

IG4 - Polytech Montpellier

Vincent Berry - Jérôme Fortin

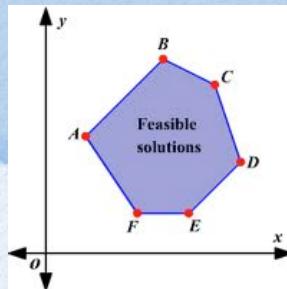
COMPLEXITÉ - IG4

Séance 1

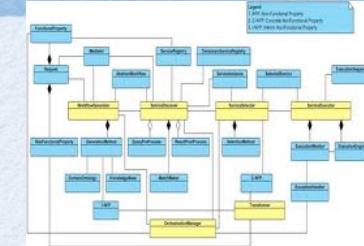
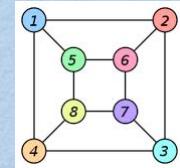
Evaluation de la complexité



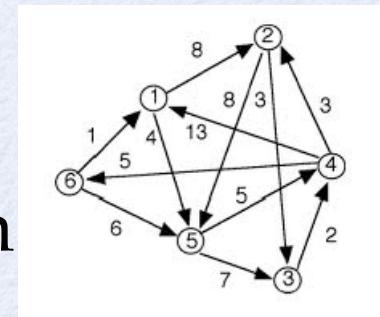
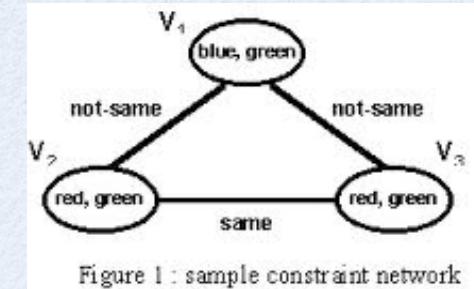
MOTIVATION



$$\begin{cases} \max Z = x_1 + x_2 \\ \text{s.a } 2x_1 + 3x_2 \geq 6 \\ \quad 2x_1 + x_2 \leq 10 \\ \quad x_2 \leq 4 \\ \quad x \geq 0 \end{cases}$$



- Un ingénieur doit avoir une **boîte à outils méthodologiques** pour pouvoir :
 - comprendre quel est le problème
 - vers quel type de solutions se tourner
 - savoir si un prototype/implémentation passera à l'échelle (anticiper)



MODALITÉ CONTRÔLE DE CONNAISSANCES

● Partiel en fin de module

- Si notePartiel > 10 : C'est parfait.
- Si note <10 : nous regarderons la participation en cours (grâce à la plateforme NearPod) et TP pour éventuellement remonter la note (jusqu'à 10 maximum).
- Si UE « optimisation et méthode mathématiques pour l'entreprise non acquise : Examen de rattrapage (Ne changera pas la note de l'UE, mais permet d'acquérir l'UE par décision du Jury)

NEARPOD

- Nous permet de suivre votre progression
- Nous permettra de rattraper certaines notes de partiel
- => Il est important de répondre soumettre vos réponse dans le temps demandé par votre enseignant.
- Pour les réponses « à dessiner », vous pouvez dessiner dans n'importe quelle application et envoyer une copie d'écran via NearPod
- Vous pouvez garder trace de vos réponses des corrections en faisant des copies d'écran de votre Ipad (par exemple juste avant de soumettre un réponse)

OBJECTIFS DE CE COURS

- En quelques séances
- Savoir évaluer (consulting)
 - la complexité d'une solution
 - la difficulté d'un problème (variantes de problèmes combinatoires récurrents)
- Savoir réagir (engineering) : arsenal de techniques de résolution
- Savoir se documenter : derrière les problèmes rencontrés par les ingénieurs en pratique se cachent souvent des problèmes déjà bien étudiés :
 - les reconnaître (réductions)
 - trouver des informations sur la façon de les résoudre (documentation, veille technologique : culture à former)

QUELQUES EXEMPLES TIRÉS DE STAGES ET PROJETS INDUSTRIELS



● 1 - Salon mettant en rapport entreprises et étudiants demandeurs d'emploi :

- candidat -> entreprises qu'il veut voir
- entreprise -> créneaux disponibles
- PBM à résoudre = préparer un planning affectant des rendez-vous (candidat,entreprise)
- Questions = peut-on satisfaire tout le monde ? Comment satisfaire un maximum de demandes ?
- Contrainte : l'algorithme doit tourner vite (mise à jour en temps réel)

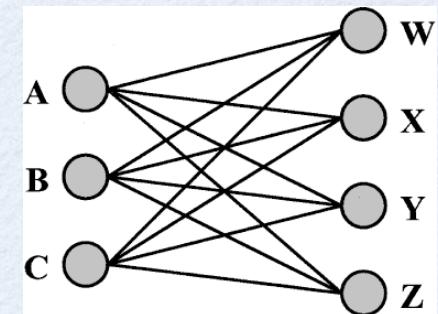
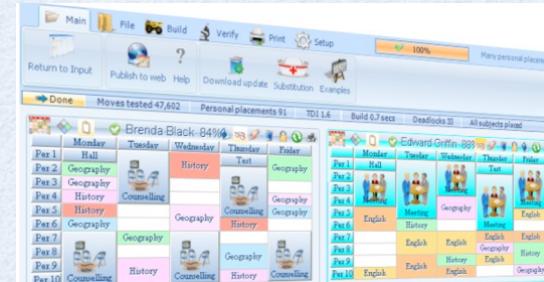
QUELQUES EXEMPLES

TIRÉS DE STAGES ET PROJETS INDUSTRIELS (POLYTECH MTP)

- 1 - Salon mettant en rapport entreprises demandeurs d'emploi

- Questions naturelles :

- quelle modélisation ?
- le problème est-il connu ?
- quelles solutions sont possibles ?
- quel compromis facilité / temps calcul / optimalité ?



sol° pour un créneau

QUELQUES EXEMPLES

TIRÉS DE STAGES ET PROJETS INDUSTRIELS (POLYTECH MTP)

• 2 - Dictionnaire stocké dans un SGBD questionné via un serveur web

- **Données** : deux langues L1 et L2.
- **Question 1** : un mot de L1 -> { mots de L2 }
- **Question 2** : question inverse : mot L2 -> { mots de L1}
 - pas de réponse depuis BD initiale après 1h.
 - Que faut-il faire ? -> changer la structure de la BD

• Paramètres à prendre en compte !

- coût de chaque connexion au **SGBD**,
- coût de chaque requête sur L1 (requête d'interrogation)
- coût re-calculation des indexées à chaque req. modification de L2
- coût d'interrogation BD -> **structure de données ad-hoc**
(arbre suffixe) sur le serveur.
- Le logiciel doit **prendre en compte** les aspects acquisition/production/accès aux données quand les tailles ne sont pas négligeables



QUELQUES EXEMPLES TIRÉS DE STAGES ET PROJETS INDUSTRIELS (POLYTECHNIQUE)

Auteur importé 05/01/14
These au lirmm de Florent Hernandez en 2010 :
Grenelle de l'environnement : production durable (tracteur agriculteur : minimiser les déplacements en respectant les temps de service possible (épandage de produit). Extension à une flotte de

- 3 - **Problème de logistique** : une entreprise doit rationaliser la façon dont elle approvisionne ses différents centres français.
 - liste des villes où un camion de livraison doit passer
 - temps + coût de trajet entre villes
 - contrainte : algorithme rapide car re-calculation fréquent : le nb de camions pour livrer chaque centre est variable.
- **Questions :**
 - quelle route pour minimiser les coûts (= fonction distance)
 - combien de temps pour desservir tous les centres ?
 - combien de camions si on veut approvisionner tous les centres en moins de 2 jours ?
- **Modélisation** -> **circuit hamiltonien dans un graphe**
- **Solution** ? aucune connue ayant un temps polynomial
- **Veille technologique** : il existe un algorithme polynomial si on tolère une marge d'erreur max de 10%

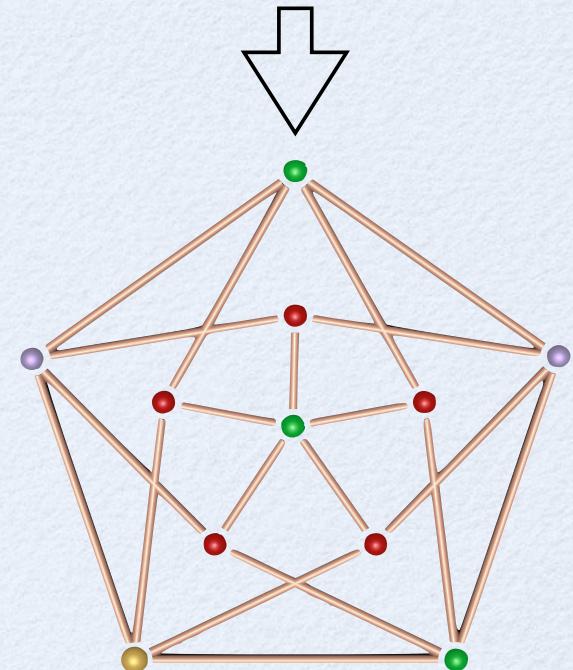
PROBLÈMES DE GRAPHES

De nombreux problèmes d'optimisation peuvent être modélisés sous la forme de problèmes de **graphes**.

8 gardiens suffisent-ils à surveiller le parc ?

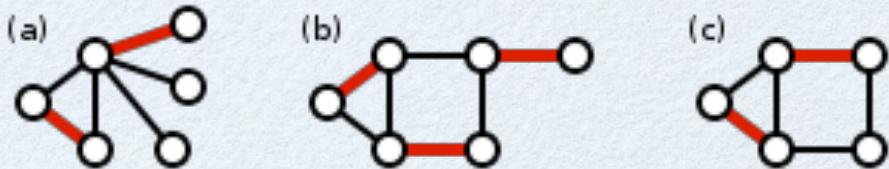
Problème Minimum VertexCover

Soit un G non orienté $G = (S, A)$ (où S est l'ensemble des sommets et A l'ensemble des arêtes), une couverture des sommets (un **vertex cover**) est un sous-ensemble C de S tel que chaque arête st incidente à au moins un sommet.



(problème difficile)

Soit un G non orienté $G = (S, A)$, un **couplage** M est un ensemble d'arêtes deux à deux non adjacentes.



Un **couplage maximum** est un couplage contenant le plus grand nombre possible d'arêtes.

Thm : un couplage maximum peut être trouvé en $O(n^2m)$

Problème Min MaxMatching :

Un **couplage maximal** est un couplage M du graphe tel que, si je rajoute une arête à M , M n'est plus un couplage.

Le problème **Min MaxMatching** est de trouver un couplage maximal avec le plus petit nombre d'arêtes.

Trouver un vertex cover de taille minimum est un **problème difficile** : un algorithme exact aura une complexité exponentielle.

Mais, il existe un algorithme d'approximation très simple (2-approx)

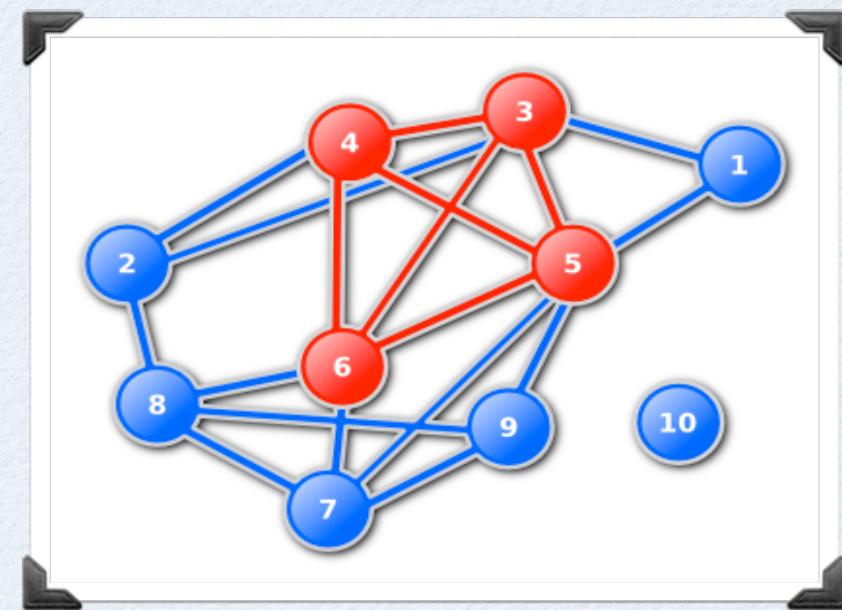
Certains problèmes reviennent dans de nombreux contextes :

MaxClique :

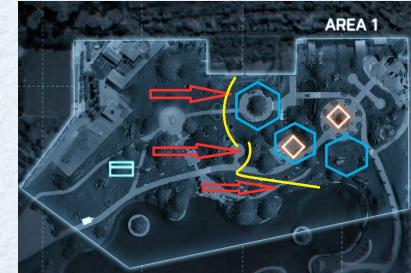
Rappel : une **clique** est un ensemble de sommets tous reliés deux à deux.

Savoir si un graphe possède une clique plus grande qu'un entier k demande un temps exponentiel en le nombre de sommets dans le pire des cas.

Applications : recherche d'information, vision par ordinateur, analyse de réseaux sociaux, chémo-informatique, bioinformatique, ...



PLAN du COURS



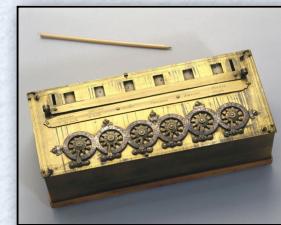
- 1. Rappels sur la complexité d'un algorithme**
- 2. Complexité d'algoritmes récursifs**
- 3. Répartition des problèmes en classes de difficultés**
- 4. Algorithmes exacts par programmation par contrainte**
- 5. Méta-heuristiques : exemple du recuit simulé**
- 6. Travaux dirigés : modélisation d'un problème de la vraie vie**
- 7. Travaux pratiques : résolution d'un problème difficile**

ANALYSE D'UN ALGORITHME

Analyser un algorithme :

- permet de prévoir les ressources nécessaires :
temps calcul, temps E/S, espace mémoire, bande passante, ...

- modèle de machine pour «calculer»



- évaluer le temps de calcul / l'espace mémoire.

Ils dépendent :



- de la machine (jeu d'instructions, ...)
- de la taille de la donnée
- de la nature de la donnée (cas favorable / défavorable)

COMPLEXITÉ D'UN ALGORITHME

- Evaluation du **temps calcul** d'un programme :

on compte le nombre d'**opérations élémentaires** (dont le temps d'exécution est < constante)

- dans le plus mauvais cas..... Pourquoi ?
 - ▶ plus simple à calculer qu'en moyenne
 - ▶ garantie
- en fonction de la **taille de la donnée** :

TAILLE DE LA DONNÉE

Exemples :

deux «petits» entiers : 2

tableau ou ensemble de n éléments : n

algorithme sur des graphes à n sommets : $n+m$

COMPLEXITÉ D'UN ALGORITHME

Ordre de grandeur

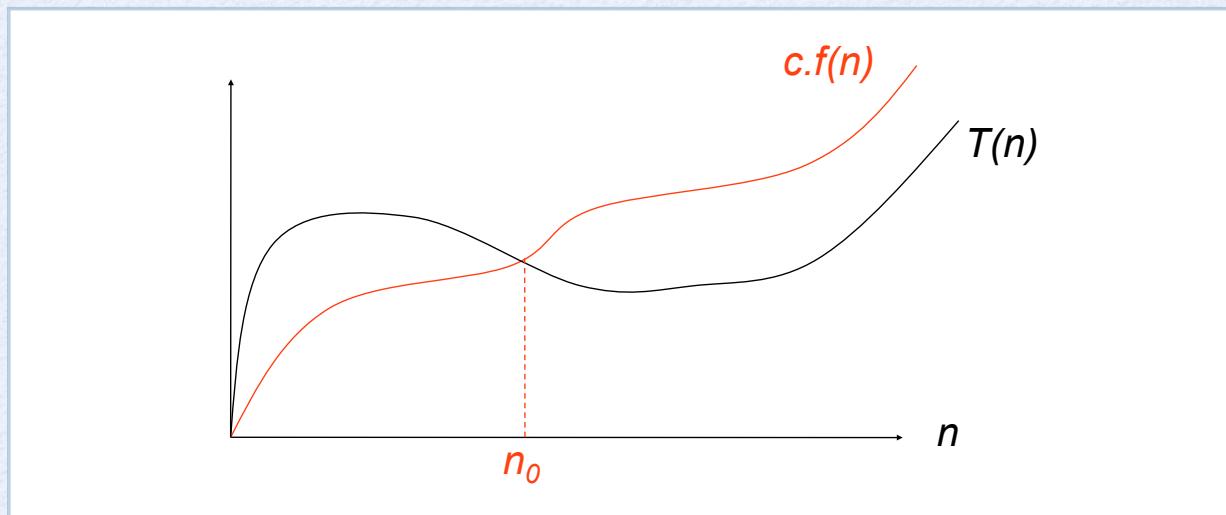
Soit n la taille de la donnée

1. On veut une expression $T_p(n)$ du temps d'exécution du programme p
2. Le temps d'exécution varie d'un facteur plus ou moins constant d'un ordinateur à l'autre
3. Le temps d'exécution pour les petites données est négligeable donc on calcule seulement l'ordre de grandeur de $T(n)$, à une constante près.

$$\mathcal{O}(T(n)) \leq \mathcal{O}(f(n)) \iff \exists c, n_0 : \forall n \geq n_0, \text{on a } T(n) \leq c \times f(n)$$

NOTATION DE LANDAU

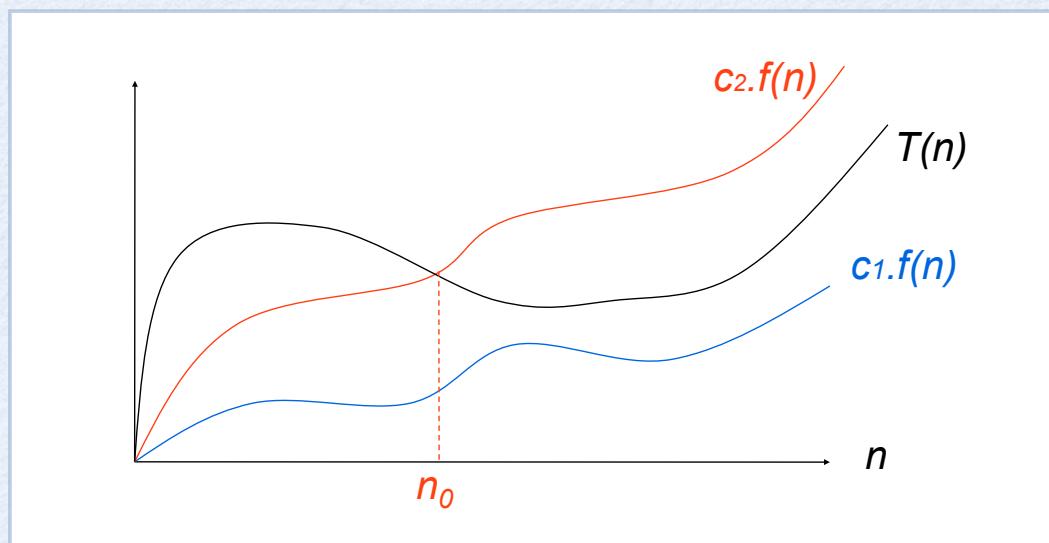
Caractérise le comportement asymptotique de la complexité d'un algorithme (i.e. quand $n \rightarrow \infty$)



$$T(n) \in O(f(n)) \Leftrightarrow \exists c, n_0 > 1 \text{ tels que} \\ \forall n \geq n_0, T(n) \leq c.f(n)$$

NOTATION DE LANDAU

Caractérise le comportement asymptotique de la complexité d'un algorithme (i.e. quand $n \rightarrow \infty$)



$$T(n) \in \Theta(f(n)) \Leftrightarrow \exists c_1, c_2, n_0 > 0, \text{ tels que} \\ \forall n \geq n_0, c_1.f(n) \leq T(n) \leq c_2.f(n)$$

EXEMPLES

$$T(n) = 2n^3 + 6n^2 + 12n + 20 = O(n^3)$$

$$T(n) = n \log n + 4n + 10 = O(n \log n)$$

$$T(n) = 12, n^7 + \frac{2^n}{16} = O(2^n)$$

Remarque : L'utilisation du symbole = est un abus de langage, on devrait écrire $f \in O(g(n))$ car $O(g(n))$ est un ensemble de fonctions.

$$T(n) \in O(f(n)) \Leftrightarrow \exists c, n_0 > 1 \text{ tels que} \\ \forall n \geq n_0, T(n) \leq c.f(n)$$

Dans la série de questions qui suit, l'affirmation proposée est-elle juste ?

Justifier votre réponse en proposant :

- des valeurs pour n_0 et c si l'affirmation est vrai
- ou en donnant l'idée d'une preuve si c'est faux

$$100 = O(1)$$

$T(n) \in O(f(n)) \Leftrightarrow \exists c, n_0 > 1$ tels que
 $\forall n \geq n_0, T(n) \leq c.f(n)$

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

$T(n) \in O(f(n)) \Leftrightarrow \exists c, n_0 > 1$ tels que
 $\forall n \geq n_0, T(n) \leq c.f(n)$

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

$T(n) \in O(f(n)) \Leftrightarrow \exists c, n_0 > 1$ tels que
 $\forall n \geq n_0, T(n) \leq c.f(n)$

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

$T(n) \in O(f(n)) \Leftrightarrow \exists c, n_0 > 1$ tels que
 $\forall n \geq n_0, T(n) \leq c.f(n)$

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

$T(n) \in O(f(n)) \Leftrightarrow \exists c, n_0 > 1$ tels que
 $\forall n \geq n_0, T(n) \leq c.f(n)$

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

$T(n) \in O(f(n)) \Leftrightarrow \exists c, n_0 > 1$ tels que
 $\forall n \geq n_0, T(n) \leq c.f(n)$

si $f(n) = \mathcal{O}(g(n))$ alors $g(n) = \mathcal{O}(f(n))$?

$T(n) \in O(f(n)) \Leftrightarrow \exists c, n_0 > 1$ tels que
 $\forall n \geq n_0, T(n) \leq c.f(n)$

$\forall k \in \mathcal{R}, k.f(n) = \mathcal{O}(f(n))$?

$T(n) \in O(f(n)) \Leftrightarrow \exists c, n_0 > 1$ tels que
 $\forall n \geq n_0, T(n) \leq c.f(n)$

si $f(n) > 1, f(n) + k = \mathcal{O}f(n)$?

$$T(n) \in O(f(n)) \Leftrightarrow \exists c, n_0 > 1 \text{ tels que} \\ \forall n \geq n_0, T(n) \leq c.f(n)$$

Montrez que $(n+a)^b = O(n^b)$ où a et b sont des constantes et $b>0$

COMPLEXITÉ D'UN ALGORITHME

Evaluation de $T(n)$ **séquence**

Traitement1 (*effectué en $T_1(n)$*)

Traitement2 (*effectué en $T_2(n)$*)

Somme des coûts : $T(n) = T_1(n) + T_2(n)$

COMPLEXITÉ D'UN ALGORITHME

Evaluation de T (n) conditionnelle

si < condition > alors

Traitemet1 $T_1(n)$

sinon

Traitemet2 $T_2(n)$

T(n) = ?

COMPLEXITÉ D'UN ALGORITHME

Evaluation de $T(n)$ **conditionnelle**

si < condition > alors

Traitement1 $T_1(n)$

sinon

Traitement2 $T_2(n)$

Max des coûts :

$$T(n) \leq T(\text{condition}) + \max(T_1(n), T_2(n))$$

$$T(n) \in O(f(n)) \Leftrightarrow \exists c, n_0 > 1 \text{ tels que} \\ \forall n \geq n_0, T(n) \leq c.f(n)$$

Soit $f(n)$ et $g(n)$ deux complexités d'algorithmes.

Pouvez-vous trouver l'idée pour expliquer que

$$\max(f(n), g(n)) \text{ est en } O(f(n)+g(n))$$

$$T(n) \in O(f(n)) \Leftrightarrow \exists c, n_0 > 1 \text{ tels que} \\ \forall n \geq n_0, T(n) \leq c.f(n)$$

Soit $f(n)$ et $g(n)$ deux complexités d'algorithmes.

Pouvez-vous trouver l'idée pour expliquer que

$$\max(f(n), g(n)) \text{ est en } O(f(n)+g(n))$$

Solution :

on choisit $c=1$ et $n_0=1$, il faut alors justifier que

$$\max(f(n), g(n)) \leq f(n)+g(n)$$

Ce qui est évident car en fait $f(n)+g(n) = \max(f(n), g(n)) + \min(f(n), g(n))$

COMPLEXITÉ D'UN ALGORITHME

**Evaluation de $T(n)$ boucle POUR
Coût de l'ensemble des itérations**

Pour i de 1 à n par pas de p faire

 Traitement itération $T_i(n)$
fin faire

- $T(n) =$

COMPLEXITÉ D'UN ALGORITHME

**Evaluation de $T(n)$ boucle POUR
Coût de l'ensemble des itérations**

Pour i de 1 à n par pas de p faire

 Traitement itération $T_i(n)$
fin faire

- $T(n) = (n/p)^* T_i(n)$

COMPLEXITÉ D'UN ALGORITHME

Evaluation de $T(n)$

Coût de l'ensemble des itérations

tant que < condition > faire

 Traitement itération

fin faire

boucle TANT QUE

$T_i(n)$

soit n_i le nombre d'itérations

$$\bullet T(n) = n_i * T_i(n) + (n_i + 1) * T(\text{cond}^\circ)$$

$$\bullet T(n) = \sum_{i=1}^k T_i(n) + k * T(\text{cond}^\circ)$$

- ATTENTION : on ne peut parler de complexité que lorsque le nombre d'itération de la boucle est fini (à prouver grâce aux invariants de boucles)

Complexité amortie

fonction Tri (S, n)

```
1  pour  $i := n$  à 2 faire          ( $n - 1$  fois)
2    pour  $j := 1$  à  $i - 1$  faire      ( $i - 1$  fois)
3      si ( $S[j] > S[j + 1]$ ) alors
4        permuter  $S[j]$  et  $S[j + 1]$  dans  $S$ ,
```

Combien de fois le bloc SIAlorsSinon est-il exécuté ?

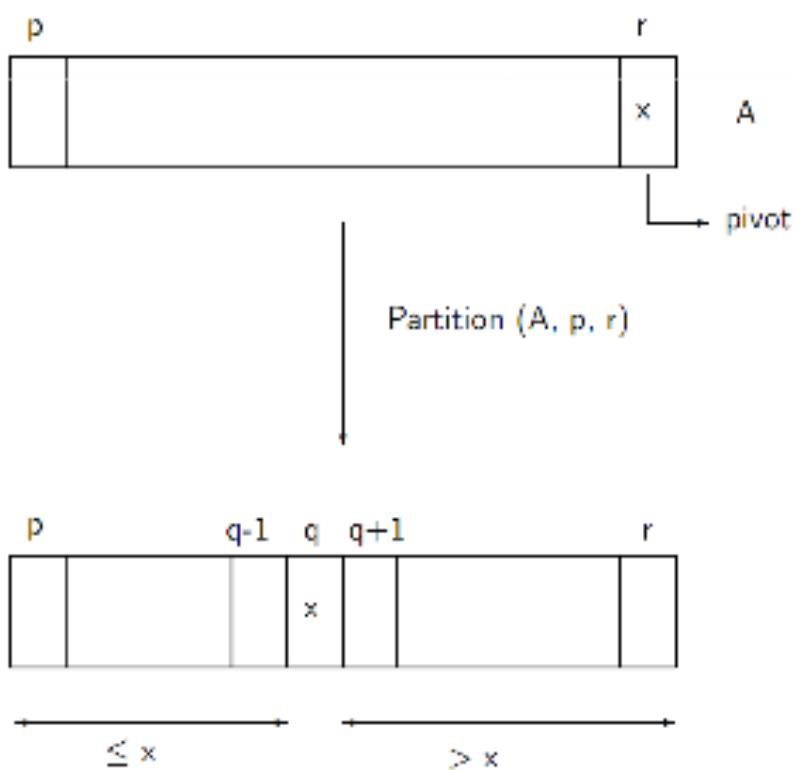
Quel est la complexité de l'algorithme ?

4. Donnez la complexité de Partitionner (A,p,r) :

A : un tableau de n éléments
p,r : des indices de cases

QuickSort (A, p , r)

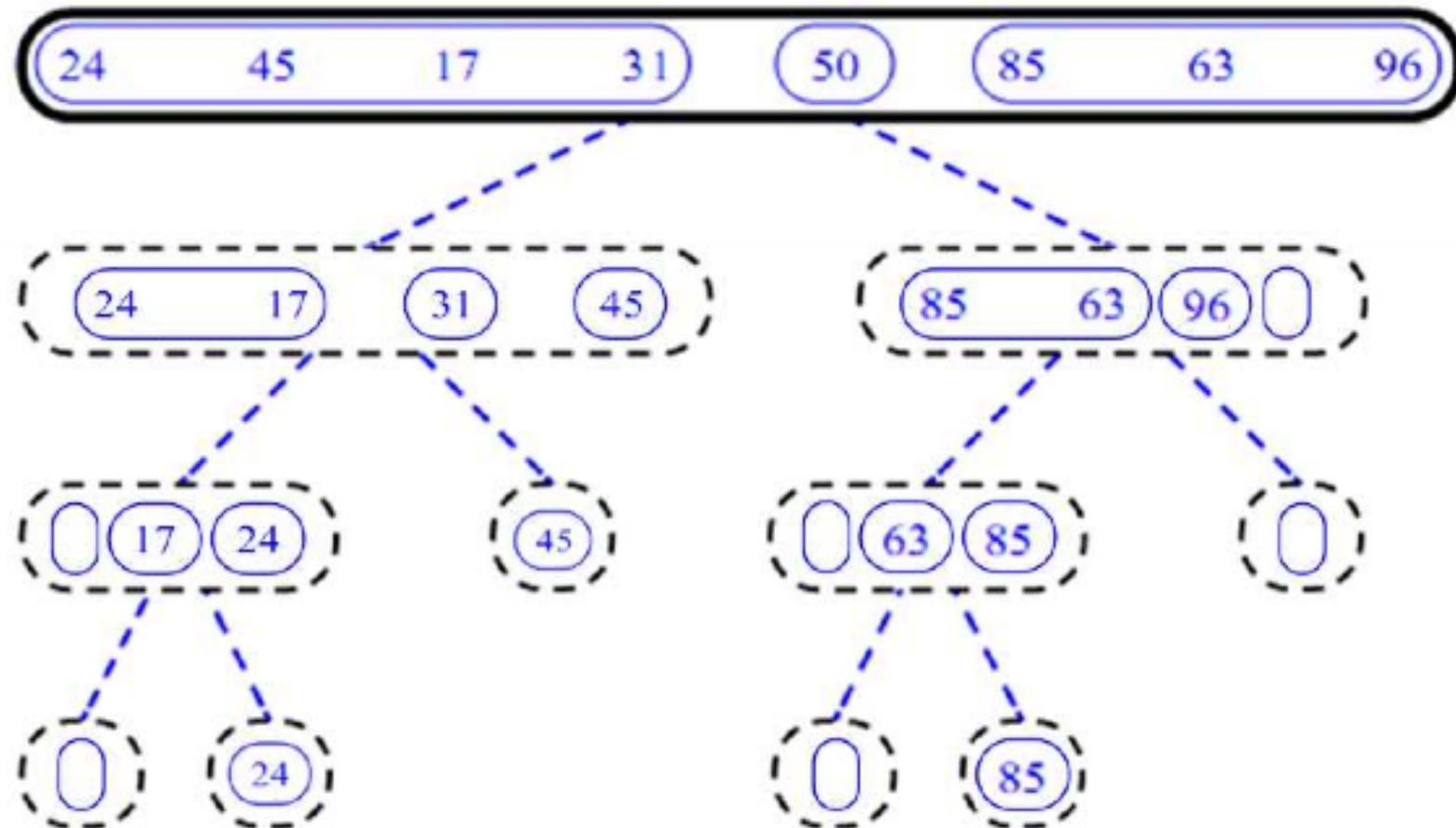
```
si p<r
    q = Partition (A, p, r);
    QuickSort (A, p, q-1);
```



Partition (A, p, r) :

```
pivot = A[r];
i=p; j=r;
while i<j
    while pivot>A[i] && (i<j) i++;
    if (i<j) {Echanger A[i] et
               A[j]; j-- ;}
    while pivot<A[j] && (i<j) j--;
    if (i<j) {Echanger A[i] et
               A[j]; i++ ;}
renvoyer i;
```

Exemple d'arbre de partition du **QuickSort**



d'après P. Prosser

Evaluation de $T(n)$ (fonctions récursives)

fonction exemple-récursive (n)

1 **si** ($n > 1$) **alors**

2 **Recursive**($n/2$), **coût** $T(n/2)$

3 **Traitement**(n), **coût** $C(n)$

4 **Recursive**($n/2$), **coût** $T(n/2)$

Equation récursive

$$T(n) = 2 * T(n/2) + C(n)$$

si $C(n) = 1$ **alors** $T(n) = K \times n$

pour K une constante

si $C(n) = n$ **alors** $T(n) = K \times n \times \log n$

COMPLEXITÉ AMORTIE

Il s'agit d'évaluer globalement la complexité d'une suite d'opérations, plutôt que de raisonner séparément pour chacune.

Exemple :

Tri Fusion (T[1..n])

- 1 si ($n = 1$) alors **renvoyer** T
- 2 **décomposer** T en T_1 et T_2
- 3 $T_1 := \text{Tri Fusion} (T_1)$
- 4 $T_2 := \text{Tri Fusion} (T_2)$
- 5 $T := \text{fusion} (T_1, T_2)$
- 6 **renvoyer** T

Remarques :

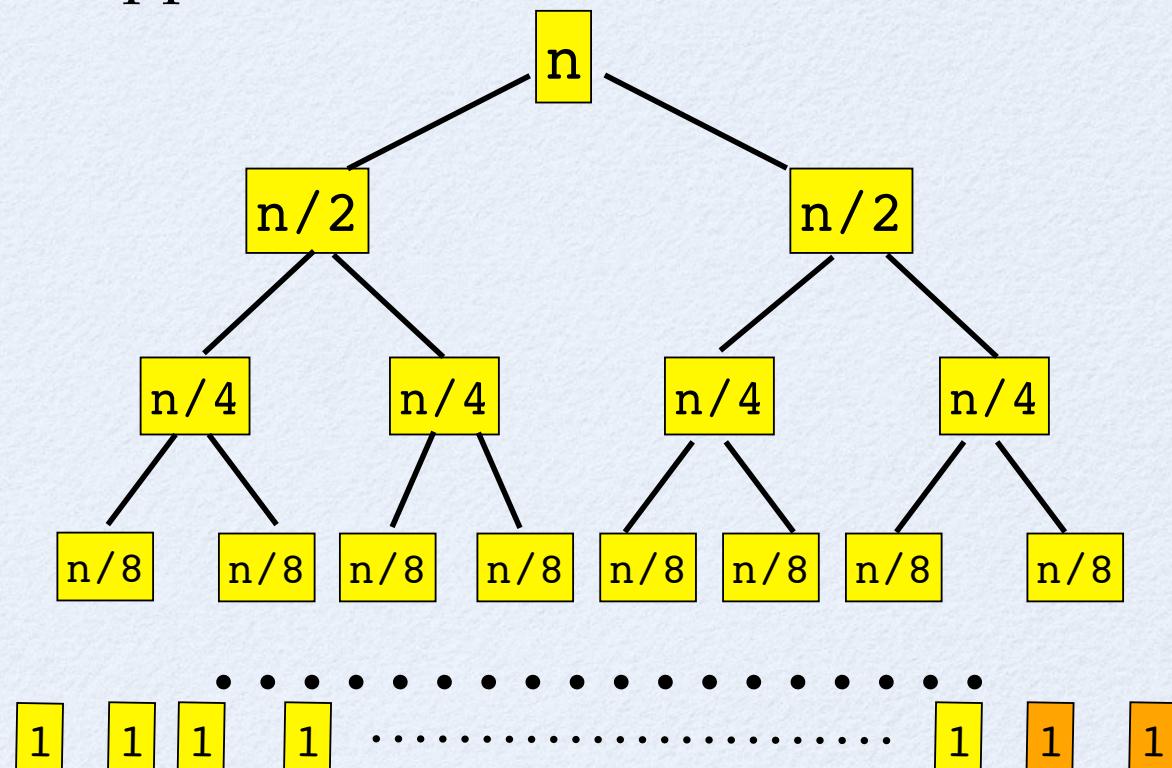
$O(1)$
 $\Theta(n)$
appel réc. sur $n/2$ éléments
appel réc. sur $n/2$ éléments
 $\Theta(n)$

ANALYSE AMORTIE

Analysez le tri fusion en raisonnant par niveau :

Arbre visualisant les appels récursifs :

à chaque étage de l'arbre, on doit faire des fusions qui coûtent $O(n)$ en tout, et on a $O(\log(n))$ étages



$$T(n) \leq c \times n \times (\log_2(n) + 1)$$

RÉSISTANCE AU PROGRÈS

- On pourrait penser qu'au fur et à mesure que les ordinateurs sont plus rapides, l'importance d'avoir un algorithme efficace soit moindre...
- Quelle est la taille **maximale** du problème que l'on peut traiter ?

$f(n)$	En 1 seconde	En 1 minute	En 1 heure
n	1×10^6	6×10^7	3.6×10^9
$400n$	2500	150 000	9×10^6
$2n^2$	707	5477	42426
n^4	31	88	244
2^n	19	25	31

RÉSISTANCE À L'UNIVERS

Quelques ordres de grandeur :

Nb d'atomes de la terre : 10^{50}

Nombre d'atomes de l'univers : 10^{80}

8 10^{20} octets en 2010 (80 milliards de gigaoctets) Quantité de données numériques dans le monde.

Nb minimum de parties possibles aux échecs : 10^{120}

Nb de parties de Go : estimé (selon les sources) entre 10^{172} et 10^{600}

Limite théorique de la puissance de calcul = 10^{102} : en supposant la mobilisation de toutes les ressources de l'univers, on obtiendrait : une capacité mémoire de 10^{120} bits, une cadence de l'horloge interne de 10^{102} Hz

RÉSISTANCE AU PROGRÈS

- Si m est la taille maximale que l'on pouvait traiter en un temps donné, que devient m si l'on reçoit de notre sponsor favori un processeur 256 fois plus rapide?

$f(n)$	Nouvelle taille maximale
n	$256m$
$400n$	$256m$
$2n^2$	$16m$
n^4	$4m$
2^n	$m+8$

De ce calcul il ressort qu'il vaut bien mieux passer d'un algorithme exponentiel à un algorithme polynomial que d'attendre après les progrès technologiques (tant que les ordinateurs utilisent le même modèle de calcul)

PRÉCAUTIONS

La règle générale s'applique mais donne dans ce cas une équation récurrente.

Attention \mathcal{O} ne s'applique qu'à une fonction déjà définie.

Pour les appels récursifs, il ne faut pas employer \mathcal{O} .

Exemple 1

$T(n) = T(n - 1)$ a comme solution $T(n) = T(0)$ tandis que

$T(n) = 2T(n - 1)$ a comme solution $T(n) = T(0).2^n$, même si

$\mathcal{O}(T(n - 1)) = \mathcal{O}(2T(n - 2))$

Attention à
l'utilisation de $\mathcal{O}()$
sur des fonctions
récursives

PRÉCAUTIONS

$T(n) = T(n - 1)$	$T(n) = 2T(n - 1)$
$T(0)$	$T(0)$
$T(1) = T(0)$	$T(1) = 2T(0)$
$T(2) = T(0)$	$T(2) = 4T(0)$
$T(3) = T(0)$	$T(3) = 8T(0)$
...	...
$T(n) = T(0)$	$T(n) = 2^n T(0)$

PRÉCAUTIONS

Exemple 2

$$T(n) = T(n - 1) + \mathcal{O}(1)$$

$$T(0) = \mathcal{O}(1)$$

On pourrait croire que

$$T(0) = \mathcal{O}(1)$$

$$T(1) = \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(1)$$

$$T(2) = \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(1)$$

...

et que par induction :

$T(n) = \mathcal{O}(1)$, ce qui est évidemment faux puisque $T(n) = \mathcal{O}(n)$.