

Algorithmique Avancée & Programmation

Séance 3 : complexité

Test 3

- Déposer fichiers sources + makefile sur moodle
- 20 minutes

Cadrage séance 3 : Complexité

- Test papier : exos sur structures de piles/listes
 - Surtout pratique
- Présentation d'algorithmes, étude de leur complexité
- Equations de récurrence
 - Théorème maître
- Implémentation de tris

Rappels *Cf. Capsule 6*

Complexité

Méfiez-vous de ceux qui affirment :

“Pourquoi s'embêter ? Il suffit d'attendre !”

- « *A coût constant, la rapidité des processeurs double tous les 18 mois* » - Les capacités de stockage suivent la même loi
 - Gordon Moore, co-fondateur d'Intel, 1965
- « *Software is getting slower more rapidly than hardware becomes faster* » - Le logiciel ralentit plus vite que le matériel n'accélère !
 - Niklaus Wirth, Prix Turing, 1995

Complexité : effet d'une amélioration

- Soit N , la taille maximale des données que l'on peut aujourd'hui traiter en 1 heure
- Quelle taille pourra-t-on traiter en 1 heure avec le même programme lorsque les ordinateurs seront 100 fois plus rapides ?
- Exemple 1 : $T(n) = \Theta(n^2)$
 - Aujourd'hui : $v \times N^2 = 1h$
 - Demain : $v/100 \times N'^2 = 1h$
 - $N' = 10.N$
- Exemple 2 : $T(n) = \Theta(2^n)$
 - Aujourd'hui : $v \times 2^N = 1h$
 - Demain : $v/100 \times 2^{N'} = 1h$
 - $N' = 6,6 + N$

⇒ Le surcroît de puissance importe peu !

⇒ Analysez la complexité de vos algorithmes !

⇒ Tâchez d'en produire de complexité raisonnable !

Gnuplot

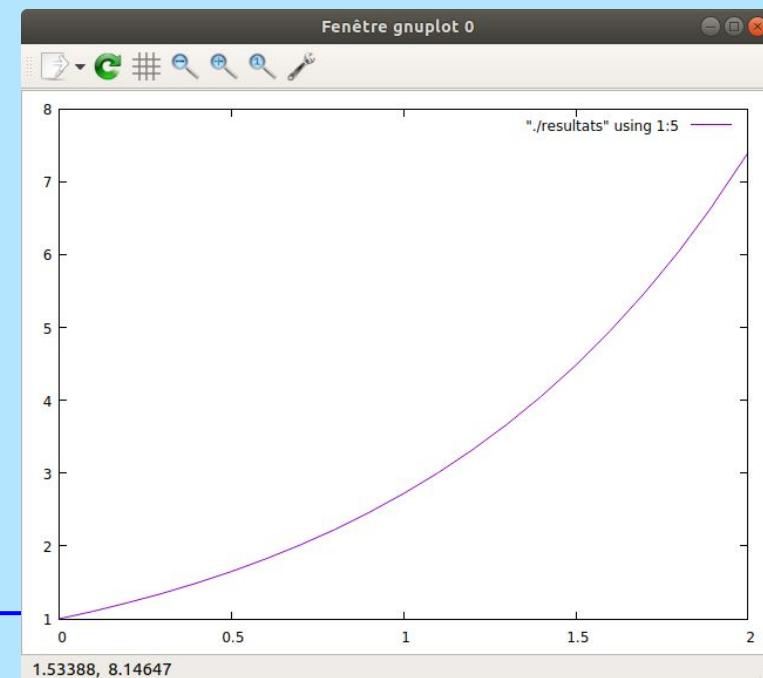
- Interface pour la représentation graphique de données provenant d'un fichier texte
- Paquet **gnuplot** à installer sur votre distribution
- Présentation : <https://doc.ubuntu-fr.org/gnuplot>
- Galerie de démos : <http://gnuplot.sourceforge.net/demo/>

Exemple

\$ gnuplot

```
gnuplot> plot "./resultats"
using 1:5 with lines
```

- **using valeurs_abs:val_ord**
 - Spécifier données en abs/ord
- **with lines**
 - Relier les points par une ligne brisée



```
x cos sin log exp
0 1 0 -inf 1
0.1 0.995004 0.0998334 -2.30259 1.10517
0.2 0.980067 0.198669 -1.60944 1.2214
0.3 0.955336 0.29552 -1.20397 1.34986
0.4 0.921061 0.389418 -0.916291 1.49182
0.5 0.877583 0.479426 -0.693147 1.64872
0.6 0.825336 0.564642 -0.510826 1.82212
0.7 0.764842 0.644218 -0.356675 2.01375
0.8 0.696707 0.717356 -0.223144 2.22554
0.9 0.62161 0.783327 -0.105361 2.4596
1 0.540302 0.841471 -1.11022e-16 2.71828
1.1 0.453596 0.891207 0.0953102 3.00417
1.2 0.362358 0.932039 0.182322 3.32012
1.3 0.267499 0.963558 0.262364 3.6693
1.4 0.169967 0.98545 0.336472 4.0552
1.5 0.0707372 0.997495 0.405465 4.48169
1.6 -0.0291995 0.999574 0.470004 4.95303
1.7 -0.128844 0.991665 0.530628 5.47395
1.8 -0.227202 0.973848 0.587787 6.04965
1.9 -0.32329 0.9463 0.641854 6.68589
2 -0.416147 0.909297 0.693147 7.38906
```

[resultats.dem](#)

Plan

- Notations de Landau, classes de complexité
- Diviser pour régner, Résolution d'équations de récurrence
- Développement de fonctions récursives, expression des équations de coût
- Tris rapides
- Code Couleur

Après cette séance, vous devez savoir :

- Ecrire des équations de récurrence correspondant à vos algorithmes
- Résoudre des équations de récurrence “manuellement” ou avec le théorème maître



Notations asymptotiques de Landau

https://fr.wikipedia.org/wiki/Comparaison_asymptotique

Objectifs

- L'évolution des matériels informatiques et la complexité des problèmes traités aujourd'hui et à traiter demain nécessitent une réelle **analyse**
- Besoins d'**outils mathématiques** permettant l'analyse des performances d'un algorithme
 - Avoir une idée de ce qui est faisable et infaisable actuellement
 - Améliorer les performances des problèmes faciles
 - Savoir comment aborder les problèmes difficiles

Complexité des algorithmes

- L'exécution d'un programme a toujours un coût et il existe deux paramètres essentiels pour l'évaluer :
 - Le temps d'exécution : la **complexité temporelle**
 - L'espace mémoire requis : la **complexité spatiale**
- Aujourd'hui la **complexité temporelle** est le point sensible

Analyse de la complexité des algorithmes : Objectifs

- Proposer des méthodes pour **estimer** le coût d'un algorithme
- Être capable de **comparer** deux algorithmes sans avoir à les programmer
- Définir une **mesure** :
 - Indépendante de l'ordinateur
 - Indépendante du langage de programmation
 - i.e. indépendante des spécificités d'implémentation

Analyse de la complexité des algorithmes : Opération de base

- Une opération de base d'un algorithme est une **opération élémentaire significative** pour le problème traité et qui, à une constante près, est effectuée au moins aussi souvent que n'importe quelle autre opération élémentaire de l'algorithme
- Exemples :
 - Pour les algorithmes de tri (par comparaison), l'opération de base est la **comparaison de deux valeurs** (ou de deux **clés**)
 - Pour la multiplication de matrices, l'opération de base est la **multiplication de deux nombres**

Quelle complexité ?

- On distingue trois sortes de complexité :
 - La complexité dans le **meilleur des cas**
 - La complexité dans le **pire des cas**
 - La complexité en **moyenne**
 - On calcule le coût pour chaque jeu de données possible puis on divise la somme de ces coûts par le nombre de jeux de données différents
- En analyse de complexité, on étudie souvent le **pire cas** qui donne une borne supérieure de la complexité de l'algorithme

Remarques

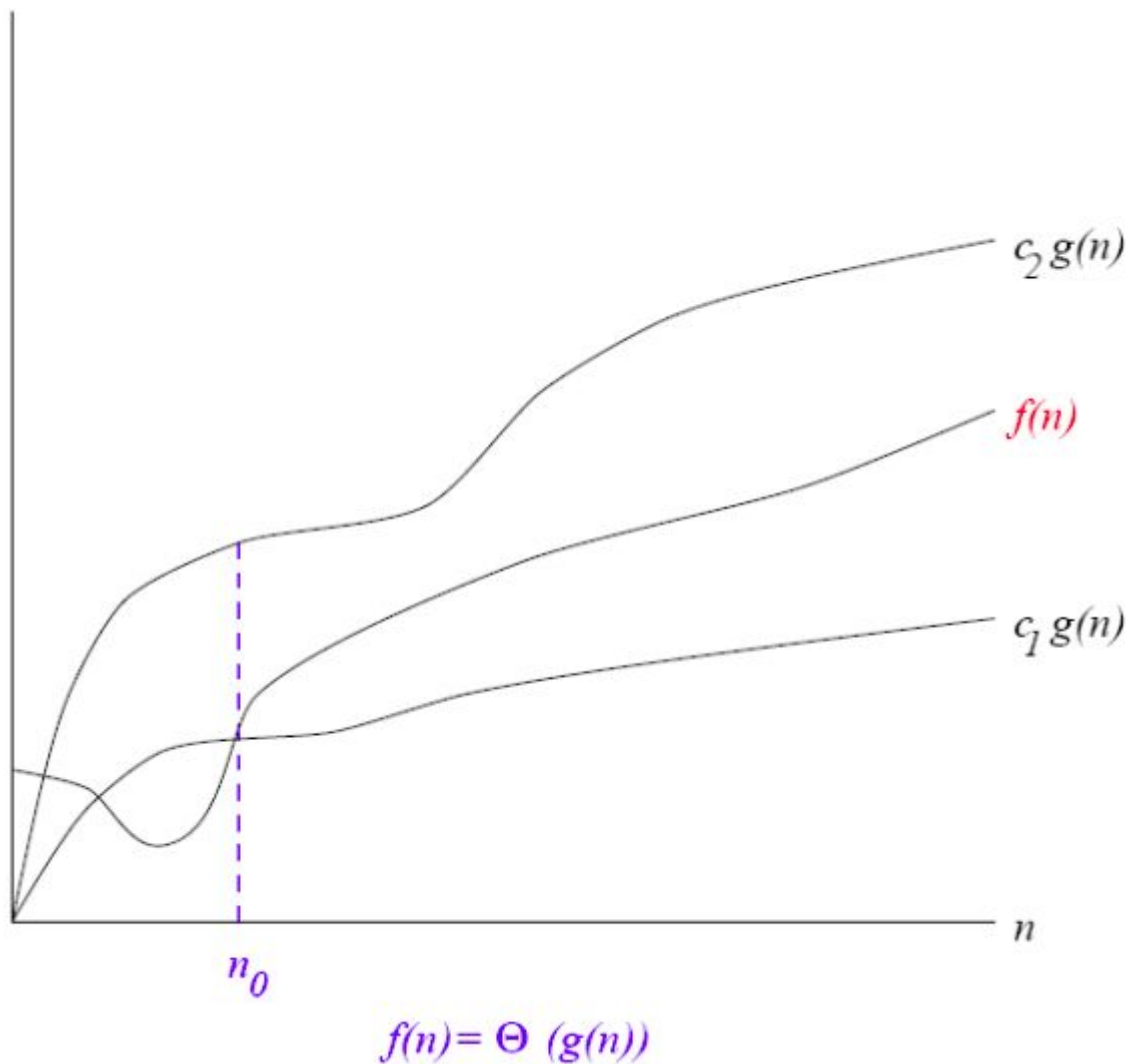
- Le cas **moyen** et le **pire** cas sont **souvent** de complexité équivalente
- Lorsqu'on étudie la complexité d'un algorithme, on ne s'intéresse pas au temps de calcul réel mais à un **ordre de grandeur**
 - Pour une complexité polynomiale, par exemple, on ne s'intéresse qu'au **terme de plus grand ordre**
- Lorsqu'on analyse deux algorithmes de traitement d'un même problème, on compare leur complexité **asymptotiquement**
- On exprime la complexité d'un algorithme comme une **fonction $f(n)$ de \mathbb{N} dans \mathbb{R}**

Notation Θ (grand Theta)

- Soient f et g deux fonctions de \mathbb{N} dans \mathbb{R} .
- On dit que $g(n)$ est une **borne approchée asymptotique** pour $f(n)$ s'il existe deux constantes $c_1 > 0$ et $c_2 > 0$, ainsi que $n_0 \in \mathbb{N}$ tels que pour tout $n \geq n_0$ on ait :
 - $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$
- Ceci revient à dire que :
 - **$f(n)$ a une croissance comparable à celle de $g(n)$**
- On écrit **$f(n) = \Theta(g(n))$**

Notation Θ (grand Theta)

Illustration



La notation Θ borne
asymptotiquement
une fonction
à la fois par excès
et par défaut

Exemple

- $f(n) = n^2/2 \mp 3n = \Theta(n^2)$
- Il faut déterminer c_1 , c_2 et n_0 tels que :
 - $0 \leq c_1 n^2 \leq n^2/2 \mp 3n \leq c_2 n^2 \quad 0 \leq n_0 \leq n$
 - $c_1 \leq 1/2 \mp 3/n \leq c_2$
- Pour $n_0 = 8$, $c_1 \leq 4/8 \mp 3/8 \leq c_2$
- Pour $n_0 \leq n$, $1/8 \leq 1/2 \mp 3/n \leq 7/8$
 $\Rightarrow c_1 = 1/8 \quad c_2 = 7/8 \quad \text{et} \quad n_0 = 8$

Evaluation d'un polynôme

Opération de base :
multiplication entre
nombres réels

- $P(x_0) = \sum_{k=0 \rightarrow n} a_k x_0^k$
 - n additions
 - $n(n+1)/2$ multiplications

⇒ $\Theta(n^2)$

- Mise sous forme de Horner (cf. [TEA S2 - 2021](#))

$$P(x_0) = ((\dots ((a_n x_0 + a_{n-1})x_0 + a_{n-2})x_0 + \dots)x_0 + a_1)x_0 + a_0$$

- n multiplications et n additions

⇒ $\Theta(n)$

$$P = a_n x_0^n + a_{n-1} x_0^{n-1} + \dots + a_0$$

Diagram illustrating the number of multiplications required for each term in the polynomial expansion:

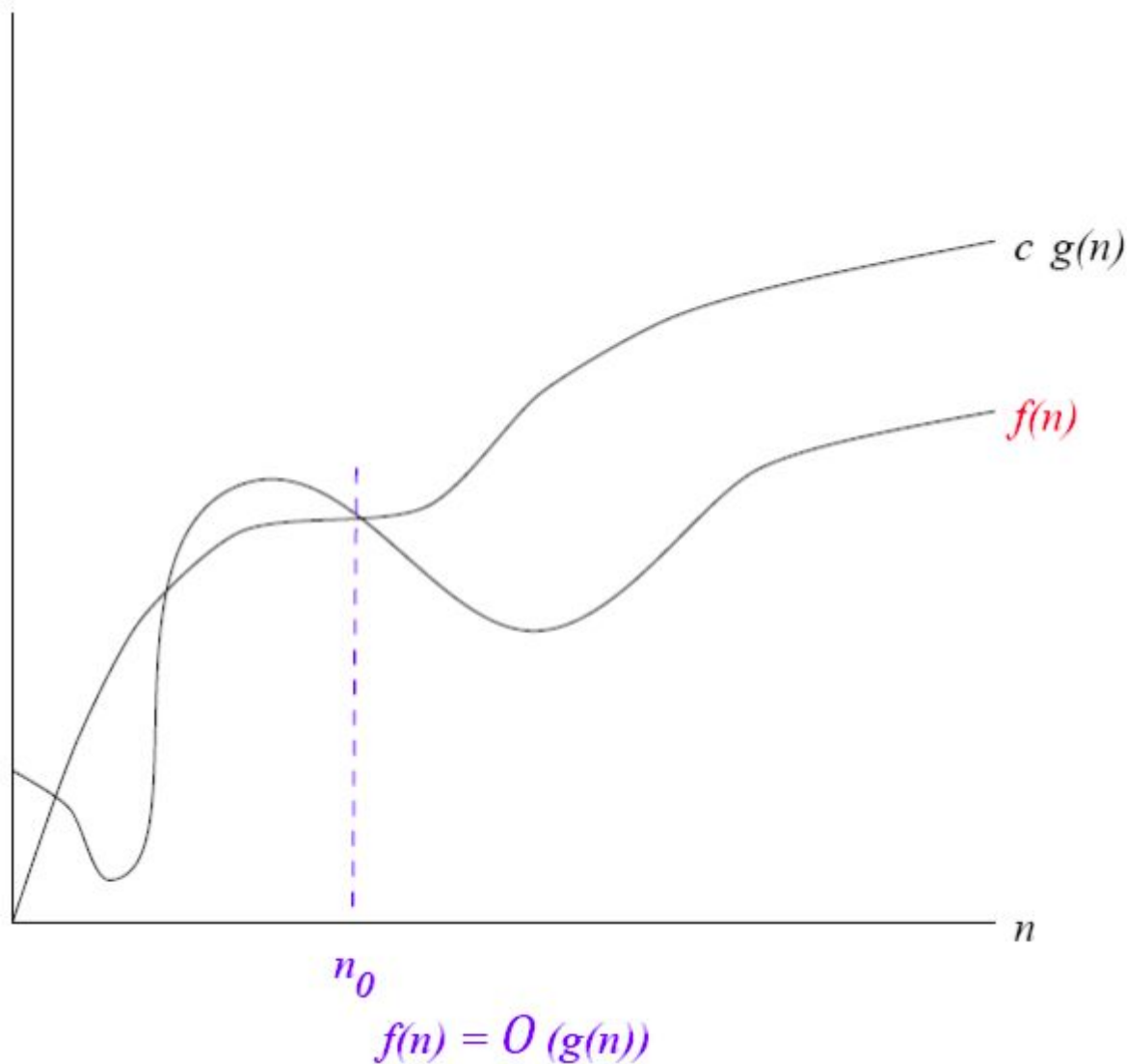
- For $a_n x_0^n$: $1 + (n-1) = n$ multiplications
- For $a_{n-1} x_0^{n-1}$: $1 + (n-2) = n-1$ multiplications
- For a_0 : 0 multiplication

Notation O (grand O)

- On dit que $g(n)$ est une **borne supérieure asymptotique** pour $f(n)$ s'il existe une constante $c > 0$, ainsi qu'un entier naturel n_0 tels que pour tout $n \geq n_0$ on ait :
 - $0 \leq f(n) \leq c g(n)$
- Ceci revient à dire que :
 - **$f(n)$ est inférieure à $g(n)$** pour n assez grand
- On écrit **$f(n) = O(g(n))$**

Notation O (grand O)

Illustration



*La notation O (grand O) est utilisée quand on ne dispose que d'une **borne supérieure asymptotique***

Exemples

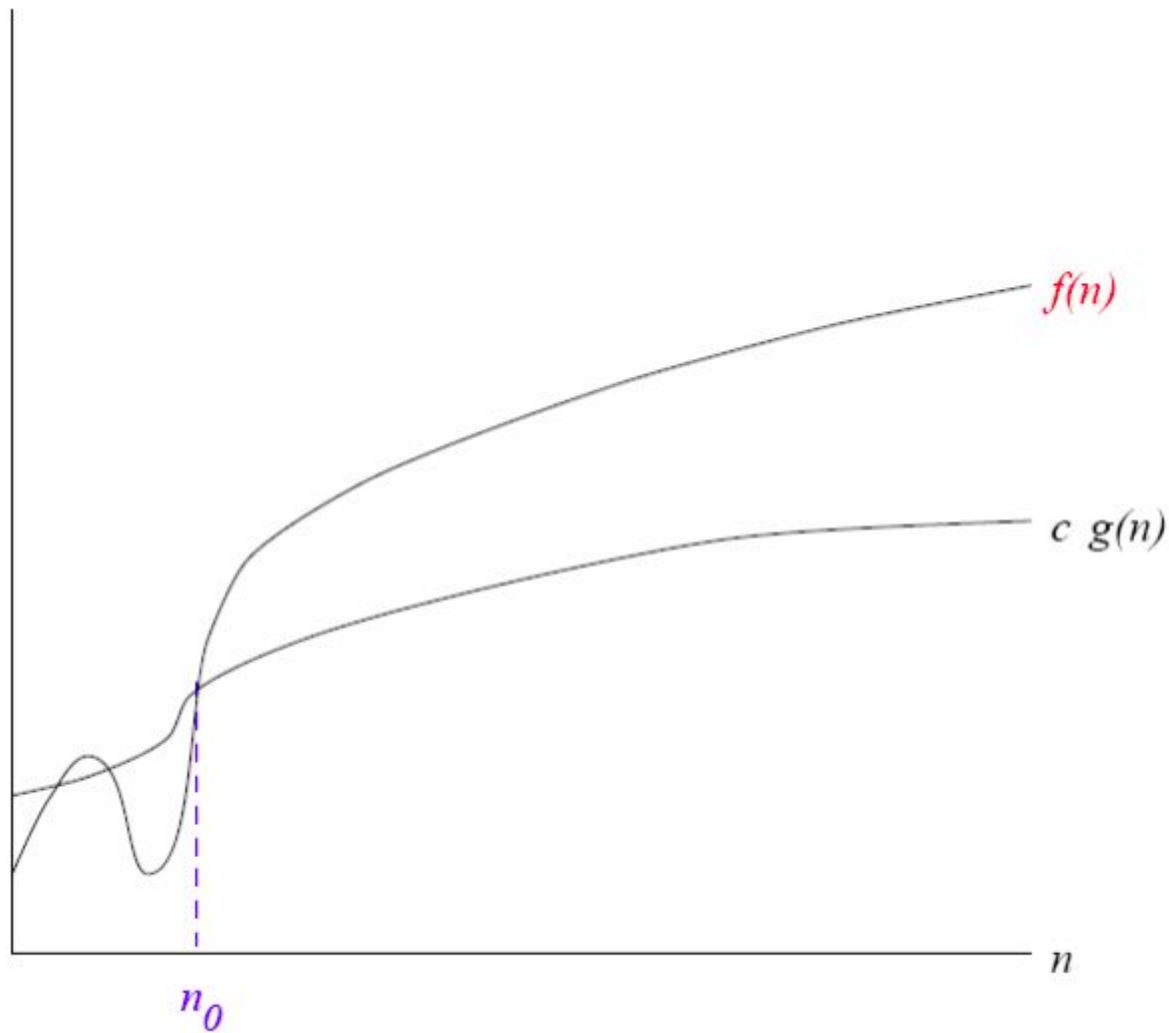
- $f(n) = a n^2 + b n + c$ avec $a > 0$
 - $f(n) = \Theta(n^2)$
 - On a également $f(n) = O(n^2)$
- $f(n) = b n + c$
 - $f(n) = \Theta(n)$
 - Les propositions $f(n) = O(n^2)$ et $f(n) = O(n)$ sont vraies !
- $f(n) = n \log n + 12 n + 567 = O(n \log n)$
- $f(n) = 123 n^{10} - 45 n^7 + 67 n^4 + 2^n / 890 = O(2^n)$

Notation Ω (grand Omega)

- On dit que $g(n)$ est une **borne inférieure asymptotique** pour $f(n)$ s'il existe une constante $c > 0$ ainsi qu'un entier naturel n_0 tels que pour tout $n \geq n_0$ on ait :
 - $0 \leq c g(n) \leq f(n)$
- Ceci revient à dire que **$f(n)$ est supérieure à $g(n)$** pour n assez grand.
- On écrit **$f(n) = \Omega(g(n))$**

Notation Ω (grand Omega)

Illustration



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

La notation Ω (grand Ω) est utilisée quand on ne dispose que d'une **borne inférieure asymptotique**

Exemple

- $f(n) = b n + c$
 - $f(n) = \Theta(n)$
 - On a également $f(n) = \Omega(n)$
- $f(n) = a n^2 + b n + c$ avec $a > 0$
 - $f(n) = \Theta(n^2)$
 - Les propositions $f(n) = \Omega(n^2)$ et $f(n) = \Omega(n)$ sont vraies !

Propriétés

- $f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$
- $f(n) = \Theta(g(n)) \Rightarrow f(n) = \Omega(g(n))$
- $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \text{ et } f(n) = \Omega(g(n))$
- Les notations Θ , O , Ω sont transitives
- Les notations Θ , O , Ω sont réflexives
- Seule la notation Θ est symétrique

Règles de sommation

- $O(f(n)) + O(g(n)) \rightarrow O(\max(f(n), g(n)))$
- Si $g(n)$ est d'ordre inférieur à $f(n)$:
 - $O(f(n) \pm g(n)) \rightarrow O(f(n))$
- Considérons un programme constitué de deux blocs en séquence P et Q , respectivement de complexité en $O(f(n))$ et $O(g(n))$
- $P ; Q \rightarrow O(f(n) + g(n))$

Conditionnelle/Itération

nombre d'éléments
entre a et b *inclus* :
 $b - a + 1$

- Si le test d'une instruction conditionnelle ne comporte pas d'appel à une fonction :
 - if (test) P ; else Q ; $\rightarrow O(\max(f(n), g(n)))$
- Boucle for :
 - for ($i = a$; $i \leq b$; $i++$) P ; $\rightarrow O((b-a+1) f(n))$
- De même, pour une boucle while, on compte le nombre d'itérations $i(n)$:
 - while (test) P ; $\rightarrow O(i(n) f(n))$



Exemple : **ex1**

- Tester les algorithmes de tris présentés ci-dessous
 - Compter le nombre d'affectations et de comparaisons pour chacun d'eux
- Afficher les courbes représentant leur comportement
 - Vérifier les évaluations formelles de complexité
- Ajouter la courbe correspondant à l'expression formelle de complexité aux courbes produites par nos programmes

⇒ Il suffit d'ajouter , x^{**2} à la fin des commandes **plot** dans les fichiers `output/.../*.plt`, puis d'exécuter la commande `gnuplot output/.../*.plt`

Gnuplot

Cf. Capsule 6

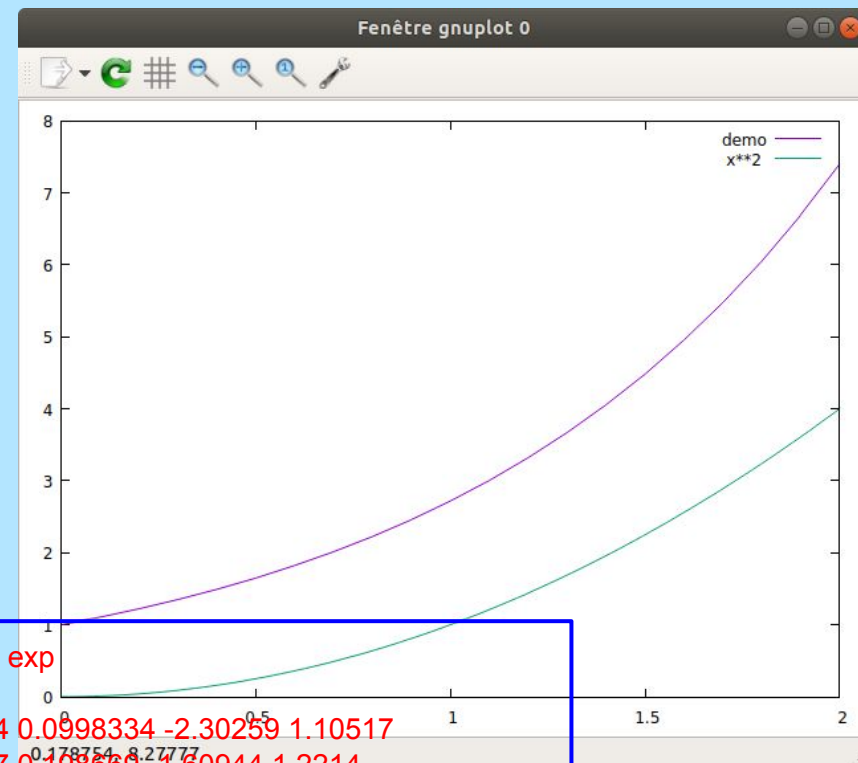
- Interface pour la représentation graphique de données provenant d'un fichier texte
- Paquet **gnuplot** à installer sur votre distribution
- Présentation : <https://doc.ubuntu-fr.org/gnuplot>
- Galerie de démos : <http://gnuplot.sourceforge.net/demo/>

Exemple

\$ gnuplot

```
gnuplot> plot "./resultats.dem"  
using 1:5 with lines title 'demo', x**2
```

- **using valeurs_abs:val_ord**
 - Spécifier données en abs/ord
- **with lines**
 - Relier les points par une ligne brisée
- **title 'demo'**
 - Titre de la courbe dans la légende



```
x cos sin log exp  
0 1 0 -inf 1  
0.1 0.995004 0.0998334 -2.30259 1.10517  
0.2 0.980067 0.198669 -1.60944 1.2214  
0.3 0.955336 0.29552 -1.20397 1.34986  
0.4 0.921061 0.389418 -0.916291 1.49182  
0.5 0.877583 0.479426 -0.693147 1.64872  
0.6 0.825336 0.564642 -0.510826 1.82212  
0.7 0.764842 0.644218 -0.356675 2.01375  
0.8 0.696707 0.717356 -0.223144 2.22554  
0.9 0.62161 0.783327 -0.105361 2.4596  
1 0.540302 0.841471 -1.11022e-16 2.71828  
1.1 0.453596 0.891207 0.0953102 3.00417  
1.2 0.362358 0.932039 0.182322 3.32012  
1.3 0.267499 0.963558 0.262364 3.6693  
1.4 0.169967 0.98545 0.336472 4.0552  
1.5 0.0707372 0.997495 0.405465 4.48169  
1.6 -0.0291995 0.999574 0.470004 4.95303  
1.7 -0.128844 0.991665 0.530628 5.47395  
1.8 -0.227202 0.973848 0.587787 6.04965  
1.9 -0.32329 0.9463 0.641854 6.68589  
2 -0.416147 0.909297 0.693147 7.38906
```

[resultats.dem](#)

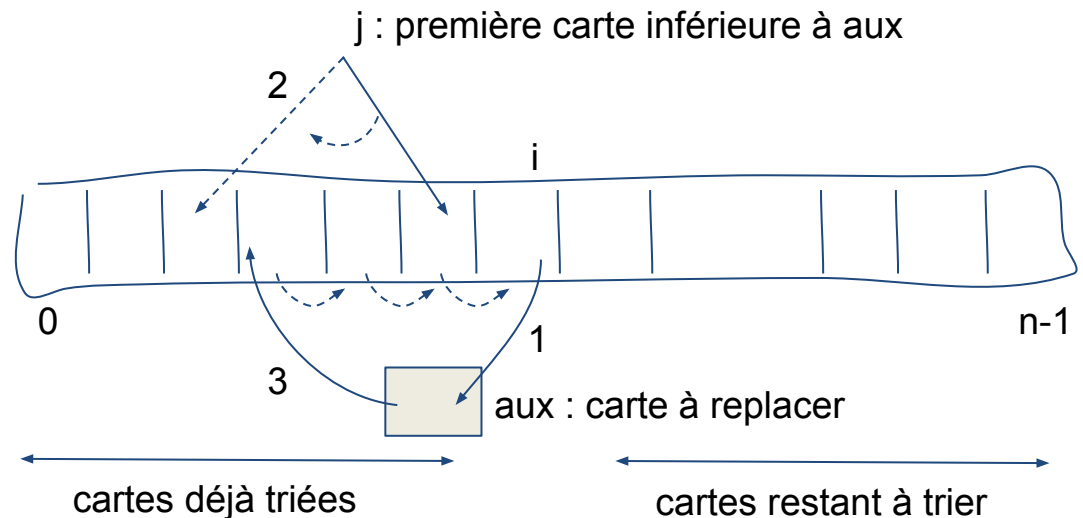
Tri Insertion

Le tri du joueur de cartes...

Opération de base :
comparaison entre
éléments de A

```
i = 1;
while (i < n) {
    aux = A[i] ;
    j = i - 1;
    while (j >= 0 && A[j] > aux) {
        A[j+1] = A[j] ;
        j-- ;
    }
    A[j+1] = aux ;
    i++;
}
```

- Que fait ce code ?
- Quelle complexité ?
 - meilleur cas ?
 - pire cas ?



Tri Insertion

Opération de base :
comparaison entre
éléments de A

```
i = 1;
```

```
while (i < n) {
```

```
    aux = A[i] ;
```

```
    j = i - 1;
```

```
    while (j >= 0 && A[j] > aux) {
```

```
        A[j+1] = A[j] ;
```

```
        j-- ;
```

```
    }
```

```
    A[j+1] = aux ;
```

```
    i++;
```

```
}
```

← n-1 itérations

Meilleur cas : 1 comparaison

$$\sum_{i=1 \rightarrow n-1} 1 = n-1 \Rightarrow \mathbf{O(n)}$$

← Pire cas : i comparaisons

$$\sum_{i=1 \rightarrow n-1} i = (n-1).n/2$$

$$\Rightarrow \mathbf{O(n^2)}$$

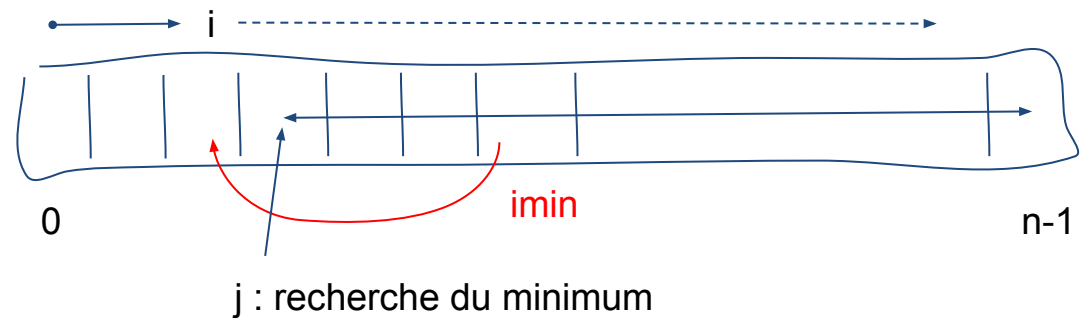
Tri Sélection

Les cartes sont sur la table...

Opération de base :
comparaison entre
éléments de A

```
for (i = 0; i < n-1; i++) {  
    imin = i;  
    for (j = i+1; j < n; j++)  
        if (A[j] < A[imin])  
            imin = j;  
    if (i != imin){  
        aux = A[imin];  
        A[imin] = A[i];  
        A[i] = aux;  
    }  
}
```

- Que fait ce code ?
- Quelle complexité ?



Tri Sélection

Quelle complexité ?

Opération de base :
comparaison entre
éléments de A

```
for (i = 0; i < n-1; i++) {
```

← n-1 itérations

```
    imin = i;
```

```
    for (j = i+1; j < n; j++)
```

```
        if (A[j] < A[imin])
```

← n-i-1 comparaisons

```
            imin = j;
```

```
    if (i != imin) {
```

```
        aux = A[imin];
```

```
        A[imin] = A[i];
```

```
        A[i] = aux;
```

```
    }
```

```
}
```

$$\begin{aligned}\sum_{i=0 \rightarrow n-2} (n-1)-i &= \sum_{i=0 \rightarrow n-1} (n-1)-i \\ &= \sum_{i=0 \rightarrow n-1} i \\ &= (n-1).n/2\end{aligned}$$

⇒ $O(n^2)$

Programmes de test

test_utils.h

- Dans chaque fonction, compter les comparaisons et opérations représentatives de la complexité de l'algorithme mis en oeuvre
 - stats.nbComparisons ++;
 - stats.nbOperations ++;

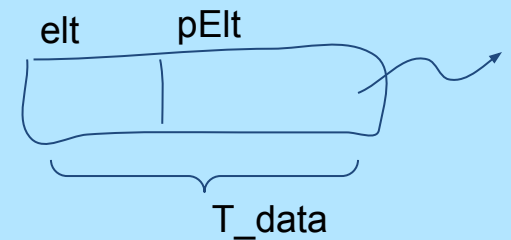
```
typedef struct {  
    unsigned long long  nbComparisons; // nombre de comparaisons effectuées  
    unsigned long long  nbOperations;  // nombre d'opérations effectuées  
    clock_t duration_clock_t; // durée de la fonction en nombre de tops d'horloge  
} T_stat;  
extern T_stat stats;
```

Programmes de test

test_utils.h

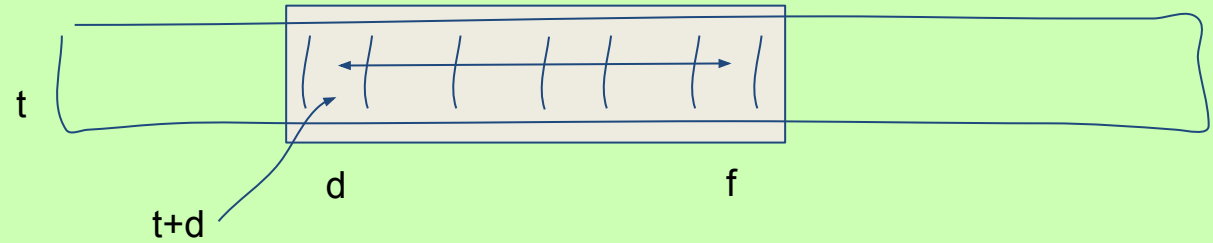
- Toutes les fonctions à tester doivent respecter ces contraintes :
 - Arg 1 : données du problème : **T_data** : **T_elt** + pointeur sur **T_elt**
 - Arg 2 : **entier n** : taille du problème considéré
 - Valeur de retour : **T_data** : **T_elt** + pointeur sur **T_elt**
- Exemple : **T_data** foo(**T_data** d, int n)

```
typedef struct {  
    T_elt elt; // un elt  
    T_elt * pElt; // ET un pointeur sur T_elt (adresse d'un T_elt ou d'un tableau de T_elt)  
} T_data;  
  
T_data genData(T_elt e, T_elt * pE); // renvoie une structure T_data initialisée
```



Utilisation des arguments

T_data



```
int foo (T_elt t[], int d, int f) {
```

```
    // Comment appeler la fonction foo2 pour qu'elle manipule le tableau  
    t entre les indices d et f compris ?
```

```
    foo2(genData(0,t+d), f-d+1);
```

```
    // On pouvait aussi utiliser &(t[d])
```

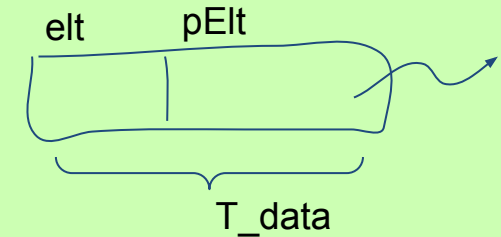
```
}
```

```
T_data foo2(T_data d, int n) {
```

```
    // Comment renvoyer la valeur et l'adresse de la case i ?
```

```
    return genData(d.pElt[i], d.pElt+i);
```

```
}
```



Test_Fn

Test_FnV2

```
T_mode m[] = {  
    {MODE_TAB_ORDONNE, "ordonne", 0, 1},    // tableau initialement ordonné (meilleur cas)  
    {MODE_TAB_ALEATOIRE, "aleatoire", 0, 1}, // tableau initialement organisé aléatoirement  
    {MODE_TAB_INVERSE, "inverse", 0, 1},    // tableau initialement ordonné à l'envers (pire cas)  
    {MODE_EVAL_X, "x=2.0", 2.0, 0} // évaluation de la fonction à tester au point X = 2.0  
    ...  
};
```

void Test_Fn (const char * nom, T_pFToTest fn, T_elt Table[], int nmax, T_mode mode)

- Exécute **fn** sur des jeux de données de taille croissante jusqu'à **nmax**, organisés suivant le **mode** souhaité
- Affiche les **indicateurs de complexité** à l'écran
- Ex: **Test_Fn("TRI INSERTION", insertSort, tab, MAX_ELT/10, m[MODE_TAB_ORDONNE])**
 - Appelle la fonction **insertSort** en lui fournissant des tableaux **déjà ordonnés**, de taille croissante, jusqu'à la taille **MAX_ELT/10**

void Test_FnV2 (const char * nom, T_pFToTest fn, T_elt Table[], int nmax, T_mode mode)

- Idem mais produit des **courbes au format png** (utilisation de **gnuplot**)
- Ex: **Test_FnV2("TRI INSERTION", insertSort, tab, 512, m[MODE_TAB_ALEATOIRE])**
 - Appelle la fonction **insertSort** en lui fournissant des tableaux **organisés aléatoirement**, de taille croissante, jusqu'à la taille **512**



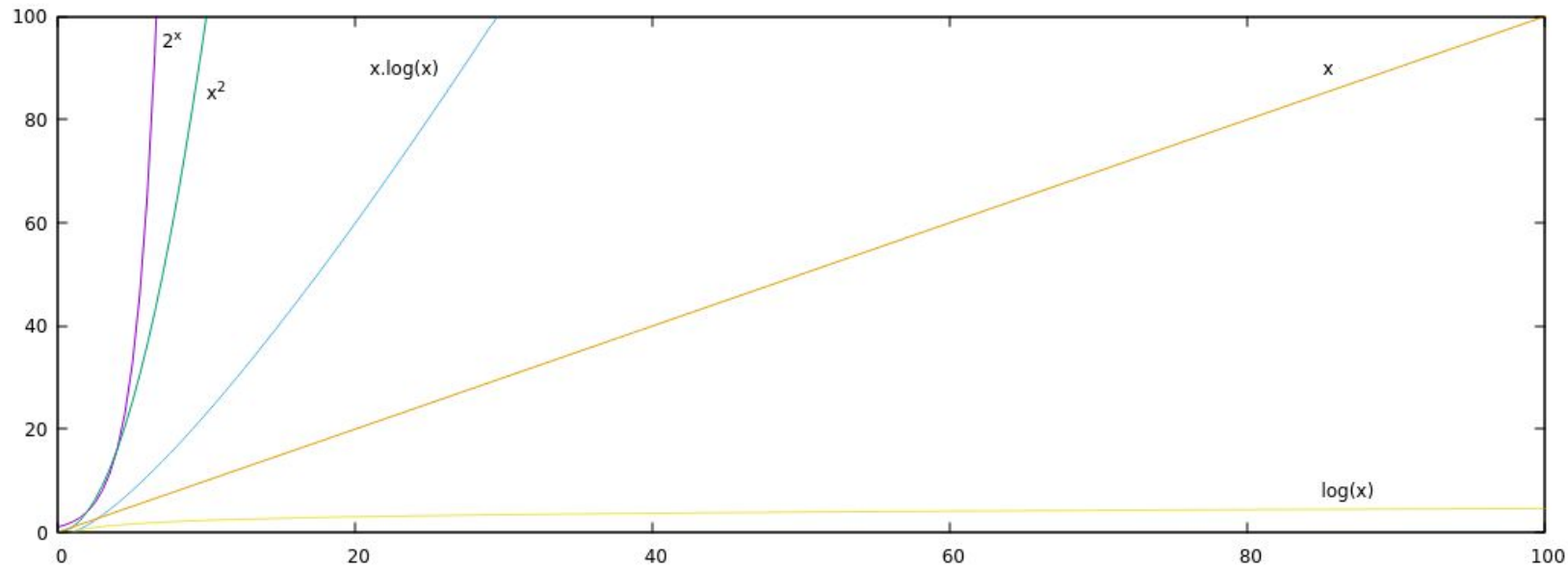
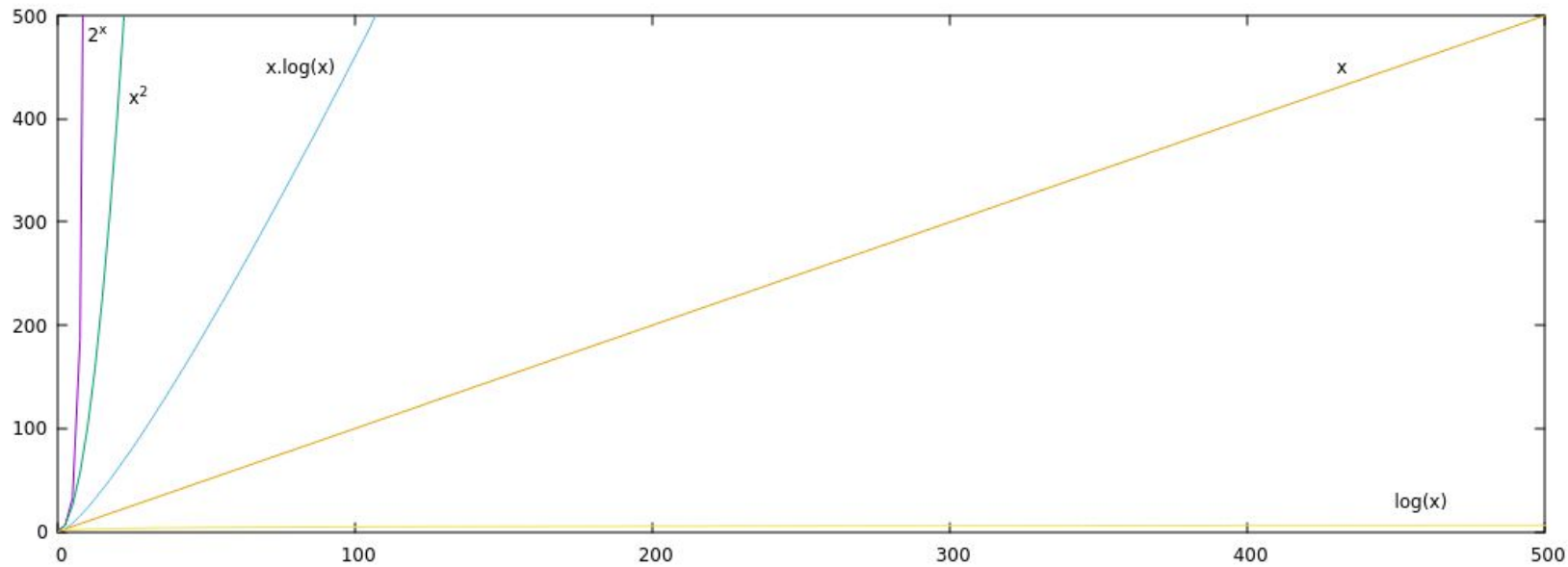
Classes de Complexité

Classes de complexité

- Lors de l'analyse de complexité, on se ramène généralement aux classes suivantes, présentées par complexité croissante
- Complexité **logarithmique** : Coût en $\Theta(\log n)$
 - Exemple : Recherche dichotomique dans un tableau ordonné de n éléments, puissance rapide
- Complexité **linéaire** : Coût en $\Theta(n)$
 - Exemple : Calcul du produit scalaire de deux vecteurs de taille n , recherche dans un tableau non trié

Classes de complexité

- Complexité **quasi-linéaire** : coût en $\Theta(n \log n)$
 - Exemple : Tri par fusion, tri rapide, tri par tas
- Complexité **polynomiale** : coût en $\Theta(n^k)$ avec $k > 1$
 - Exemple : Multiplication de deux matrices carrées d'ordre n : $\Theta(n^3)$
 - Complexité quadratique : $\Theta(n^2)$
 - Exemple : tris “lents”
- Complexité **exponentielle** : coût en $\Theta(a^n)$ avec $a > 1$
 - Exemple : Tours de Hanoï

$n = 0..100$  $n = 0..500$ 



Diviser pour régner

Arbre récursif
Théorème Maître

Diviser pour régner

- Principe de conception d'algorithmes et de décomposition d'un problème complexe en sous-problèmes plus faciles à traiter
- Algorithme **récuratif**
- Trois étapes à chaque niveau de récursivité :
 - **Diviser** le problème en un certain nombre de sous-problèmes
 - **Régner** sur les sous-problèmes en les **résolvant récursivement**
 - Si la taille d'un sous-problème est assez réduite, on peut le résoudre directement
 - **Combiner** les solutions des sous-problèmes en une solution complète pour le problème initial

Diviser pour régner : Équation de récurrence

- « Équation de coût »
- $T(n) = a T(n/b) + f(n)$
- a : nombre de sous-problèmes
- b : facteur de division du problème en sous-problèmes
- $f(n)$: coût des opérations de division / combinaison

Arbre récursif

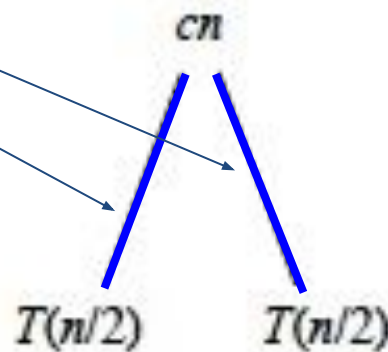
$$T(1) = c$$

$$T(n) = 2 T(n/2) + c.n, \text{ pour } n > 1$$

Depuis chaque noeud,
on crée deux nouveaux
noeuds pour changer
de niveau

On remplace n par
 $n/2$ en descendant à
chaque sous-niveau

Le coût de division
du problème (cn)
est rapporté en
chaque noeud



Arbre récursif

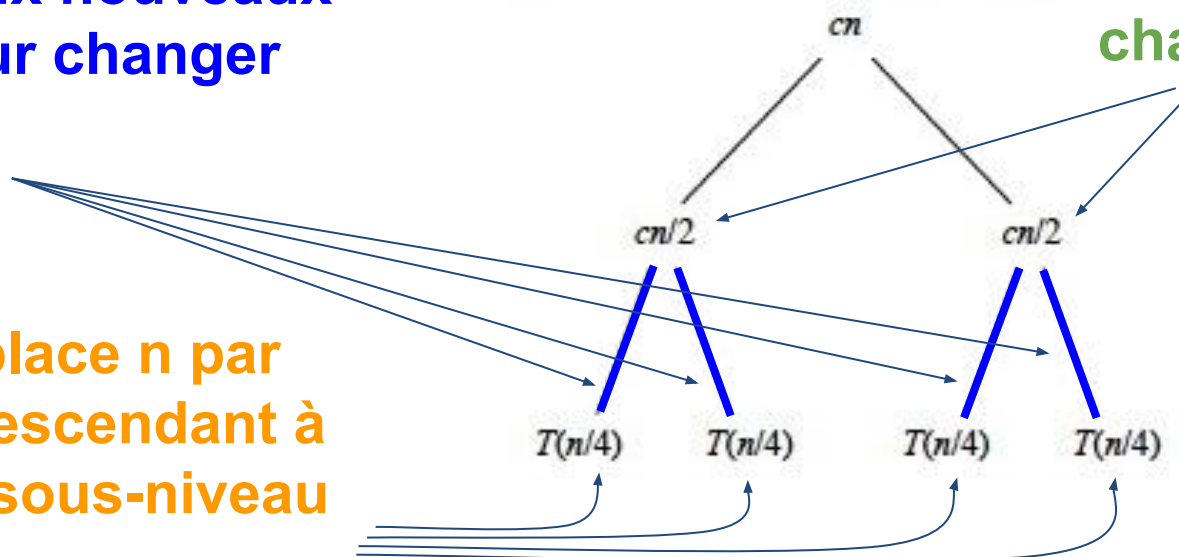
$$T(1) = c$$

$$T(n) = 2 T(n/2) + c.n, \text{ pour } n > 1$$

Depuis chaque noeud,
on crée deux nouveaux
noeuds pour changer
de niveau

On remplace n par
 $n/2$ en descendant à
chaque sous-niveau

Le coût de division
du problème (cn)
est rapporté en
chaque noeud



$$a = 2 ; b = 2; \log_b(a) = 1$$

$$a^{\log_b(n)} = n^{\log_b(a)} = n$$

Arbre récursif

$$T(1) = c$$

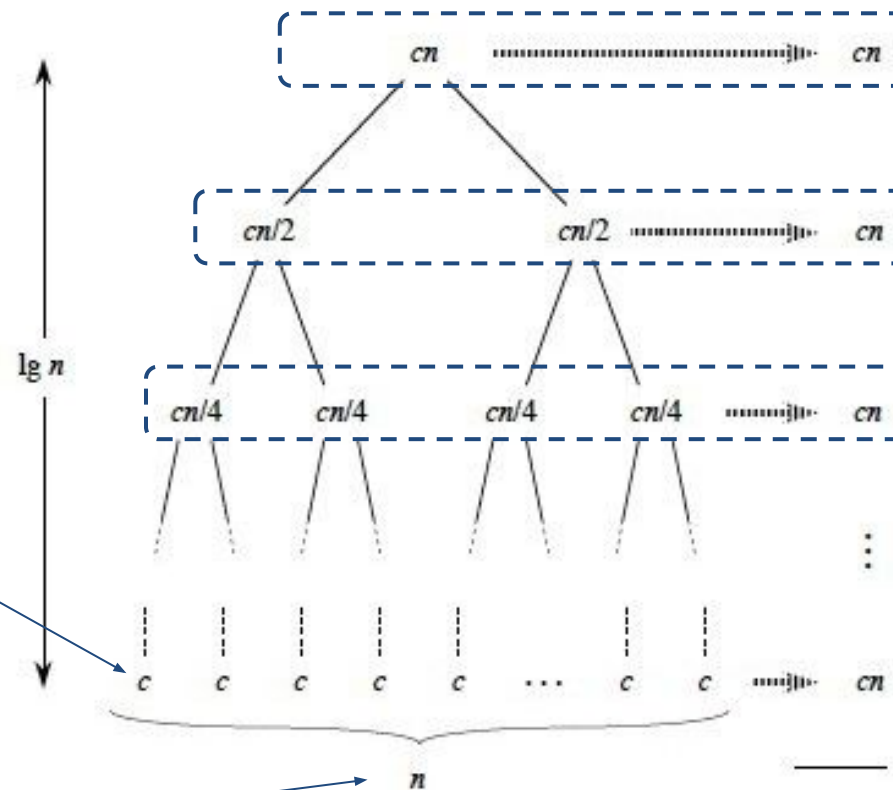
$$T(n) = 2 T(n/2) + c.n, \text{ pour } n > 1$$

On continue ainsi, en faisant apparaître le nombre de niveaux nécessaires pour arriver aux feuilles

On somme les coûts pour chaque niveau

On calcule le coût des feuilles

On compte le nombre de feuilles



On calcule le **coût total** :
coût des étapes
intermédiaires
+ coût des feuilles

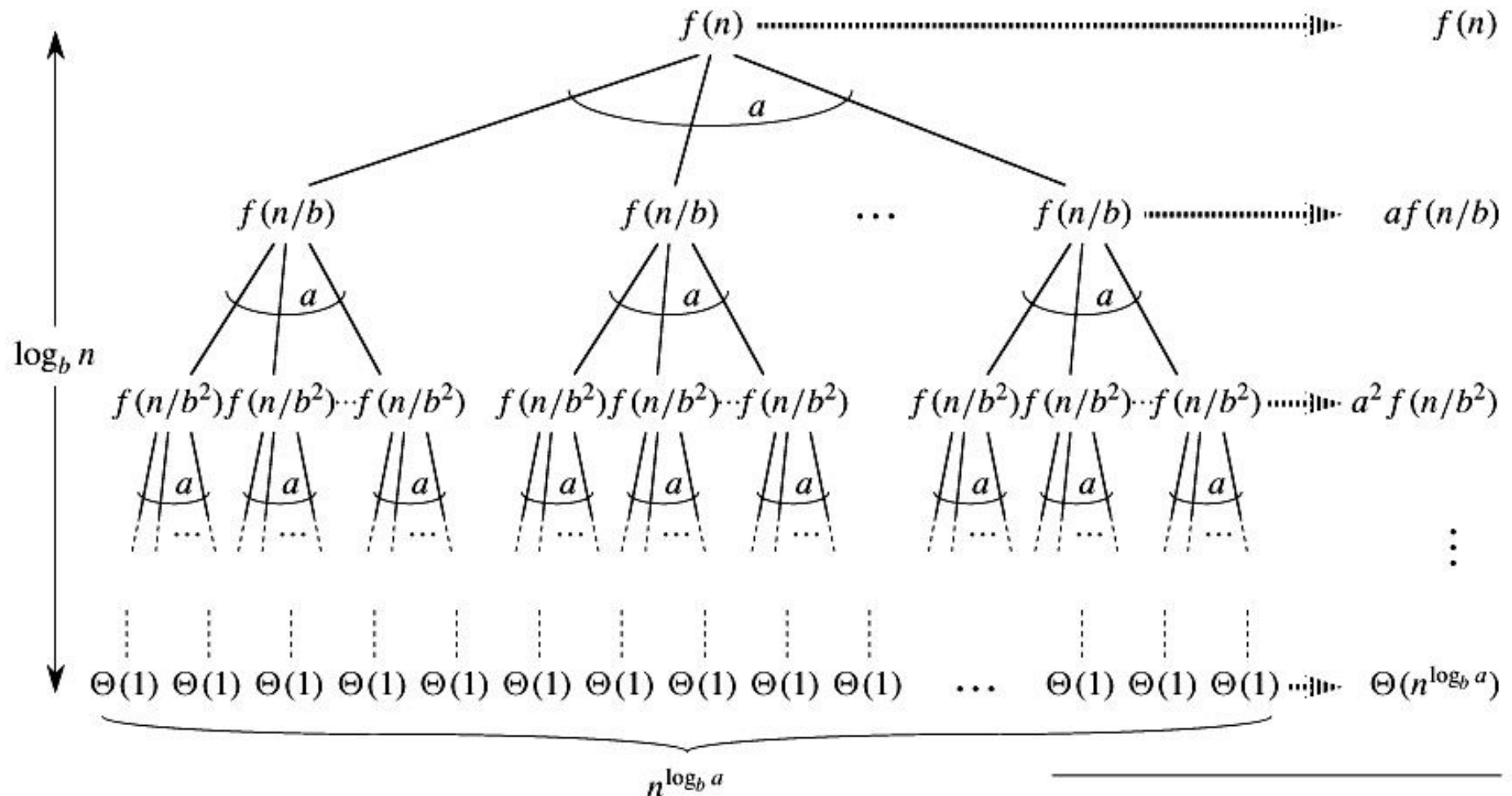
$$\text{Total: } cn \lg n + cn$$

Théorème maître

Attention : il existe des cas qui ne peuvent pas être réglés par le théorème !

- S'applique à des **équations de récurrence** de la forme :
 $T(n) = a T(n/b) + f(n)$ avec $a \geq 1$, $b > 1$
- Trois cas, selon l'importance du **surcoût induit par $f(n)$**
- Ce surcoût se mesure en fonction de la croissance asymptotique de $f(n)$ par rapport à n^c
 - $c = \log_b(a)$ représente l'*exposant critique*
- Sens physique :
 - $\log_b(n)$ = hauteur de l'arbre récursif
 - $n^c = n^{\log_b(a)} = a^{\log_b(n)}$ = nombre de feuilles de l'arbre récursif
 - Coût de tous les cas de base = $\Theta(n^{\log_b(a)}) = \Theta(n^c)$

Théorème maître : Sens physique



$$\text{Total: } \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$$

Théorème maître : sélection de la règle :

On compare la croissance de $n^c = n^{\log_b(a)}$ à celle de $f(n)$

- Règle 1 : n^c croît plus vite que $f(n)$
 - Si $f(n) = O(n^{\log_b(a-\epsilon)})$ avec $\epsilon > 0$ $\equiv f(n) = O(n^d)$ avec $d < c$
 - Alors, $T(n) = \Theta(n^c)$
- Règle 2 : n^c et $f(n)$ ont des croissances équivalentes
 - Si $f(n) = \Theta(n^c \cdot \log^k(n))$ avec $k \geq 0$ (généralement, $k=0$)
 - Alors, $T(n) = \Theta(n^c \cdot \log^{k+1}(n))$
- Règle 3 : n^c croît moins vite que $f(n)$, et $f(n)$ satisfait une *condition de régularité*
 - Si $f(n) = \Omega(n^{\log_b(a+\epsilon)})$ avec $\epsilon > 0$ $\equiv f(n) = \Omega(n^d)$ avec $d > c$
 - Et $\exists k < 1$ tq. $a f(n/b) \leq k f(n)$ pour n suffisamment grand
 - Alors, $T(n) = \Theta(f(n))$

Théorème maître : exemple 1

$$T(n) = 16T(n/4) + n$$

- $a = 16,$
- $b = 4$
- $c = \log_b(a)$
 $= 2$
- $n^c = n^2$
- $f(n) = n$
 $= \Theta(n)$
 $= O(n)$

Théorème maître : sélection de la règle :

On compare la croissance de $n^c = n^{\log_b(a)}$ à celle de $f(n)$

- Règle 1 : n^c croît plus vite que $f(n)$
 - Si $f(n) = O(n^{\log_b(a)-\epsilon})$ avec $\epsilon > 0$ $\equiv f(n) = O(n^d)$ avec $d < c$
 - Alors, $T(n) = \Theta(n^c)$
- Règle 2 : n^c et $f(n)$ ont des croissances équivalentes
 - Si $f(n) = \Theta(n^c \cdot \log^k(n))$ avec $k \geq 0$ (généralement, $k=0$)
 - Alors, $T(n) = \Theta(n^c \cdot \log^{k+1}(n))$
- Règle 3 : n^c croît moins vite que $f(n)$, et $f(n)$ satisfait une *condition de régularité*
 - Si $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ avec $\epsilon > 0$ $\equiv f(n) = \Omega(n^d)$ avec $d > c$
 - Et $\exists k < 1$ tq. $a f(n/b) \leq k f(n)$ pour n suffisamment grand
 - Alors, $T(n) = \Theta(f(n))$

Théorème maître : exemple 1

- $T(n) = 16T(n/4) + n$
 - $a = 16, b = 4$
 - $c = \log_b(a) = 2, n^c = n^2$
 - $f(n) = n = \Theta(n) = O(n)$
- Cas n^0 : $f(n) = O(n^{\log_4(16-\epsilon)})$ en prenant $\epsilon = 12$
- Alors $T(n) = \Theta(n^2)$

- Règle 1 : n^c croît plus vite que $f(n)$
 - $f(n) = O(n^{\log_b(a-\epsilon)})$ avec $\epsilon > 0$
 - Alors, $T(n) = \Theta(n^c)$

Théorème maître : exemple 2

$$T(n) = T(n/2) + 1$$

- $a = 1$
- $b = 2$
- $c = \log_b(a)$
 $= 0$
- $n^c = 1$
- $f(n) = 1$
 $= \Theta(1)$

Théorème maître : sélection de la règle :

On compare la croissance de $n^c = n^{\log_b(a)}$ à celle de $f(n)$

- Règle 1 : n^c croît plus vite que $f(n)$
 - Si $f(n) = O(n^{\log_b(a)-\varepsilon})$ avec $\varepsilon > 0$ $\equiv f(n) = O(n^d)$ avec $d < c$
 - Alors, $T(n) = \Theta(n^c)$
- Règle 2 : n^c et $f(n)$ ont des croissances équivalentes
 - Si $f(n) = \Theta(n^c \cdot \log^k(n))$ avec $k \geq 0$ (généralement, $k=0$)
 - Alors, $T(n) = \Theta(n^c \cdot \log^{k+1}(n))$
- Règle 3 : n^c croît moins vite que $f(n)$, et $f(n)$ satisfait une *condition de régularité*
 - Si $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$ avec $\varepsilon > 0$ $\equiv f(n) = \Omega(n^d)$ avec $d > c$
 - Et $\exists k < 1$ tq. $a f(n/b) \leq k f(n)$ pour n suffisamment grand
 - Alors, $T(n) = \Theta(f(n))$

Théorème maître : exemple 2

- $T(n) = T(n/2) + 1$
 - $a = 1, b = 2$
 - $c = \log_b(a) = 0, n^c = 1$
 - $f(n) = 1 = \Theta(1)$
- Cas numéro 2 en prenant $k = 0$
 - Alors $T(n) = \Theta(n^{\log_2(1)} \log^1(n)) = \Theta(\log(n))$

- Règle 2 : n^c et $f(n)$ ont des croissances équivalentes
 - $f(n) = \Theta(n^c \cdot \log^k(n))$ avec $k \geq 0$ (généralement, $k=0$)
 - Alors, $T(n) = \Theta(n^c \cdot \log^{k+1}(n))$

Théorème maître : exemple 3

$$T(n) = 3T(n/2) + n^2$$

- $a = 3$
- $b = 2$
- $c = \log_b(a)$
 $= \log_2(3)$
 ≈ 1.58
- $n^c = n^{1.58}$
- $f(n) = n^2$
 $= \Theta(n^2)$
 $= \Omega(n^2)$

Théorème maître : sélection de la règle :

On compare la croissance de $n^c = n^{\log_b(a)}$ à celle de $f(n)$

- Règle 1 : n^c croît plus vite que $f(n)$
 - Si $f(n) = O(n^{\log_b(a)-\epsilon})$ avec $\epsilon > 0$ $\equiv f(n) = O(n^d)$ avec $d < c$
 - Alors, $T(n) = \Theta(n^c)$
- Règle 2 : n^c et $f(n)$ ont des croissances équivalentes
 - Si $f(n) = \Theta(n^c \cdot \log^k(n))$ avec $k \geq 0$ (généralement, $k=0$)
 - Alors, $T(n) = \Theta(n^c \cdot \log^{k+1}(n))$
- Règle 3 : n^c croît moins vite que $f(n)$, et $f(n)$ satisfait une *condition de régularité*
 - Si $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ avec $\epsilon > 0$ $\equiv f(n) = \Omega(n^d)$ avec $d > c$
 - Et $\exists k < 1$ tq. $a f(n/b) \leq k f(n)$ pour n suffisamment grand
 - Alors, $T(n) = \Theta(f(n))$

Théorème maître : exemple 3

- $T(n) = 3T(n/2) + n^2$
 - $a = 3, b = 2$
 - $c = \log_b(a) = \log_2(3) \approx 1.58, n^c \approx n^{1.58}$
 - $f(n) = n^2 = \Theta(n^2) = \Omega(n^2)$
 - Cas $n^{\circ 3}$: $f(n) = \Omega(n^{\log_2(3+\varepsilon)})$ en prenant $\varepsilon = 1$
 - $3(n/2)^2 \leq k n^2$ en prenant $k = 3/4 < 1$
 - Alors $T(n) = \Theta(n^2)$
- Règle 3 : n^c croît moins vite que $f(n)$, et $f(n)$ satisfait une *condition de régularité*
 - $f(n) = \Omega(n^{\log_b(a+\varepsilon)})$ avec $\varepsilon > 0$
 - $\exists k < 1$ tq. $af(n/b) \leq k f(n)$ pour n suffisamment grand
 - Alors, $T(n) = \Theta(f(n))$

Théorème maître : bibliographie

- https://fr.wikipedia.org/wiki/Master_theorem
- [https://en.wikipedia.org/wiki/Master_theorem_\(analysis_of_algorithms\)](https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms))
- <https://people.csail.mit.edu/thies/6.046-web/master.pdf>
- <http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap04.htm>
- <https://www.cs.cornell.edu/courses/cs3110/2012sp/lectures/lec20-master/lec20.html>
- <http://www2.hawaii.edu/~nodari/teaching/s18/Notes/Topic-07.html>
- Algorithmique, 3è édition, Dunod, 2009
 - Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, et Clifford Stein
 - http://www.euroinformatica.ro/documentation/programming/!!!Algorithms_CORMEN!!!/DDU0027.html



Récurtivité

Développement de fonctions récursives
Expression des fonctions de coût

Récurtivité

- En C, il est possible de développer une fonction **qui s'appelle elle-même**
 - Fonction **récursive**
- Processus de développement similaire à la démonstration par récurrence en mathématiques
- **Pas forcément performant**, mais rapide à développer !
 - Sollicite beaucoup la... pile d'exécution !
- Attention aux boucles infinies !

Récurtivité

- En Maths
- Exprimer la propriété $P(n)$
 - Ordre de la récurrence : n
- Vérifier $P(0)$
- Faire l'hypothèse $P(n)$
 - Vérifier $P(n+1)$

- En Info
 - Identifier l'**ordre de la récurrence**
 - un paramètre entier, la taille d'une liste, d'un tableau...
- Coder **foo()** :
 - Coder le **cas de base** (si ordre d'appel = ordre initial)
 - Coder le **cas général** en utilisant des appels récursifs à **foo()**
 - Vérifier que l'ordre de la récurrence a **strictement diminué** entre l'appel initial et l'appel récursif !
 - Sinon... Boucle infinie !

Exponentiation

Opération de base :
multiplication entre
 T_elt

```
T_elt Puissance (T_elt x, int n) {  
    T_elt Result;
```

- Calcul de x^n
- Quelle complexité ?

```
    if (n == 0) return 1;
```

```
    for (Result = x; n > 1; n--) ← n-1 itérations
```

```
        Result *= x;           ← 1 multiplication
```

```
    return Result;
```

```
}
```



Exercice corrigé : **ex2** : Puissance

- Écrire la fonction Puissance en récursif
 - Quelle complexité ?
- Quel problème se présente lorsque l'on calcule des puissances trop grandes avec la version récursive ?

Exponentiation Rapide ("indienne")

Principe :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ (x^2)^{n/2} & \text{si } n \text{ pair} \\ x * (x^2)^{(n-1)/2} & \text{si } n \text{ impair} \end{cases}$$

```
T_elt PuissanceRapide(T_elt x, int n){
```

```
    T_elt Result = 1;
```

```
    while (n > 0) {
```

```
        if ((n % 2) == 1){
```

```
            Result *= x;
```

```
            n--;
```

```
        }
```

```
        x = x*x;
```

```
        n = n>>1; // équivalent à n=n/2
```

```
    }
```

```
    return Result;
```

```
}
```

← $\lfloor \log_2(n) \rfloor + 1$ itérations

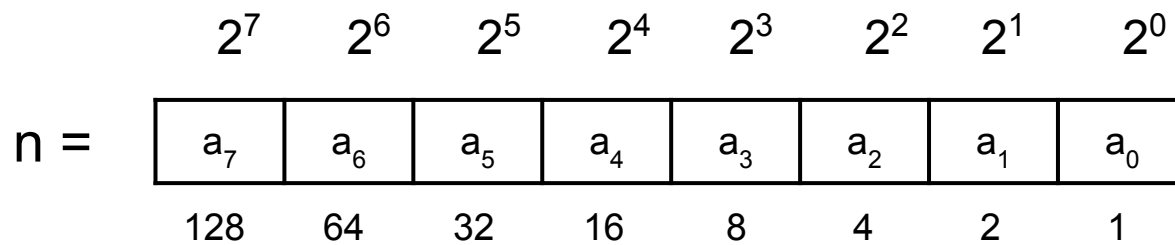
← 1 multiplication

← 1 multiplication

⇒ Coût en $\Theta(\log n)$

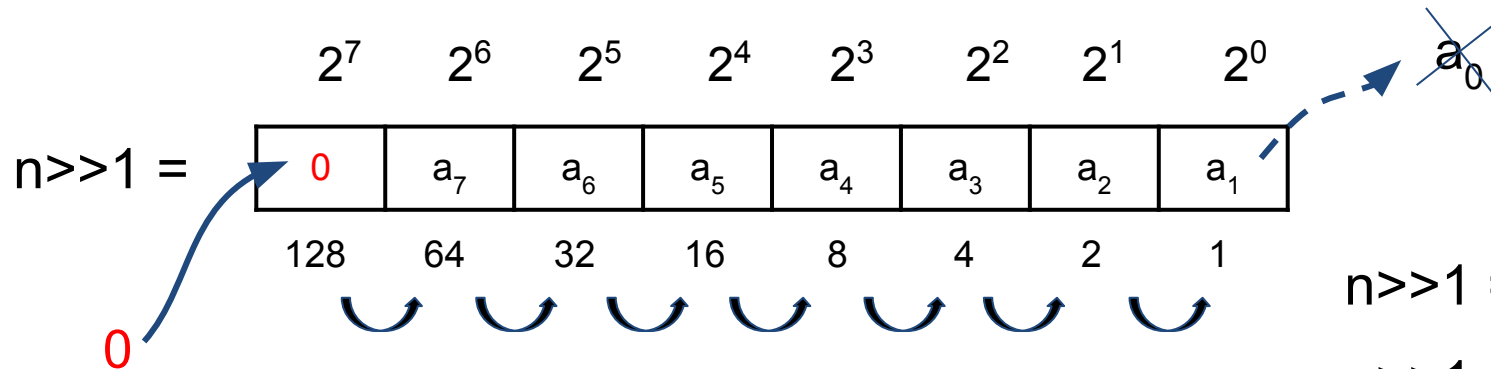
Opérateur >>

Décalage arithmétique ou logique



$$n = \sum_{k=0 \rightarrow 7} a_k 2^k$$

$$= a_0 + 2 \cdot \sum_{k=1 \rightarrow 7} a_k 2^{k-1}$$



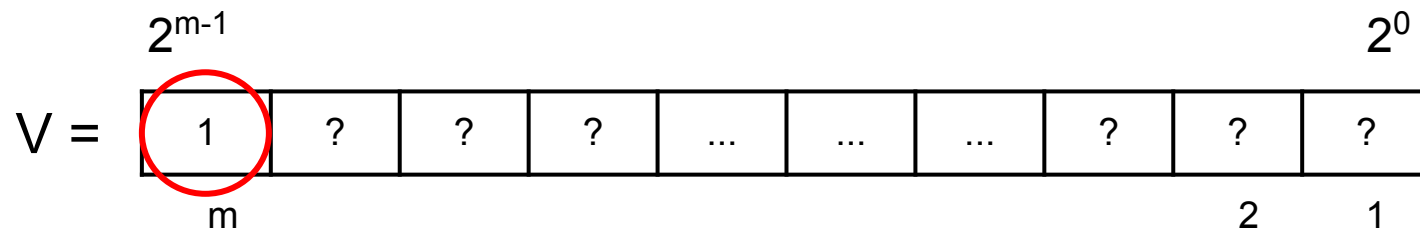
$$n \gg 1 = \sum_{k=1 \rightarrow 7} a_k 2^{k-1}$$

$$n \gg 1 = n/2$$

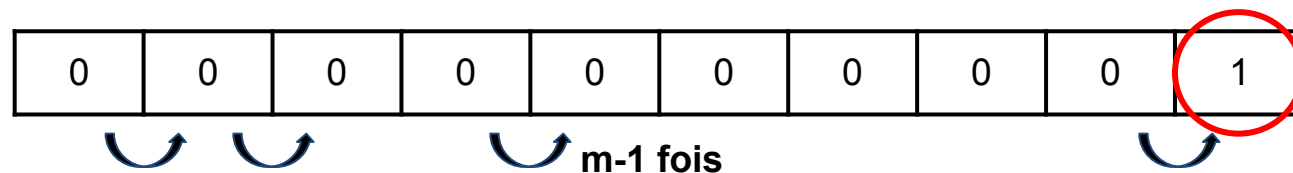
0 (décalage logique, si X est unsigned)

0 ou 1 (décalage arithmétique,
conservation du bit de signe, cf. capsule 5)

Pourquoi $\lfloor \log_2(n) \rfloor + 1$ itérations ?



- Considérons une valeur V représentée sur m bits
 - Il faut $m-1$ décalages vers la droite (opérateur $>>$) pour obtenir 1



- Quelles valeurs V se représentent sur m bits ?

- $2^{m-1} \leq V \leq 2^m - 1 < 2^m$

NB: $\sum_{k=0 \rightarrow m-1} 2^k = 2^m - 1$

- $m-1 \leq \log_2(V) < m$

- $m-1$ est le plus grand entier $\leq \log_2(V)$
- On le dénote $\lfloor \log_2(V) \rfloor$

$\lceil n \rceil$: plus petit entier supérieur ou égal à n
 $\lfloor n \rfloor$: plus grand entier inférieur ou égal à n

Pourquoi $\lfloor \log_2(n) \rfloor + 1$ itérations ?

- Décalage vers la droite = division euclidienne par la base
 - En binaire : division par 2
- $\lfloor \log_2(V) \rfloor$
 - Plus grand entier inférieur ou égal à $\log_2(V)$
 - Nombre de divisions successives de V par 2 permettant d'atteindre la valeur 1
- $\lceil \log_2(V) \rceil$
 - Plus petit entier supérieur ou égal à $\log_2(V)$
- $\lfloor \log_2(V) \rfloor + 1$
 - Nombre de divisions successives de V par 2 permettant d'atteindre la valeur 0
 - Nombre de symboles nécessaires pour représenter V en binaire
 - $\lfloor \log_{10}(V) \rfloor + 1$: nombre de chiffres nécessaires pour représenter V en base 10



Exercice corrigé : **ex3** :

Exponentiation Rapide ("indienne")

Principe :

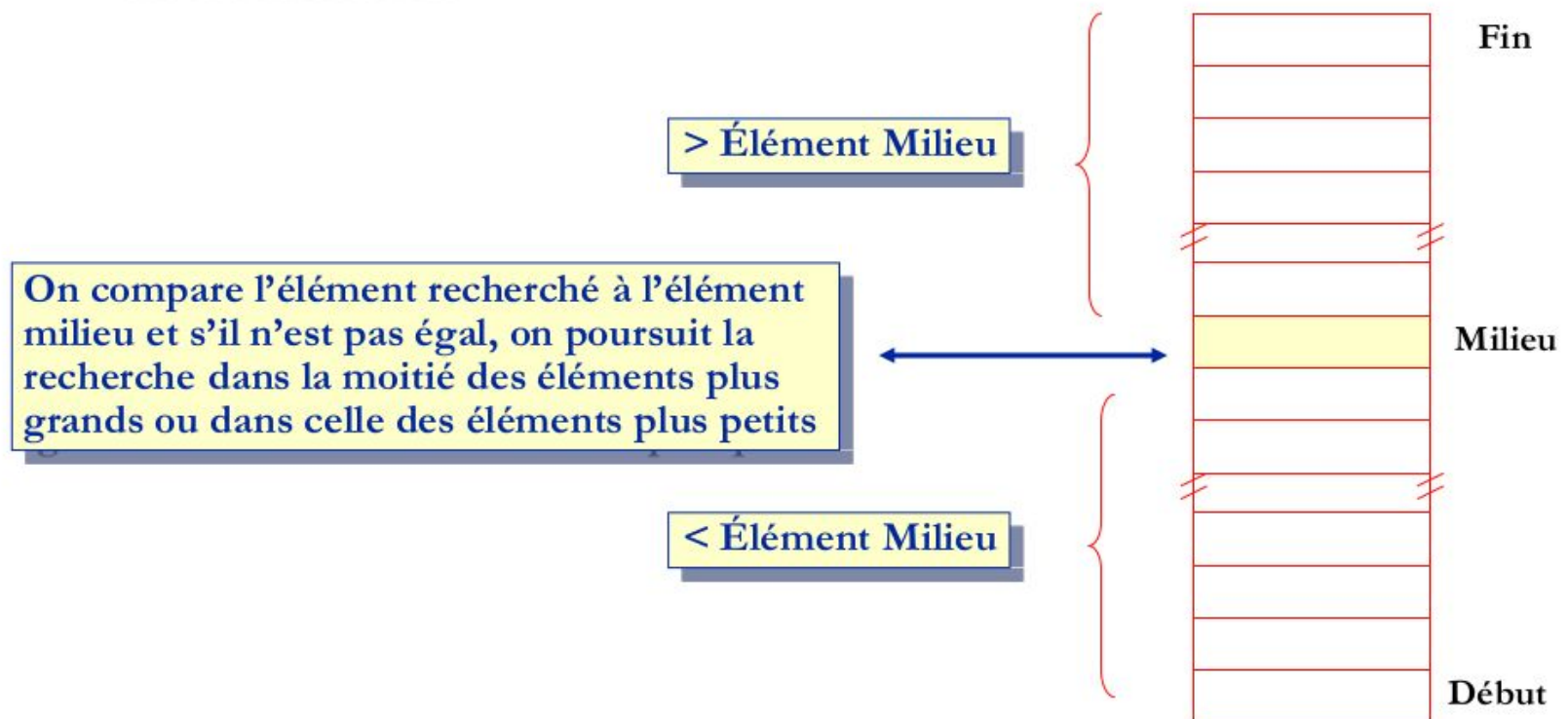
$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ (x^2)^{n/2} & \text{si } n \text{ pair} \\ x * (x^2)^{(n-1)/2} & \text{si } n \text{ impair} \end{cases}$$

- Ecrire le code de l'exponentiation rapide en récursif
- Etudier le comportement de la fonction expérimentalement
- Ecrire l'équation de coût
- Résoudre cette équation

Recherche dichotomique

Opération de base :
comparaison de T_elt

- Recherche dans une table ordonnée
- A chaque étape, réduire de moitié l'intervalle de recherche



Recherche dichotomique

```
int RechercheDicho (T_elt T[], int n, T_elt e) {  
    int Debut = 0, Fin = n - 1, Milieu;  
    int Test;  
    while (Debut <= Fin) {  
        Milieu = (Debut + Fin) / 2;  
        Test = Comparaison(e, T[Milieu]);  
        if (Test == 0)  
            return Milieu;  
        if (Test < 0)  
            Fin = Milieu - 1;  
        else  
            Debut = Milieu + 1;  
    }  
    return -1; /* e n'est pas dans T */  
}
```

Fonction de comparaison
qui retourne 0 si égalité ou
une valeur positive ou
négative

Recherche dans la moitié
inférieure

Recherche dans la moitié
supérieure

```
int Comparaison(T_elt e1, T_elt e2)
int Comparaison(const T_elt * pE1, const T_elt
* pE2)
```

- Une “bonne” fonction de comparaison doit renvoyer un entier vérifiant les critères suivants :
 - 0 si les éléments e1 et e2 sont égaux
 - < 0 si e1 est avant e2 pour la relation d'ordre considérée
 - > 0 si e1 est après e2 pour la relation d'ordre considérée
- Cf. fonction standard de comparaison des chaînes de caractères :

```
int strcmp(const char *s1, const char *s2)
```

- Cf. argument **compar** dans le prototype de la fonction **qsort** :

```
void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const
void *, const void *));
```

Recherche dichotomique

- Combien d'appels à **Comparaison(e, T[Milieu])** ?
 - 1 dans le meilleur des cas
 - Égal au nombre de passages dans la boucle while
 - À chaque itération, on réduit l'intervalle de recherche de moitié
 - La taille initiale de l'intervalle vaut n
 - **$\lfloor \log_2(n) \rfloor$** est le nombre d'itérations permettant de réduire l'intervalle de recherche à 1
 - **$\lfloor \log_2(n) \rfloor + 1$** est le nombre d'itérations permettant de réduire l'intervalle de recherche à zéro
- ⇒ Coût en **$O(\log n)$**



Exercice corrigé : **ex4** : Recherche dichotomique en récursif

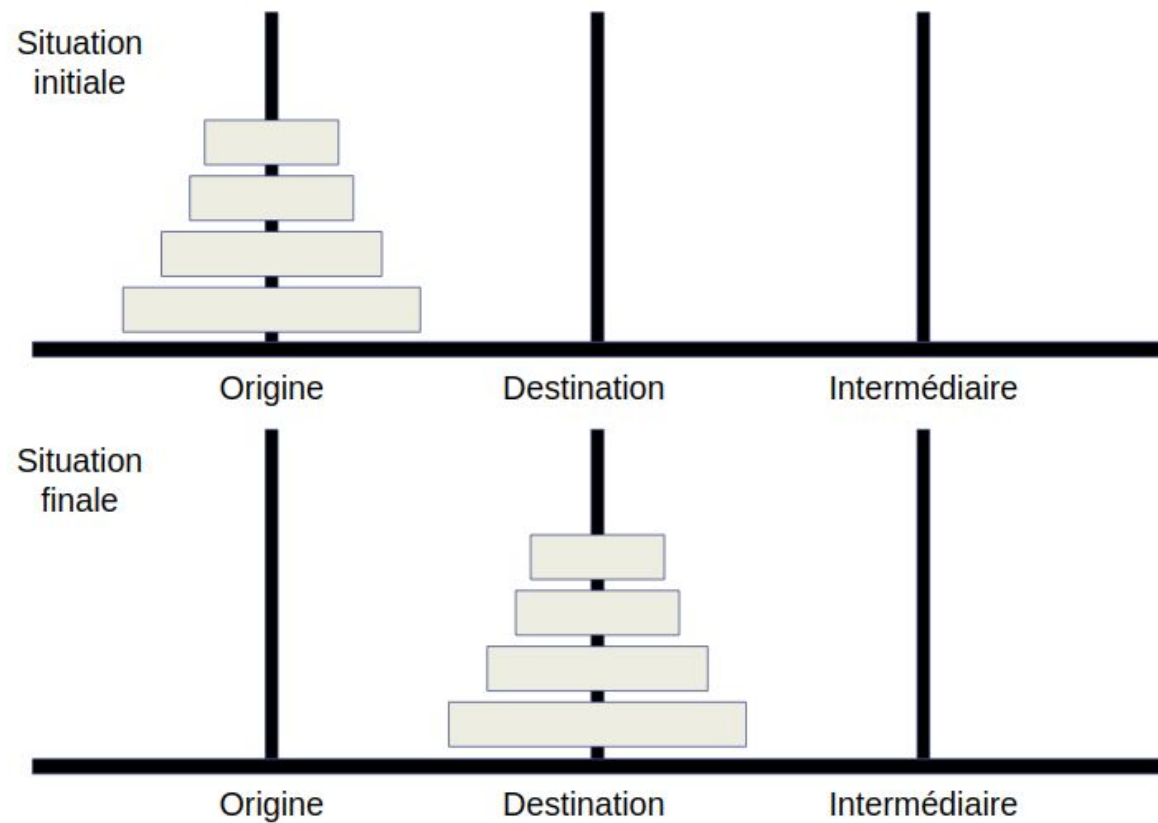
- Implémenter en récursif
- Etudier le comportement de la fonction expérimentalement
- Ecrire l'équation de coût, la résoudre
- Résolution par **substitution**

Résolution par **substitution**

- Procède par **intuition de la solution** et vérification par **substitution**
- Exemple : recherche dichotomique
 - $T(n) = 1 + T(n/2)$ pour $n > 0$; $T(0) = 0$; $T(1) = 1$
 - Intuition : $T(n) = \Theta(\log_2(n))$
- Hypothèse : $T(n) = a \cdot \log_2(n) + c$
 - Donc, $T(n/2) = a \cdot \log_2(n) - a + c$
- $$\begin{aligned} T(n) &= 1 + T(n/2) \\ \Leftrightarrow a \cdot \log_2(n) + c &= 1 + a \cdot \log_2(n) - a + c && \Leftrightarrow c = 1 - a + c \Leftrightarrow \mathbf{a = 1} \end{aligned}$$
- Puisque $T(1) = 1$, on déduit $\mathbf{c=0}$
- $\mathbf{T(n) = \log_2(n)}$

Tours de Hanoï

Opération de base :
déplacement d'un disque



- Déplacer un disque à la fois
- Ne jamais placer un disque sur un disque plus petit
- Objectif : déplacer tous les disques d'une colonne à une autre

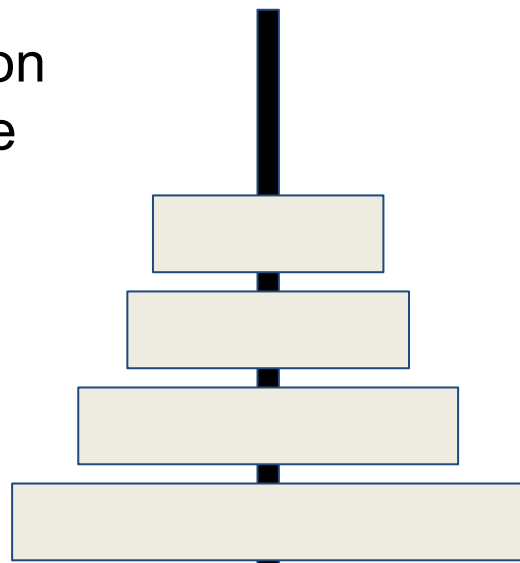


Exercice corrigé : **ex5** :

Tours de Hanoï

- Implémenter en récursif
- Etudier le comportement de la fonction expérimentalement
- Ecrire l'équation de coût
- Résolution **de manière itérative**

Situation
initiale

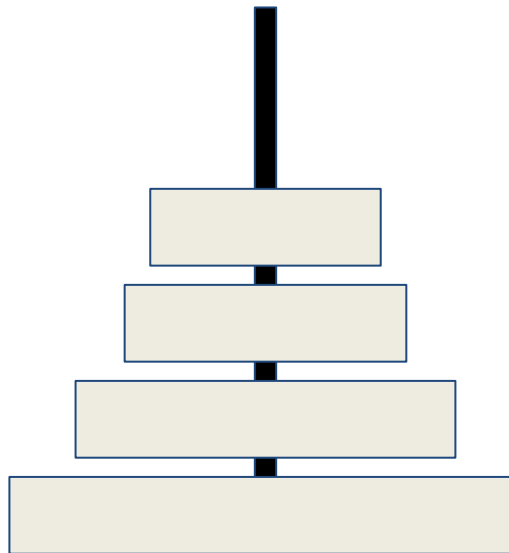


Origine

Destination

Intermédiaire

Situation
finale



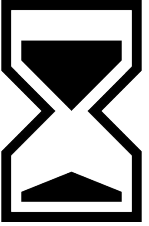
Origine

Destination

Intermédiaire

Résolution **itérative**

- Procède par développement et sommations
- Exemple : tours de Hanoï
 - $T(n) = 2 T(n-1) + 1$ pour $n > 0$; $T(0) = 0$
 - $T(n) = 2 (2 T(n-2) + 1) + 1 = 2^2 T(n-2) + 2 + 1$
 - $T(n) = 2^2 (2 T(n-3) + 1) + 2 + 1 = 2^3 T(n-3) + 2^2 + 2 + 1$
 - $T(n) = 2^i T(n-i) + 2^{i-1} + \dots + 2 + 1$
 - $T(n) = 2^n T(0) + 2^{n-1} + \dots + 2 + 1$ lorsque $i=n$
 - $T(n) = 2^{n-1} + \dots + 2 + 1 = 2^n - 1$
 - Complexité exponentielle : $\Theta(2^n)$



Tris Optimaux

Tris : vocabulaire

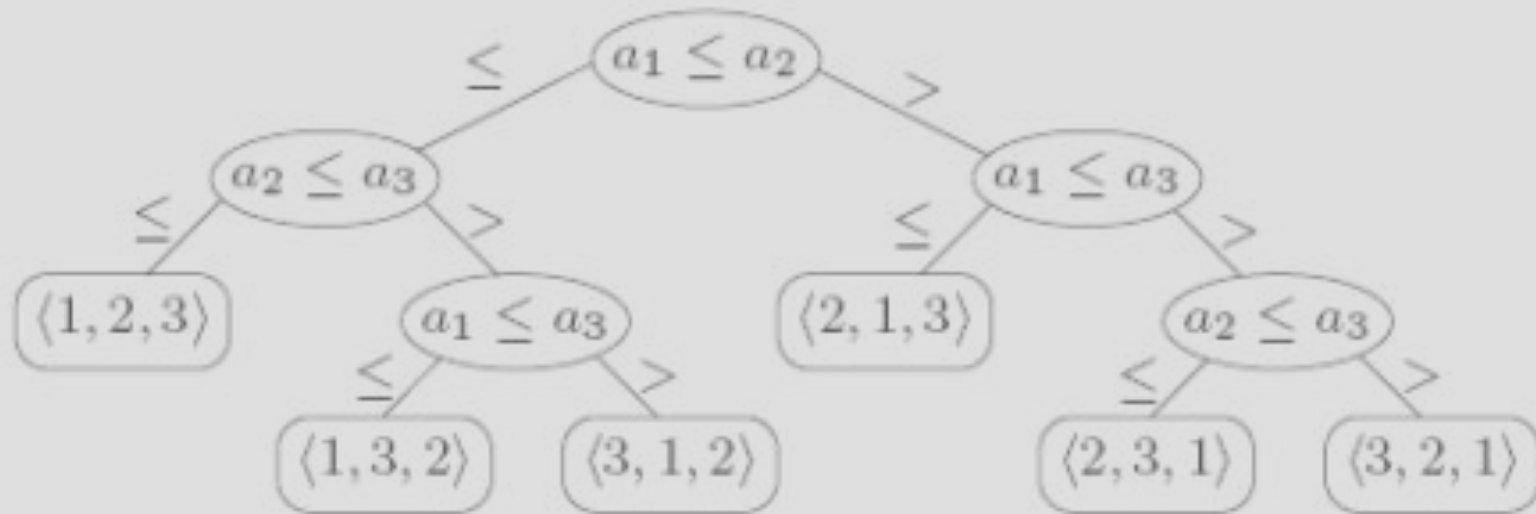
https://fr.wikipedia.org/wiki/Algorithme_de_tri

- **Clé** de tri : valeur associée aux éléments à trier qui sert de critère de comparaison entre ces éléments
- Tri **en place** : Tri qui modifie directement la structure qui est en train d'être triée, ne nécessite pas d'espace mémoire supplémentaire
- Tri **stable** : Tri qui préserve l'ordre initial des éléments que l'ordre considère comme égaux
- Tri **optimal** : Tri en **$O(n \log(n))$**
 - On peut montrer que la borne inférieure de complexité d'un algorithme de tri par comparaisons est **$n \cdot \log_2(n)$**

Tri Optimal : $O(n \log(n))$

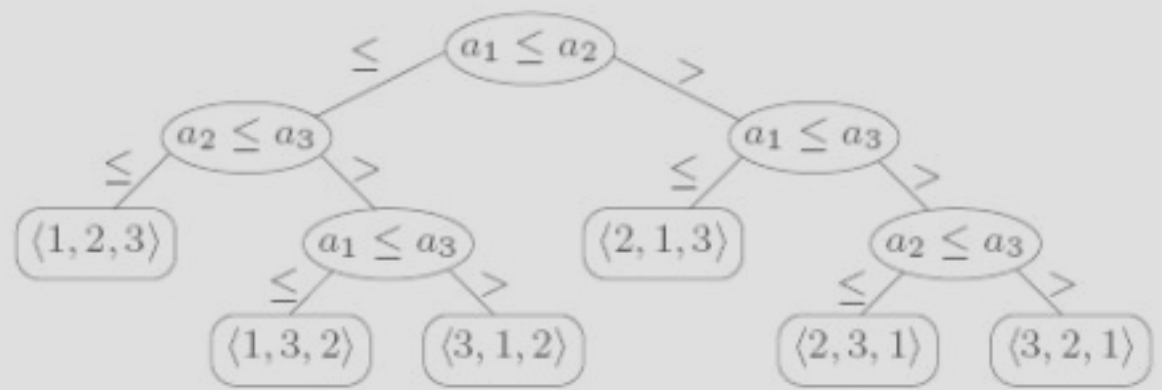
Intuition de la démonstration

- Arbre de décision correspondant au tri de 3 éléments



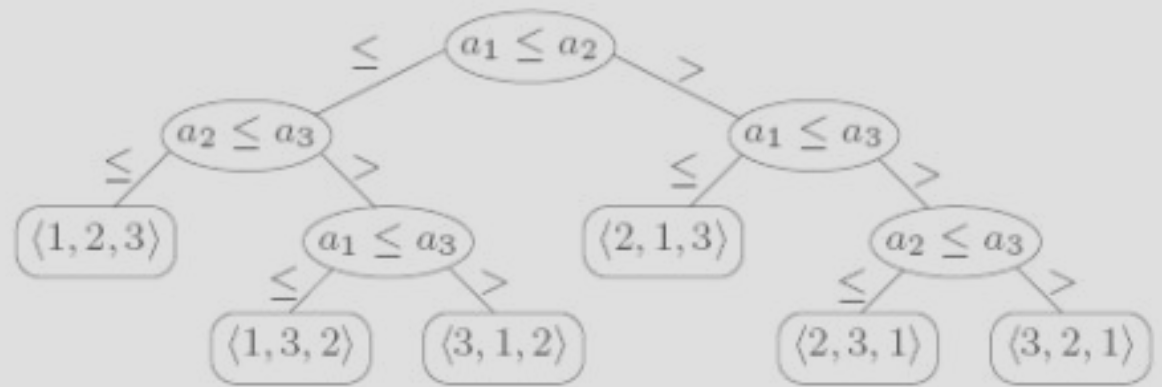
<http://icps.u-strasbg.fr/~vivien/Enseignement/Algo-2001-2002/Corrige-TD08.pdf>

Tri Optimal : Intuition



- Considérer l'arbre de décision correspondant au tri de n éléments
- Le résultat d'un tri correspond à une feuille de cet arbre de décision
- Le coût du tri, exprimé en **nombre de comparaisons**, est fonction de la **profondeur** de la feuille considérée
- Le **coût minimum** d'un tri capable de discriminer n éléments est donc égal à la **hauteur de l'arbre de décision**
 - Si on ne descend pas tout en bas, il y a des configurations différentes d'éléments qu'on ne pourra pas discriminer
- \Rightarrow Quelle est la hauteur minimale d'un arbre de décision de n éléments ?

Tri Optimal : Intuition



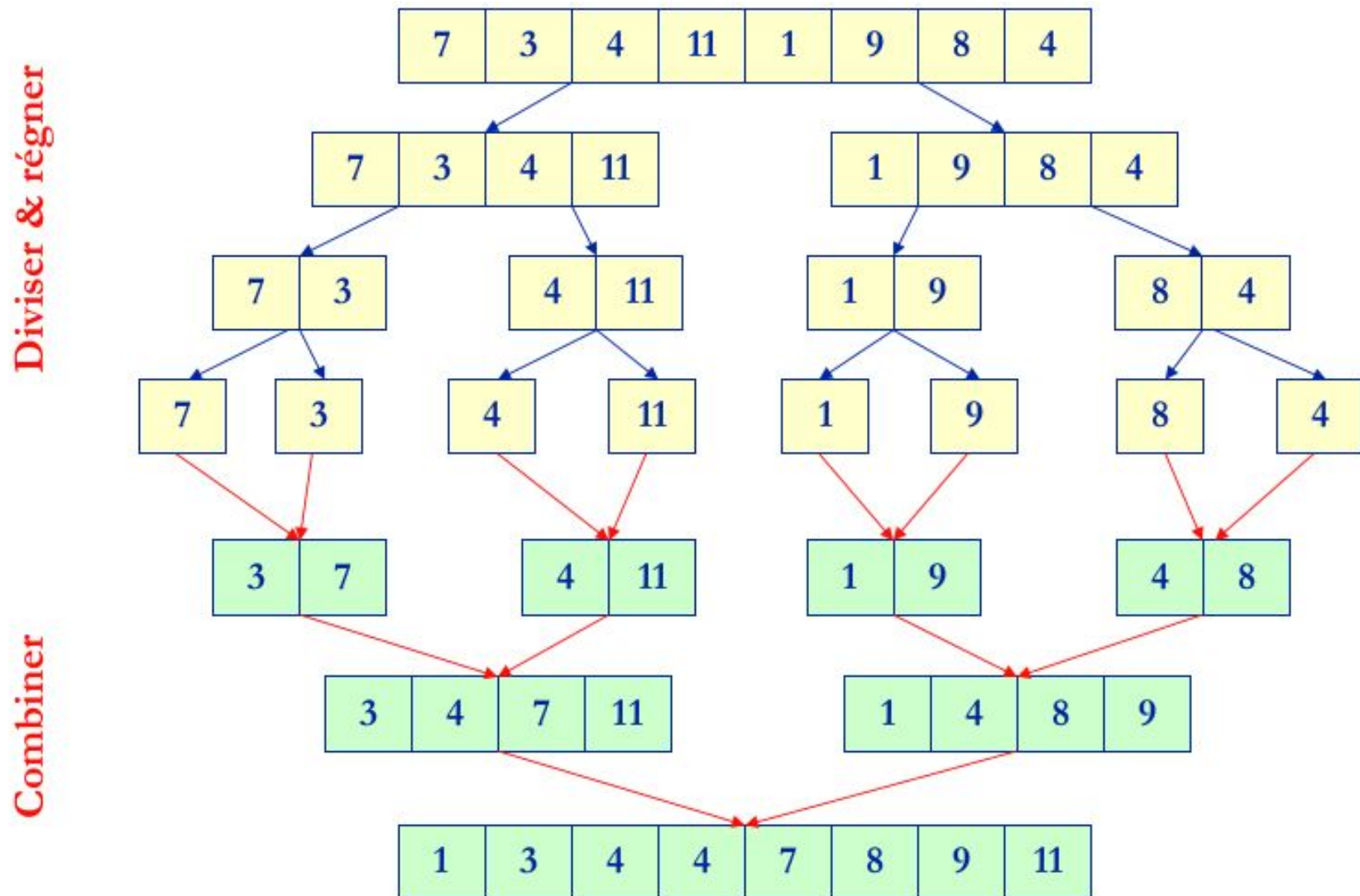
- Le nombre maximum de feuilles d'un arbre binaire de hauteur h est 2^h : $f \leq 2^h$
 - La hauteur minimale d'un arbre comportant f feuilles est $\log_2(f)$: $\log_2(f) \leq h$
- Un arbre de décision correspondant au tri de n éléments comporte $n!$ feuilles ($n!$ permutations possibles de n éléments)
- Sa hauteur minimale est donc $\log_2(n!) = \Theta(n \log_2(n))$
 - Cf. formule de Stirling : $n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \theta\left(\frac{1}{n}\right)\right)$



Tri fusion

- **Diviser** : diviser la séquence d'éléments à trier en deux sous-séquences d'éléments de tailles égales (à 1 élément près)
- **Régner** : trier chaque sous-séquence à l'aide du tri fusion (appels récursifs)
- **Combiner** : fusionner les sous-séquences triées pour produire la réponse triée

Tri fusion : exemple



Tri fusion : code C

```
void tri_fusion(T_elt t [], int debut, int fin) {
```

```
    int milieu;
```

```
    if (debut < fin)
```

```
    {
```

```
        milieu = (debut + fin)/2;
```

```
        tri_fusion(t, debut, milieu);
```

```
        tri_fusion(t, milieu + 1, fin);
```

```
        fusionner(t, debut, milieu, fin);
```

```
    }
```

```
}
```

Diviser

Régner

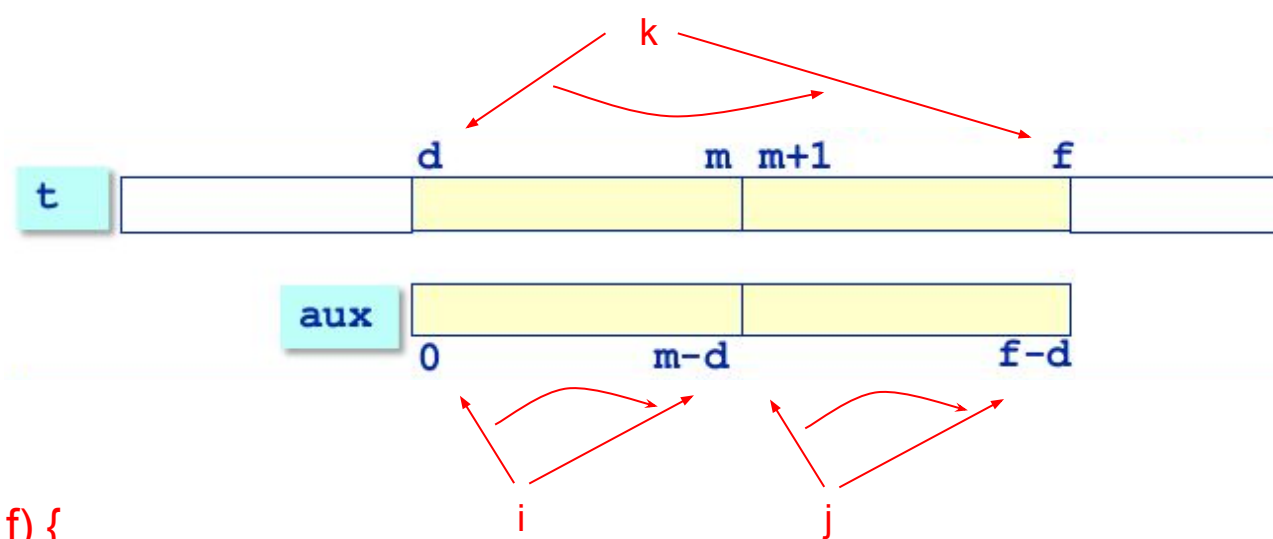
Combiner

Tri fusion

- Soit n la taille de l'ensemble à trier. L'opération de base est la comparaison entre éléments de l'ensemble
- Hypothèse : le coût de fusionner est en $\Theta(n)$
- $T(n) = 2 \times T(n/2) + \Theta(n)$ pour $n > 1$
 - $T(1) = 0$ et $T(0) = 0$

Fusionner

Opération de base :
comparaison de T_elt



```
void fusionner(T_elt t [], int d, int m, int f) {
```

```
    T_elt aux[f - d + 1]; // !! Allocation dynamique sur la pile (standard C99)
```

```
    int i, j, k;
```

```
    memcpy(aux, &t[d], (f - d + 1) * sizeof(T_elt)); // Copie des données à fusionner
```

```
    i = 0; j = m - d + 1; k = 0;
```

```
    while (i <= m - d && j <= f - d) {
```

```
        if (aux[i] <= aux[j])    t[d + k++] = aux[i++]; // aux[i] est plus petit : on le place dans t
```

```
        else                    t[d + k++] = aux[j++]; // aux[j] est plus petit : on le place dans t
```

```
    }
```

```
    for (; i <= m - d; t[d + k++] = aux[i++]); // le reste du tableau gauche
```

```
    for (; j <= f - d; t[d + k++] = aux[j++]); // le reste du tableau droit
```

```
}
```

f-d+1 itérations au maximum

Tri Fusion : Résolution itérative

$$T(n) = 2 \times T(n/2) + \Theta(n), \quad T(1) = 0, \\ T(0) = 0$$

- $T(n) = 2 T(n/2) + cn$
- $T(n) = 2 (2 T(n/4) + cn/2) + cn = 4 T(n/4) + 2 cn$
- $T(n) = 4 (2 T(n/8) + cn/4) + 2 cn = 2^3 T(n/2^3) + 3 cn$
- $T(n) = 2^i T(n/2^i) + i cn$
- Quand $i = \lfloor \log_2(n) \rfloor$:
- $T(n) = n T(1) + \lfloor \log_2(n) \rfloor cn = \lfloor \log_2(n) \rfloor cn$
- $T(n) = \Theta(n \log_2(n))$

Tri Fusion : Théorème Maître

$$T(n) = 2 \times T(n/2) + \Theta(n), \quad T(1) = 0, \\ T(0) = 0$$

- $a = 2, b = 2, c = \log_b(a) = 1, n^c = n$
- $f(n) = \Theta(n)$
- Règle 2 avec $k=0$
- $T(n) = \Theta(n \log(n))$

- Règle 2 : n^c et $f(n)$ ont des croissances équivalentes
 - $f(n) = \Theta(n^c \cdot \log^k(n))$ avec $k \geq 0$ (généralement, $k=0$)
 - Alors, $T(n) = \Theta(n^c \cdot \log^{k+1}(n))$

Tri fusion : conclusion

- Complexité en $\Theta(n \times \log(n))$
 - Quel que soit l'état initial de l'ensemble à trier
- Tri **stable**
- Ce **n'est pas un tri en place** si l'ensemble est un tableau
 - Nécessité d'une table auxiliaire
- Tri **en place si l'ensemble est une liste chaînée**
 - Pas d'échange de valeurs : seuls les pointeurs sont modifiés
 - Pas d'espace mémoire supplémentaire nécessaire

Exercice : **ex6** : TEA

- Implémenter tri fusion d'un tableau
- Implémenter tri fusion d'une liste avec affichage graphique des listes
- Etudier le comportement de vos algorithmes de tri



Tri rapide

- Quicksort : inventé par Tony Hoare en 1960
- **Diviser** : décomposer l'ensemble des éléments en deux partitions :
 - Une partition contenant les éléments dont la clé est inférieure à celle d'un élément particulier appelé **pivot**
 - Une partition contenant tous les éléments dont la clé est supérieure à celle du pivot
- **Régner** : appels récursifs à la procédure de tri pour chacune de ces partitions
- **Combiner** : inutile ici

Tri rapide : code C

```
void Tri_rapide( T_elt t[], int debut, int fin) {  
    int iPivot;  
    if (fin > debut) {  
        iPivot = Partitionner(t, debut, fin);  
        Tri_rapide(t, debut, iPivot - 1);  
        Tri_rapide(t, iPivot + 1, fin);  
    }  
}
```

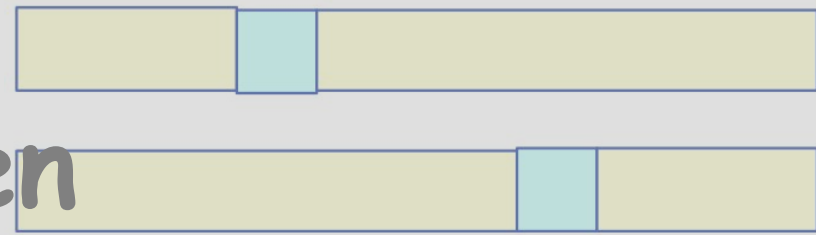
Diviser

Régner

Tri rapide : coûts

- Cas favorable : chaque partition contient $n/2$ éléments (à ± 1 près)
 - $T(n) = 2 T(n/2) + \Theta(n)$ pour $n > 1$, $T(0) = 0$, $T(1) = 0$
 - $T(n) = \Theta(n \log(n))$
- Cas défavorable : une partition vide, l'autre contenant tous les éléments sauf le pivot
 - $T(n) = T(n-1) + \Theta(n)$ pour $n > 1$, $T(0) = 0$, $T(1) = 0$
 - $T(n) = \Theta(n^2)$
- En moyenne : $T(n) = \Theta(n \log(n))$
 - $T(0) = 0$, $T(1) = 0$
 - $T(n) = n-1 + \frac{1}{n} \sum_{i=0 \rightarrow n-1} (T(i) + T(n-i-1))$ $n > 1$

Tri rapide : coût moyen



Même coût

- $T(n) = n-1 + \frac{1}{n} \sum_{i=0}^{n-1} (T(i) + T(n-i-1))$
- $T(n) = n-1 + \frac{2}{n} \sum_{i=0}^{n-1} T(i)$
- $n T(n) = n (n-1) + 2 \sum_{i=0}^{n-1} T(i) \quad (1)$
- $(n-1) T(n-1) = (n-1) (n-2) + 2 \sum_{i=0}^{n-2} T(i) \quad (2)$
- Après avoir calculé (1) - (2)
- $nT(n) - (n+1)T(n-1) = 2 (n-1)$
- $T(n)/(n+1) - T(n-1)/n = 2 (n-1) / (n (n+1))$

Tri rapide : coût moyen

- $T(n)/(n+1) - T(n-1)/n = 2(n-1) / (n(n+1))$
- $D(n) = T(n) / (n+1)$
- $D(n) = D(n-1) + 2(n-1) / (n(n+1))$
- $D(n) \approx D(n-1) + 2 / (n+1) \approx D(2) + 2 \sum_{k=4..n+1} 1/k$
- $D(n) < D(2) + 2 \log(n)$
- $T(n) < (n+1) T(2)/3 + 2(n+1) \log(n)$
- $T(n) = O(n \log(n))$


Partitionner

```
int Partitionner (T_elt t [], int d, int f){
    int i=d , j=f-1; // On utilise i et j comme « pointeurs » qui se déplacent
    int pivot = f ; // On choisit le dernier élément comme pivot

    while (i<j) {
        // On déplace i et j jusqu'à trouver des valeurs incohérentes % pivot
        while ((i<j) && (t[i]<=t[f])) i++ ;
        while ((i<j) && (t[j]>t[f])) j-- ;

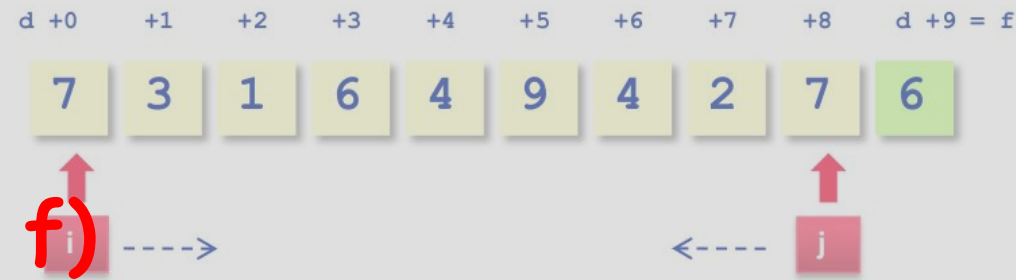
        if (i < j) {
            echanger(t,i,j);
            i++ ; j-- ;
        }
    }
    if (t[i]<=t[f]) i++ ; // Cf. ci-contre
    echanger(t, i, f) ;
    return i ;
}
```

// Après la boucle principale,
// les pointeurs i et j sont inversés : $i \geq j$
// On sait que tous ceux avant i sont plus petits ou égaux
// Tous ceux après j sont strictement plus grands
// - - - - j(-) i(+) + + + Piv.
// - - - - - j=i(?) + + + + + Piv.
// Où remplacer le pivot ?
// On choisit de le remplacer en i
// Il faut vérifier la valeur de la case i

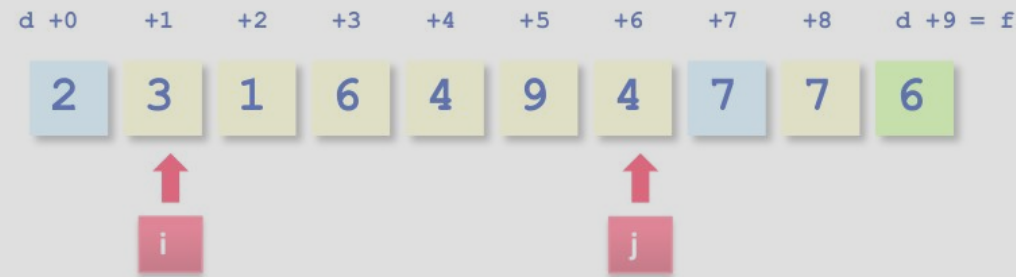


Partitionner (1)

ipivot = Partitionner(T, d, f)

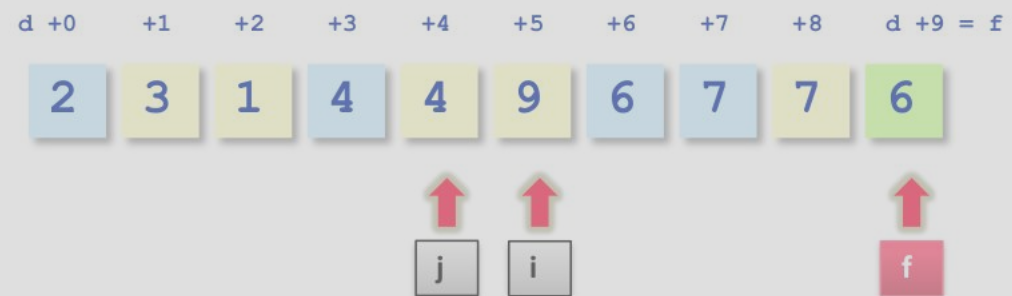
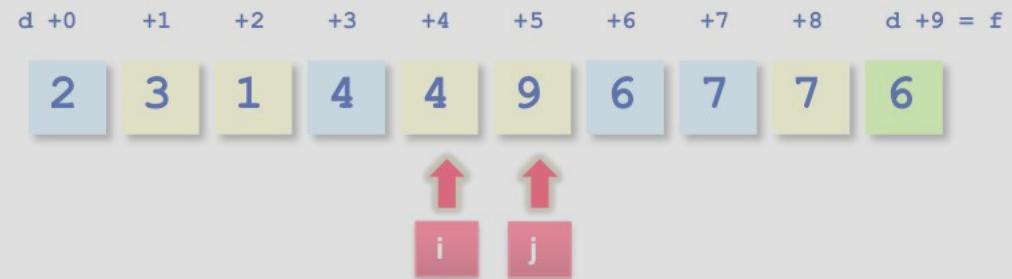
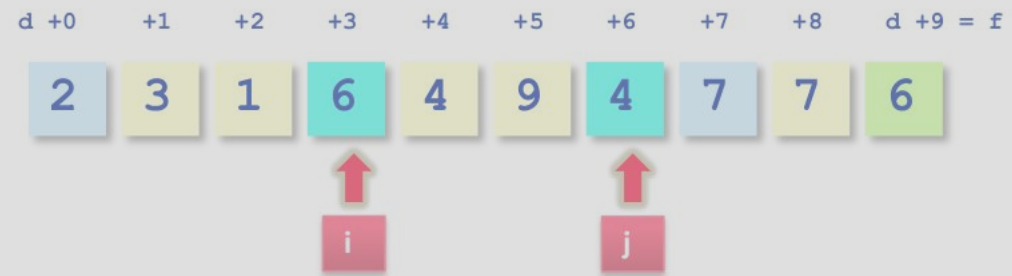


- Choix du pivot : par exemple le dernier
- L'indice i remonte dans T jusqu'à trouver un élément qui n'est pas inférieur à pivot
 - i est initialisé à d
- L'indice j redescend dans T jusqu'à trouver un élément qui est inférieur à pivot
 - j est initialisé à f - 1.
- Lorsque le premier élément supérieur ou égal à pivot est trouvé à partir du rang i en remontant ET lorsque le premier élément inférieur à pivot à partir du rang j, en descendant, sont trouvés:
 - Permutation de ces deux éléments, ET
 - Incrémentement de l'indice i, ET
 - Décrémentement de l'indice j



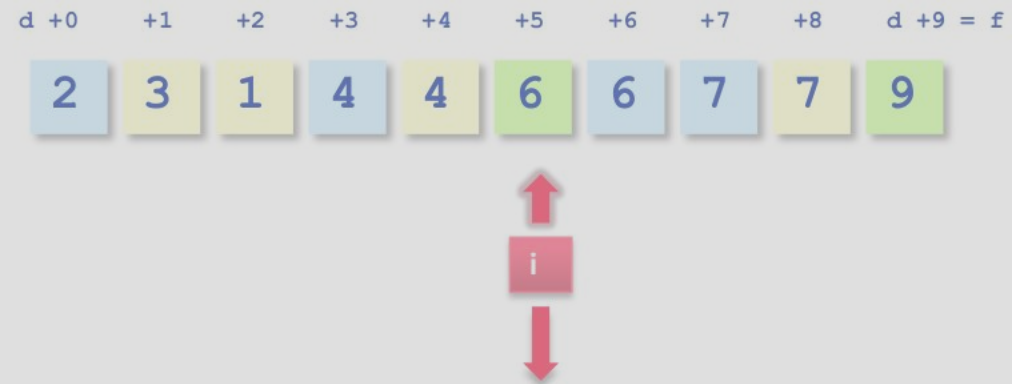
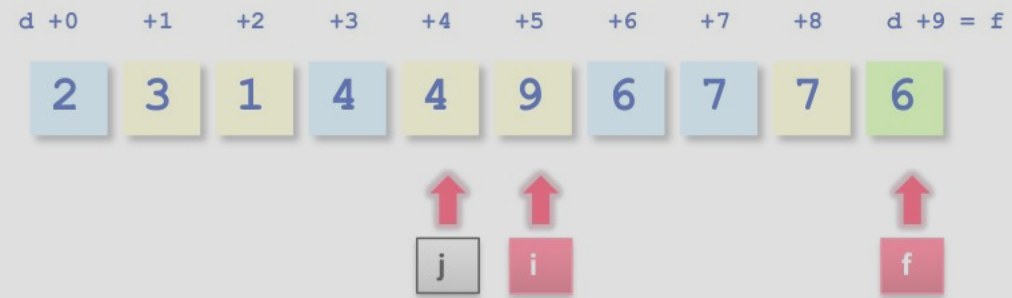
Partitionner (2)

- On recommence à partir des rangs i et j
- Lorsque le premier élément supérieur ou égal à pivot est trouvé à partir du rang i en remontant, ET lorsque le premier élément inférieur à pivot à partir du rang j , en descendant, sont trouvés :
 - Permutation de ces deux éléments, ET
 - Incrémentation de l'indice i , ET
 - Décrémentement de l'indice j
- La procédure s'arrête lorsque les indices i et j se croisent



Partitionner (3)

- Il ne reste plus qu'à permuter l'élément d'indice i avec le pivot (d'indice f)
- ... et à retourner l'indice de l'élément contenant la valeur pivot, c'est-à-dire l'indice i



Tri rapide : conclusion

- Tri qui peut **dégénérer**
- Rendre le tri indépendant des données :
 - Utiliser un pivot aléatoire
 - Appliquer au tableau une permutation aléatoire avant de le trier
- Tri **non stable**
- Tri **en place**

Tri rapide dans la librairie standard

- `void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *))`
- La fonction `qsort()` trie un tableau contenant `nmemb` éléments de taille `size`
- L'argument `base` pointe sur le début du tableau

Exercice : **ex7** : TEA

- Implémenter le tri rapide d'un tableau
- Mettre en évidence la dégénérescence
- Développer un moyen de l'éviter
- Comparer les performances de vos implémentations à **qsort**, tri fusion et tri fusion de listes

TEA

- Sujet : [TEA S3 2021](#)
- A remettre 24h avant la prochaine séance, sur moodle

Code Couleur

Légende des textes

- mot-clé important, variable, contenu d'un fichier, code source d'un programme
- chemin ou url, nom d'un paquet logiciel
- commande, raccourci
- commentaire, exercice, citation
- culturel, optionnel

Culturel / Approfondissement

- A ne pas connaître intégralement par coeur
 - Donc, le reste... est à maîtriser parfaitement !
- Pour anticiper les problématiques que vous rencontrerez en stage ou dans d'autres cours
- Pour avoir de la conversation à table ou en soirée...

Exemples ou Exercices

- Brancher le cerveau
- Participer
- Expérimenter en prenant le temps...

Bonnes pratiques, prérequis

- Des éléments d'organisation indispensables pour un travail de qualité
- Des rappels de concepts déjà connus