et celles de $\mathbf{r^2 - 5r + 6 = 0}$

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

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) :

$$t_n = c_1\ 3^n + c_2\ 2^n$$

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

c_1 et c_2 sont des constantes arbitraires

N.B. : On peut démontrer que ce sont <u>les seules</u> solutions.

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

➥ Ce choix est déterminé par les <u>conditions initiales</u> :

$t_0=0$ $t_1=1$. D'où

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

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

Qui 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 …/..

XIX.8- 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.

XIX.8.1- Exemple

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

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 $\mathbf{t_n = 1/5\ 4^n - 1/5\ (-1)^n}$

XIX.8.2- Un autre exemple

Soit $U_n = U_{n-1} - 2\,U_{n-2}$

$U_0 = \ldots$

On obtient (montrer avec 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 $$r_1, r_2=\frac{1\pm i\sqrt{7}}{2}$$

XIX.8.3- Fib(N) revisitée : Fib(N) est $\Omega(3/2)^N$ 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 $\frac{(1-\sqrt{5})}{2}$

d'où $3/2 < r = \frac{(1+\sqrt{5})}{2} < 5/3$.

- Ce qui donnera <u>une idée</u> des valeurs 3/2 et 5/3 (approx. de $\frac{(1+\sqrt{5})}{2}$).

N.B. : le nombre $\frac{(1+\sqrt{5})}{2} \cong 1{,}6180339887$ est le **nombre d'or**.

Ce nombre est le rapport entre le terme Fib d'indice (n+1) sur 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) :

$$\frac{\left[\frac{(1+\sqrt{5})}{2}\right]^n - \left[\frac{(1-\sqrt{5})}{2}\right]^n}{\sqrt{5}}$$

XIX.8.4- Cas de Fib récursive

- 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. Pour un nombre impair (2e cas), il s'agit d'un simple terme supplémentaire. Le 1er cas est donc représentatif :

→ $T_{2n} = (T_n)^2 + 2T_n.T_{n-1}$ → Le coût de $(T_n)^2$ est simplement le cout de T_n+C.

→ $T_{2n} = T_n + C + 2T_n.T_{n-1}$ → $r^{2n} = r^n + C + 2r^n\, r^{n-1}$

→ $r^n . r^n = r^n + C + 2r^n\, r^{n-1}$ → $r^n . r^n - r^n - C - 2r^n\, r^{n-1} = 0$

→ $r^n (r^n - C/r^n - 2\, r^{n-1}) = 0$ → $r^n - C/r^n - 2\, r^{n-1} = 0$ et r = 0 est une racine

→ $r^{n-1} (r - C/r^{2n-1} - 2) = 0$ → r=0 (on le sait déjà), la quantité C/r^{2n-1} est <u>négligeable</u>

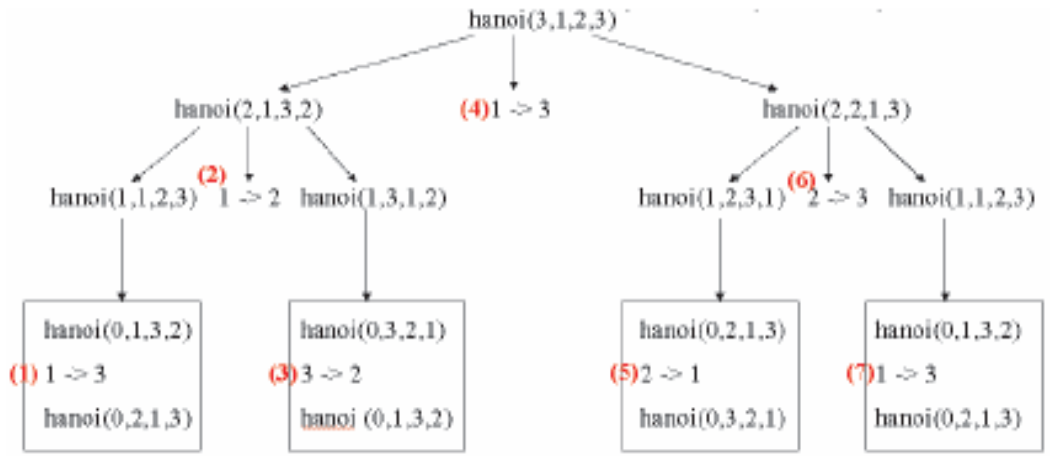
→ r - 2 = 0 → r = 2

On a donc une complexité $O(2^N)$. Comparer r=2 à 3/2 < r < 5/3 de ci-dessus.

XIX.8.5- Complexité de 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, ...

On peut le démontrer par Induction.

→ Voyons une approche moins intuitive : ../..

- 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 : $T(N)=1/2.\ 2^n$ ➟ $T(N) = O(2^n)$.

Remarque : 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 $O(2^n)$.

XIX.9- 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 $(r - 1)(r - 2)^3 = 0$

sachant que 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.

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 inclu dans la solution générale de la récurrence.

XIX.9.1- Exemple 1

Soit la récurrence :

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

$t_0 = 0$

$t_1 = 1$

$t_2 = 2$

- équation caractéristique **$r^3 - 7\,r^2 + 15r - 9 = 0$**

→ $r^3 - 7\,r^2 + 15r - 9 = (r-1)(r-3)^2 = 0$ où la racine 3 est de multiplicité 2.

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

➥ On introduit des termes **3^n** et **$n\,3^n$** car la racine 3 est de multiplicité 2.

- Les conditions initiales (à l'aide de t_0, t_1 et t_2) donnent $c_1 = -1$, $c_2 = 1$ et $c_3 = -1/3$

d'où **$t_n = (-1)\,1^n + (1)\,3^n + (-1/3)\,n\,3^n = -1 + 3n - n3^{n-1}$**

XIX.9.2- Exemple 2

Soit la récurrence :

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

$$t_0 = 0$$

$$t_1 = 2$$

$$t_2 = 3$$

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 $c_3 = 1$

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

$\mathbf{= 1 + n}$

XIX.10- Récurrence linéaire non homogène : théorème 3

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

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

où a_i 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 :

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

Deux exemples :

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

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

XIX.11- 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 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.

XIX.11.1- Exemple 1

Soit

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

$t_0 = 0$

$t_1 = 12$

Démarche :

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

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

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

$4^n\,(2n^1+1)$ où $b = 4$ et $d = 1$ (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 : $t_n = c_1 3^n + c_2 4^n + c_3 n4^n$.

- Calcul des constantes : ici, on n'a <u>que deux cas basiques</u> (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 : $t_n = 20(3^n) - 20(4^n) + 8\ n\ 4^n$.

XIX.11.2- 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* $a_0 t_n + a_1\ t_{n-1} + \ldots + a_k\ t_{n-k} = b^n\ p(n)$

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

XIX.11.3- Exemple 3

$t_n = 5\ t_{n-1} + -6\ t_{n-2} + 3.\ 2^n$

$t_0=t_1=n+1$

- 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.

XIX.11.4- Exemple 4

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 :

$t_0=0, \quad t_1=1, \quad t_2=5, \quad t_3=14$

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

On est dans le cas d'une équation de récurrence non homogène car $t_n - t_{n-1} = n^2$

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). ../..

D'où $t_n = c_1 0^n + c_2 1^n + c_3 n.1^n + c_4 n^2.1^n + c_5 n^3.1^n$

→ $t_n = c_1 + c_2 n + c_3 n^2 + c_4 n^3$ (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 $f(n) = 1/6(n.(n+1).(2n+1))$

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 $\mathbf{b^n}$ **Car** si P(n) est constante, alors n=0.

Car sinon, on introduirait une racine supplémentaire r=b → incohérent !

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

XIX.11.5- Exemple 6

Résoudre l'équation de récurrence suivante :

$$\mathbf{u_n = 3u_{n-1} - 4u_{n-3} + 3n}$$

avec $u_0 = 27/4$, $u_1 = 81/4$, $u_2 = 34$.

Solution :

- La partie Homogène : l'équation linéaire homogène associée est

$$\mathbf{u_n = 3u_{n-1} - 4u_{n-3}}$$

Et son polynôme caractéristique $\mathbf{r^3 - 3r^2 + 4} = (r+1)(r-2)^2$

Il admet (-1) comme racine simple et 2 comme racine double.

Les solutions de l'équation linéaire homogène associée sont donc de la forme

$$\mathbf{u_n = c1(-1)^n + 2^n(c2 + n.\ c3)}$$

- La partie non homogène à comparer avec $\mathbf{b^n\ p(n)}$ est $\mathbf{n}$

  → $3n = 3^1(n)$ 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.

  A compléter.

XIX.11.6- Exemple 7

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 u_0, 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 u_n .

Solution : l'équation de récurrence est $u_n = 1.1u_{n-1} + 1000$.

- *C'est 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.

XIX.11.7- Exercice

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 :

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

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

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

- Calculer les détails cette complexité.
- Indication : le résultat devrait être $t_n = c_1 + c_2\, n + c_3\, n^2 + c_4\, n^3 + c_5\, n^4$

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

XIX.11.8- Exercice

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 $(r-3)^{1+1}$.

XX- Changement de variable (transformation de domaine)

Exemple : pour un cas dichotomique tel que :

$T(n) = T(n/2) + 1$	$n>1$ et n = une puissance de 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 : **$T(n) = 1 + \log(n)$**

N.B. : une des recettes vues plus haut s'applique.

N.B. : un autre ex. de changement de variable était donné plus haut (Hanoi).

XX.1- Complexité du 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 k dans N.
- D'où l'équation de récurrence $t_k= 2t_{k-1} + 2^k$ → l'éq. caractéristique $r^k = 2\ r^{k-1} + 2^k$.
- En accord avec le théorème 3, on aura $(r^k - 2\ r^{k-1})(r- 2)=0$.

→ qui donne (r-2)(r-2) : une racine double r=2.

→ D'où la forme générale de la solution $t_k= c0+ c1\ 2^k+ c2\ k\ 2^k$.

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

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

→ D'où $t(n) = 1/2\ n \log(n)$

N.B. : une autre technique ad-hoc :

On peut utiliser une soustraction pour aboutir aux mêmes résultats :

- On avait (de la solution précédente) :

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

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

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

On soustrait (1) et (3) : $t_k = 4t_{k-1} - 4t_{k-2}$.

$r^k - 4\, r^{k-1} + 4\, r^{k-2} = 0$

→ $r^{k-2} (r^2 - 4 r + 4) = 0$ et $(r^2 - 4 r + 4) = (r-2)(r-2)$.

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.

XX.1.1- Changement de variable et 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 : ../..

Exemple : $t_n = 3\ t_{n/2} + n\ lg(n)$ pour $n>1$ et $t_1=2$

On pose $S(n) = T(2^k)$ avec $n=2^k \rightarrow k=lg(n)$ et donc

$s_k = t_{2^k} = 3\ t_{2^k/2} + 2^k . lg(2^k) = 3\ t_{2^{k-1}} + 2^k . k = 3\ s_{k-1} + k.2^k$

qui donnera $(r-3)(r-2)^{1+1}=0$ et $s_k = c1\ 3^k + c2 . 2^k + c3\ k. 2^k$.

- 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 :

$S(k) = 8. 3^k -6 . 2^k -2. k. 2^k$.

../..

- Pour obtenir T(n) qui était l'objectif, sachant que S(n) = $T(2^k)$, et k=lg(n), on a :

$$T(n) = S(lg(n)) = 8.\ 3^{lg(n)} - 6\ .\ 2^{lg(n)} - 2.\ lg(n).\ 2^{lg(n)}.$$

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 $T(n)=8n^{lg(3)} - 6n - 2nlg(n)$

  → Sachant que $1 < lg(3) < 2$ (car $2^1 < 3 < 2^2$) alors on a une complexité $O(n^2)$ pour $n=2^k$.

N.B. : une règle appelée la *règle de lissage* permet d'affirmer que si $T(n) = \Theta(f(n))$ pour **n** une puissance de b >= 2 et si f(n) est polynomiale (f(n) est dite *nice si* f(n) n'est pas *exponentielle* ni *factorielle*) alors T(n) est réellement $\Theta(f(n))$ pour toute valeur de n (ici > 1, voir le système initial).

XX.2- Cas particulier : Racines imaginaires !

- Il arrive (rarement) que l'on ait besoin de racines imaginaires.
- Exemple : 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 et en particulier, on a :

  $T(4n)=0$ car $(1-i)^4=(1+i)^4 = -4$
- Bien que la présence de racine imaginaire laisse supposer un terme (ici la complexité) négatif, il se peut que T(n) > 0.

  Par exemple, 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.

XXI- Annexes

XXI.1- Exemple de complexité : égalité de 2 tableaux

- Soit deux tableaux T1 et T2 d'entiers de taille N dont les éléments sont **uniques**.

  Vérifier que les deux tableaux contiennent les mêmes éléments.

XXI.1.1- Première version

- 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 $\Omega(n^2)$**

XXI.1.2- Deuxième version

- 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.

XXI.2- 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 5^{e} 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.**

XXI.3- Avoir (vous-mêmes)

- Calcul de complexité de quelques algorithmes de Tri
- Remarque importante sur le choix d'un algorithme de TRI

- Démontrer que Fib(N) est de complexité $O(5/3)^N$
  - Fib(N) est $\omega(\ \mathbf{2^{N/2}}\ )$
  - Fib(N) : calcul de complexité par induction $\mathbf{T(N) > 2^{N/2}}$

- Les méthodes de preuve
- Induction Mathématique.

XXI.4- Stratégies et Heuristiques dans les algorithmes combinatoires

Rappel : les structures de données peuvent jouer un rôle important.

- **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 2** : rechercher un mot dans un dictionnaire ?

XXI.4.1- Exemple : le problème de N-reines

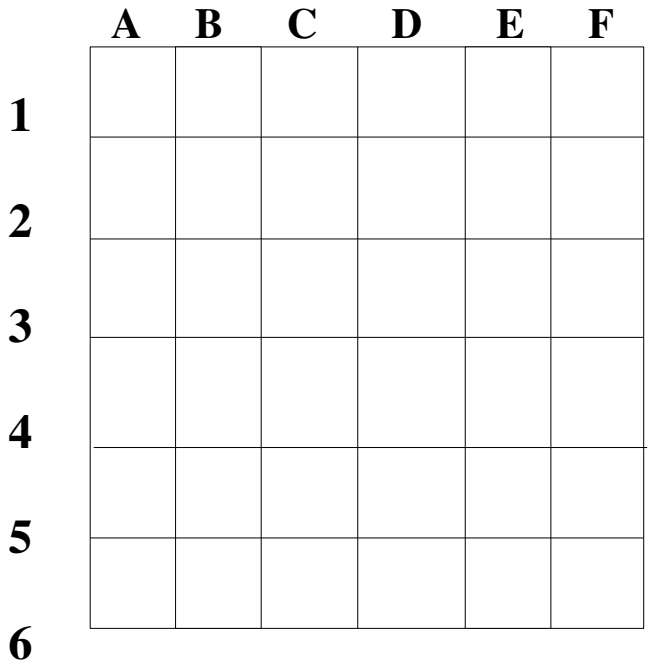
➟ Problème général d'**affectation sous contraintes.**

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.

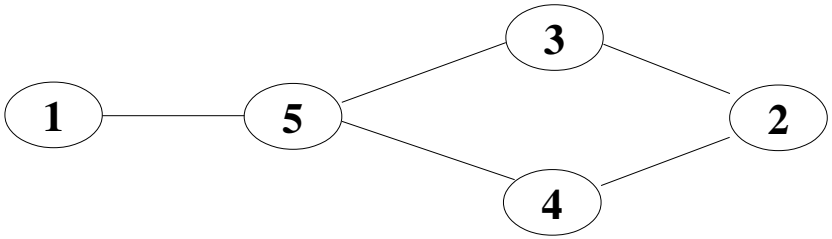
➟ **Combinatoire** : N^N possibilités

3. **Centrée Contraintes** : vérifier en permanence la validité d'un ensemble de contraintes ➟ **Complexité** théorique : N^N, mais méthode très efficace.

XXI.4.2- Notion d'heuristique : exemple de 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.

➟ 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.

Table des matières