Complexité & Algorithmique Avancée

Cours 1 : complexité

Inspiré des cours de Christian vercauter et de Thomas Bourdeaud'huy

centralelille IG2I

Sommaire

- **Cours n°1 : Analyse d'algorithmes et complexité**
- **Cours n°2 : « Diviser pour régner » *Applications au tri : Tri fusion, Tri rapide***
- **Cours n°3 : Arbres équilibrés AVL**
- **Cours n°4 : Files de priorité, Maximier et tri pas tas**
- **Cours n°5 : Arbre d'Huffman et compression**
- **Cours n°6 : Graphes**

Analyse d'algorithmes et complexité

1. *Principe*
2. *Ordres de grandeur et notation asymptotique de Landau*
3. *Classes de complexité*
4. *Exemples*

Objectif

- 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 de algorithmes sans avoir à les programmer
- Définir une mesure
  - indépendante de l'ordinateur
  - indépendante du langage de programmation
  - c'-à-d, indépendante des spécifications d'implémentation

Estimer la taille des ressources

- Si l’on prend en compte tous les paramètres
  - fréquence d’horloge, nombre de CPU, temps d’accès mémoire, disque, etc.
- l’estimation de ces ressources peut :
  - être assez compliquée, voir irréalisable,
- Pour cela on se contente souvent d’estimer l’influence de la taille des données d’entrée sur la taille des ressources nécessaires, pour une machine abstraite RAM (Random access machine)
  - Machine séquentielle ;
  - Opérations élémentaires sur des données élémentaires ;
  - Durée des opérations connue et constante…

Estimer la taille des ressources

- Pour calculer le coût d'un algorithme, on détermine une fonction associant un coût en unité de temps à un paramètre entier n qui résume la taille des données.
- Exemple simple

<u>pour</u> i de 1 à n <u>faire</u>	*n fois*
x ← x+i	*c* (coût de l'instruction)
<u>fin pour</u>	

- coût total de l'algorithme : *c.n* (sa complexité)

Exemple 2

- Algorithme 2

```
pour i de 1 à n faire              n fois
      x ← i+2                      c1
fin pour

pour i de 1 à n faire              n fois
      pour j de 1 à n faire        n fois
            x ← (x + i)*(x+j)      c2
      fin pour
fin pour
```

Coût totale de l'algorithme 2 (Complexité)

$c_1 \times n + c_2 \times n \times n = c_2.n^2 + c_1.n$

Exemple de tri par insertion

- Entrée : un tableau A contenant n éléments indicés de 0 à n-1
- Sortie : le tableau A trié par ordre croissant.

```
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++;
}
```

Exemple (suite)

- Estimer le temps d'exécution de cet algorithme revient à prendre en compte :
  - le temps d’exécution de chaque instruction
  - le nombre de fois que chaque instruction est exécutée
- Chacune de ces instructions consiste en :
  - au plus 1 affectation,
  - au plus 1 addition/soustraction,
  - au plus 2 accès dans le tableau A,
  - au plus 2 comparaisons.

Exemple de tri par insertion

- Pour chaque ligne k, on notera c_k son temps d'exécution, et pour chaque valeur de i, on désigne par t_i le nombre d'exécutions du test de la boucle while interne
- Chaque quantité dépend de l'état initial du tableau 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++;
}
```

On a : 1 <= ti <= i+1

Exemple (suite)

```
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++;
}
```

Coût	répétitions
c_1	1 fois
c_2	n fois
c_3	n-1 fois
c_4	n-1 fois
c_5	$\sum_{i=0\to n-1} (t_i)$ fois
c_6	$\sum_{i=0\to n-1} (t_i -1)$ fois
c_7	$\sum_{i=0\to n-1} (t_i -1)$ fois
c_8	n-1 fois
c_9	n-1 fois

Exemple (suite)

- On note *T(n)* le temps d'exécution de l'algorithme du tri par insertion.

- On l'obtient en additionnant les produits des colonnes coût et répétitions

$$T(n) = c_1 + nc_2 + (n-1)(c_3 + c_4 + c_8 + c_9) + c_5 \sum_{i=1}^{n-1} t_i + (c_6 + c_7) \sum_{i=1}^{n-1} (t_i - 1)$$

Exemple (suite)

- <u>Cas favorable</u> : $t_i = 1$, cas du tableau déjà trié
  le test (A[j] > aux) est toujours faux

$$T(n) = (c_1 - c_3 - c_4 - c_5 - c_8 - c_9) + n(c_2 + c_3 + c_4 + c_5 + c_8 + c_9)$$
$$= a.n + b$$

- Cas défavorable : $t_i = i+1$, cas du tableau ordonné dans l'ordre inverse
  le test (A[j] > aux) est toujours vrai

$$T(n) = (c_1 - c_3 - c_4 - c_8 - c_9) + n(c_2 + c_3 + c_4 + c_8 + c_9)$$
$$+ c_5\left(\frac{(n+1)n}{2} - 1\right) + (c_6 + c_7)\left(\frac{(n-1)n}{2}\right)$$

Exemple (suite)

$$T(n) = (c_1 - c_3 - c_4 - c_8 - c_9) + \left(c_2 + c_3 + c_4 + \frac{c_5 - c_6 - c_7}{2} + c_8 + c_9\right)n$$
$$+ \left(\frac{c_5 + c_6 + c_7}{2}\right)n^2$$
$$= a.n^2 + b.n + c$$

- Le cas le plus favorable : Une fonction linéaire de n
- Le cas le plus défavorable : Une fonction quadratique de n

Opération de base

- Une autre façon de procéder consiste à identifier une opération de base pour l'algorithme :
  - 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.
- Exemple
  - 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 matrice, 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 : C'est le plus petit nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille n
  - La complexité dans le pire des cas : C'est le plus grand nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille n.
  - La complexité 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, $f(n)\ de\ \mathbb{N}\ dans\ \mathbb{R}$
  - par exemple, on ne s'intéresse qu'au terme le plus grand.
- On exprime la complexité d'un algorithme comme une fonction *f(n) de* $\mathbb{N}$ *dans* $\mathbb{R}$

Exemples

- $f(n) = n^3 + 3n^2 + n + 5 = O(n^3)$
- $f(n) = n\log(n) + 12n + 567 = O(n\log n)$
- $f(n) = 123n^{10} - 47n^7 + 2^n/890 = O(2^n)$

Remarques préliminaires

- Dans les fonctions de coût, les constantes ont peu d'importance. On peut les négliger
  - Elles ne dépendent que de la particularité de l'implémentation.
- On ne s'intéresse pas aux entrées de petite taille
  - Lorsqu'on analyse deux algorithmes de traitement d'un même problème, on compare leur complexité asymptotiquement

Notation Θ(grand Theta)

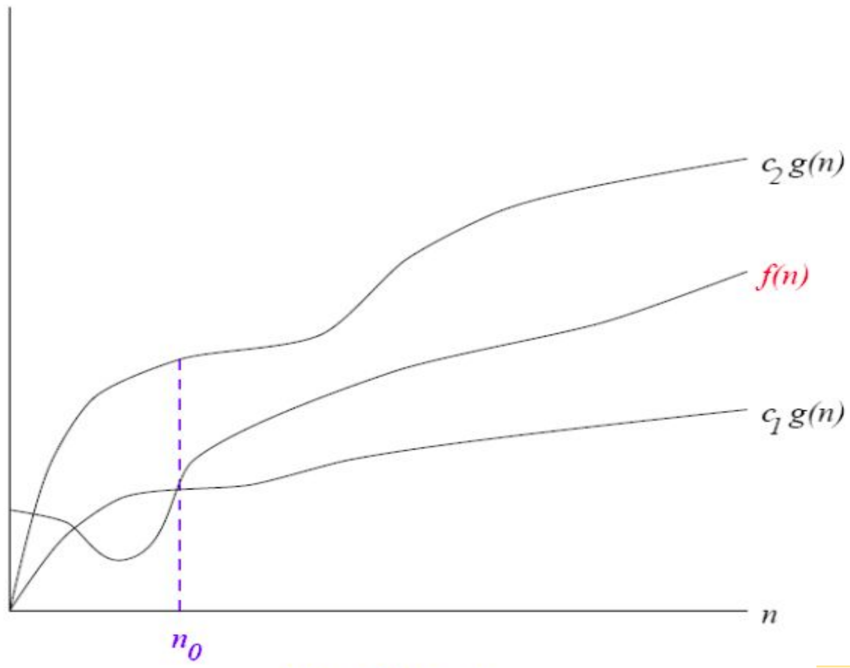
- Soient f et g deux fonctions de ℕ *dans* ℝ
- *On dit que g(n) est une* *borne approchée asymptotique* *pour f(n) s'il existe deux constantes strictement positives c_1 et c_2 ainsi que n_0* ∈ ℕ tels que pour tout $n \geq n_0$ on ait :

  $$0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)$$

- Ceci revient à dire que :
  - f(n) a une croissance comparable à celle de g(n) à un facteur constant près
- On écrit $f(n) = \boldsymbol{\Theta(g(n))}$

centralelille IG2I

Illustration



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

La notation Θ borne asymptotiquement une fonction

à la fois par excès et par défaut

Exemple

- Montrons que $n^2/2 - 3n = \mathbf{\Theta(n^2)}$
- Il faut déterminer c_1, c_2 et n_0 tels que :

$$0 \leq c_1 n^2 \leq n^2/2 - 3n \leq c_2 n^2 \qquad n_0 \geq n > 0$$

$$c_1 \leq 1/2 - 3/n \leq c_2$$

- On peut par exemple prendre $c_2 = 1/2$, alors la 1ère valeur de n_0 qui convient (c_1, $c_2 > 0$) est

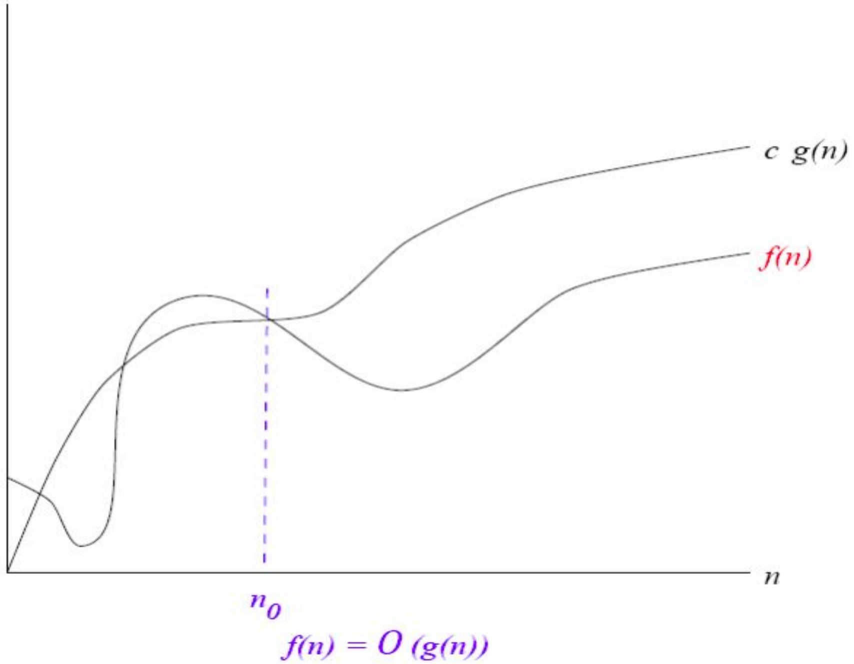
donc $n_0 = 7$ et $c_1 = 1/14$

Notation *O* (grand O)

- *On dit que g(n) est une borne supérieure asymptotique pour f(n) s'il existe une constante strictement positives c 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) à une constante près pour n assez grand
- On écrit $f(n) = O(g(n))$

Illustration



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

Exemples

- Montrons que $6n^2 + 2n - 8 = \mathbf{O(n^2)}$
- Il faut déterminer c, et n_0 tels que :

  $$0 \leq 6n^2 + 2n - 8 \leq c\,n^2 \qquad n_0 \geq n > 0$$

- On prend par exemple $c = 7$, et on cherche le n_0 tel que : $6n^2 + 2n - 8 \leq 7\,n^2$ et $n_0 \geq n > 0$
  on trouve $n_0 = 1$
  - donc $c = 7$ et $n_0 = 1$

Exemples

- $f(n) = a\,n^2 + b\,n + c$ avec $a > 0$
  alors $f(n) = \boldsymbol{\Theta}(n^2)$
  et également $f(n) = \boldsymbol{O}(n^2)$

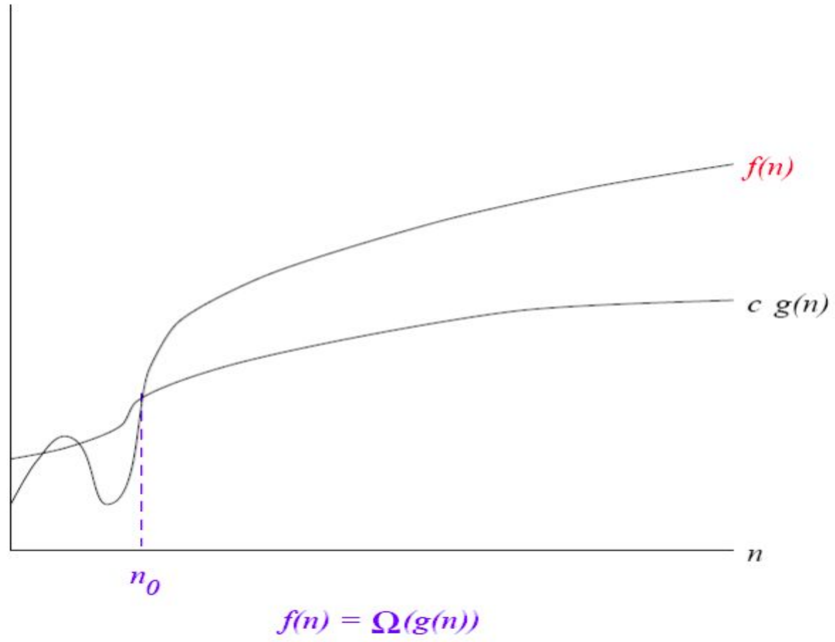
- $f(n) = b\,n + c$
  alors $f(n) = \boldsymbol{\Theta}(n)$
  Les propositions $f(n) = \boldsymbol{O}(n^2)$ et $f(n) = \boldsymbol{O}(n)$ sont vraies !

Notation Ω (grand Omega)

- *On dit que g(n) est une borne inférieure asymptotique pour f(n) s'il existe une constante strictement positive c 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) à un constante près pour n assez grand
- On écrit $f(n) = \boldsymbol{\Omega(g(n))}$

Illustration



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

Exemple

- $f(n) = b\ n + c$
  alors $f(n) = \boldsymbol{\Theta}(\ n\ )$
  et également $f(n) = \boldsymbol{\Omega}(\ n\ )$

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

Propriétés

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

Règle de sommation

- Considérons un programme constitué de deux blocs exécutés en séquence P et Q, respectivement de complexité en **O(g(n))** et ***O*(f(n))**

  P; Q; $\rightarrow$ $O(f(n) + g(n))$

  $O(f(n)) + O(g(n)) \rightarrow O(\max((f(n),g(n))))$
- Ainsi, si g(n) est d'ordre inférieure à f(n)
- $O(f(n)) \pm O(g(n)) \rightarrow O(f(n))$

Conditionnelle/itération

- 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 ≤ 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) **)***

Tri par 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

$\sum_{i=1 \to n-1} 1 = n-1 \Rightarrow$ **O (n)**

Pire cas : i comparaisons

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

⇒ O (n^2)

Tri par sélection

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;
    }
} }
```

n-1 itérations

n-i-1 comparaisons

$\sum_{i=0 \to n-2} (n-i-1) = (n-1).n - \sum_{i=1 \to n-1} i$

$= (n-1).n - (n-1).n/2$

$= (n-1).n/2$

$\Rightarrow$ O (n^2)

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 A(1..n)

- Complexité linéaire : Coût en $\Theta(n)$

  Exemple : calcul du produit scalaire de deux vecteurs de taille n, calcul factoriel de n.

- Complexité quasi-linéaire : Coût en $\Theta(n \log n)$

  Exemple : Tri par fusion

Classes de complexité

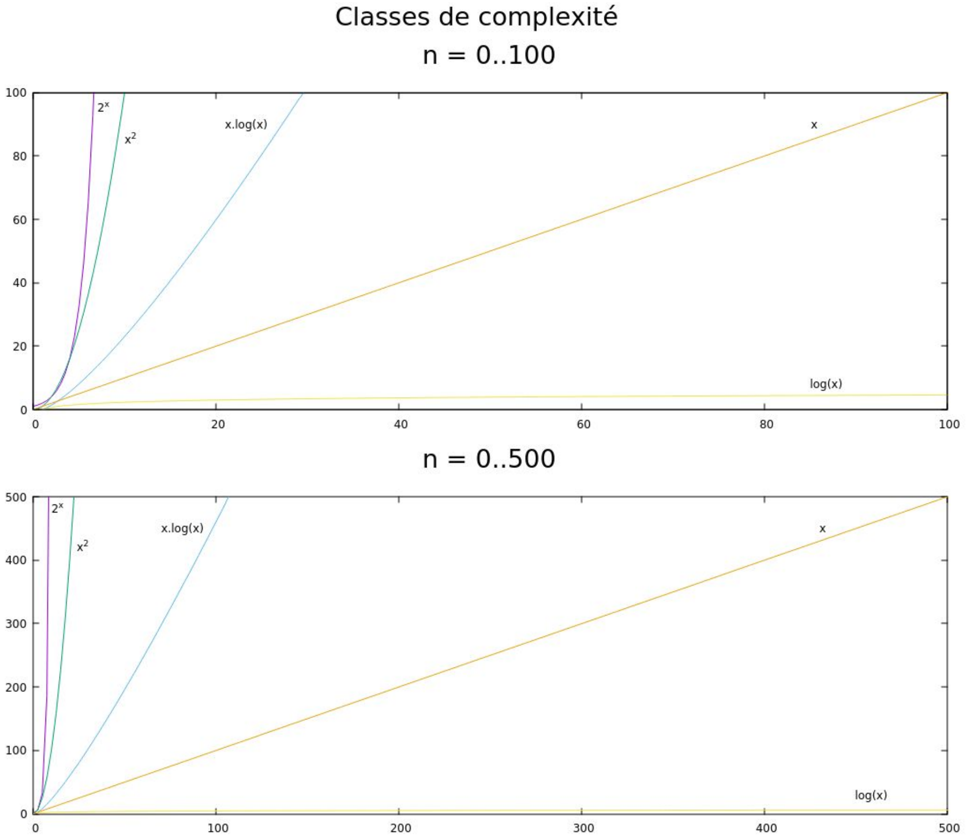
- Complexité polynomiale : Coût en avec $\Theta(n^k)$ avec k > 1

  Exemple : Cubique pour la multiplication de deux matrices carrées d'ordre n : $\Theta(n^3)$,

  Quadratique pour les tris dit lents $\Theta(n^2)$
- Complexité exponentielle : Coût en avec $\Theta(a^n)$ a >1

  Exemple : Tours de Hanoï

Allure des courbes



Taux de croissance

T(n)	n = 10	n = 20	n = 30	n = 40	n = 50	n = 60
log n	1 µs	1,3 µs	1,5 µs	1,6 µs	1,7 µs	1,8 µs
n	10 µs	20 µs	30 µs	40 µs	50 µs	60 µs
n log n	10 µs	26 µs	44 µs	64 µs	85 µs	107 µs
n^2	100 µs	400 µs	900 µs	1,6 ms	2,5 ms	3,6 µs
n^3	1 ms	8 ms	27 ms	64 ms	125 ms	216 ms
2^n	1 ms	1 s	18 mn	13 jours	36 ans	366 siècles
3^n	59 ms	59 mn	6,5 ans	3855 siècles	2,3 E +8 siècles	1,3E+13 siècles

machine avec 1 µs par opération

L'âge de l'univers est estimé à 13,7 milliards d'années

Effet de l’évolution technique

- Méfiez-vous de ceux qui affirment : “Ne vous souciez jamais de l’efficacité d’un algorithme : Contentez-vous d’attendre !!”
- Selon la loi de Moore, cofondateur d’intel (article 19/04/1965) "À coût constant, la rapidité des processeurs double tous les 18 mois”
- “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).
- En moins de cinq ans, les performances sont multipliées par 10, en moins de 10 ans, elles sont multipliées par 100.
- Cette loi est remise en cause aujourd'hui (limite physique du matériel)

Effet de l'évolution technique

- 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 et 1000 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 : $v/100 \times 2^{N'} = 1h$
    - $N' = N + \log_2 100 = 6,6 + N$
-

Effet de l'évolution technique

- Le surcroît de puissance importe peu !
- Analysez la complexité de vos algorithmes !
- Tâchez d'en produire de complexité raisonnable !

	Aujourd'hui	Demain	Après-demain
T(n)	k = 1	k' = 100 k	k' = 1000 k
n	N	100 N	1000 N
n^2	N	10 N	31,6 N
n^3	N	4,6 N	10 N
2^n	N	N + 7	N + 10
3^n	N	N + 4	N + 6

Diviser pour régner

- Principe de conception d'algorithmes et de décomposition d'un problème complexe en sous-problèmes plus faciles à traiter
- Algorithme récursif : Trois étapes
  - paramétrage du problème faisant apparaître les différents éléments dont dépend la solution en particulier la taille du problème à résoudre qui devra décroître à chaque appel récursif.
  - recherche du ou des cas triviaux et de leur solution.
  - décomposition du cas général de façon à réduire le problème à résoudre en un ou plusieurs sous-problèmes plus simples car de taille plus petite (c'est ici qu'apparaissent les appels récursifs).

Diviser pour régner (récursivité)

- Processus de développement similaire à la démonstration par récurrence en mathématiques
- Pas forcément performant, mais facile à développer !
- 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

Principe

- La complexité de traitements définis de façon récursive s’exprime toujours par une relation de récurrence
- Il existe essentiellement trois méthodes de résolution d’équations de récurrence :
  - Méthode par substitution
  - Méthode itérative
  - Le « théorème maître »

Exemple : recherche dichotomique

```
int RechercheDicho(char * T[], char * e, int debut, int fin)
{
    int Test, milieu;

    if (debut > fin)                    /* 1er cas trivial   */
        return -1;
    milieu = (debut + fin) / 2;
    Test = strcmp(e, T[milieu]);
    if (Test == 0)                      /* 2ème cas trivial */
      return milieu;
    else if (Test < 0)
      return RechercheDicho(T, e, debut, milieu - 1);
    else /* Test > 0 */
      return RechercheDicho(T, e, milieu + 1, fin);
}
```

Recherche dichotomique

- Notons *n* la taille de l'intervalle de recherche initial.
  - n = (fin – debut) + 1
- Notons *T(n)* le coût maximal en nombre de comparaisons entre éléments du tableau. Il s'exprime par la relation de récurrence suivante :

$$T(0) = 0$$

$$T(n) = 1 + T(\lceil n/2 \rceil) \qquad (n > 0)$$

$\lceil n \rceil$: plus petit entier supérieur ou égal à n

$\lfloor n \rfloor$: plus grand entier inférieur ou égal à n

Comparaison de e et de l'élément au centre de l'intervalle

A chaque appel récursif, l'intervalle de recherche est réduit de moitié

Méthode par substitution

- Procède par intuition de la solution et vérification par substitution
- Exemple : cas de la recherche dichotomique
    - $T(0) = 0$ et $T(n) = 1 + T(n/2)$ pour $n>0$
    - on a l'Intuition que $T(n)$ est $\Theta(\log_2(n))$
- Hypothèse : $T(n) = a.\log_2(n) + c \quad (n > 0)$
    - et donc, $T(n/2) = a.\log_2(n/2) + c = a.\log_2(n/2) - a + c$
- Vérification : on substitut $a.\log_2(n) - a + c$ à $T(n/2)$ dans
- Conclusion : $T(n) = 1 + a.\log_2(n) - a + c$
    - donc $1 - a + c = c$ et $a = 1$
    - enfin, sachant que $T(1) = 1$, on en déduit $c = 0$
- $T(n) = \log_2(n)$

Exemple : tours de Hanoï

```
void Hanoi(int nbDisques, int Org, int Dest, int Inter)
{
     if (nbDisques > 0)
     {
        Hanoi(nbDisques - 1, Org,   Inter, Dest);
        deplacer1Disque(Org, Dest);
        Hanoi(nbDisques - 1, Inter, Dest,  Org);
     }
}
```

$T(0) = 0$

Cas trivial : pas de disque à déplacer

$T(n) = 2 .T(n - 1) + 1 \quad (n>0)$

Cas général : 2 appels récursifs avec n-1 disque à déplacer, plus un déplacement élémentaire

Méthode itérative

- Procède par développement et sommations
- Exemple : tours de Hanoï
  - $T(n) = 2\,T(n-1) + 1$ pour $n>0$ et $T(0) = 0$
  - $T(n) = 2\,(2\,T(n-2) + 1) + 1 = 2^2\,T(n-2) + 2 + 1$
  - $T(n) = 2^2\,(2\,T(n-3) + 1) + 2 + 1 = 2^3\,T(n-3) + 2^2 + 2 + 1$
  - $T(n) = 2^i\,T(n-i) + 2^{i-1} + \ldots + 2 + 1$
  - $T(n) = 2^n\,T(0) + 2^{n-1} + \ldots + 2 + 1$ lorsque $i=n$
  - $T(n) = 2^{n-1} + \ldots + 2 + 1 = 2^n - 1$
- C'est un algorithme de complexité exponentielle : $\Theta(2^n)$
  - 1ns par déplacement élémentaire : $T(64) = 584$ années !!

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*
  - f(n) est le coût des traitements hors des appels récursifs.

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

Théorème maître : exemple 1

- **$T(n) = 16T(n/4) + n$**
  - $a = 16$, $b = 4$, donc $c = \log_b(a) = 2$, $n^c = n^2$
  - $f(n) = n = \mathbf{\Theta}(n) = \mathbf{O}(n)$
- Cas n°1 : $f(n) = O(n^{\log4(16-\varepsilon)})$ en prenant $\varepsilon = 12$
- Alors $T(n) = \mathbf{\Theta}(n^2)$

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

Théorème maître : exemple 2

- **$T(n) = T(n/2) + 1$**
  - $a = 1$, $b = 2$, donc $c = \log_b(a) = 0$, $n^c = 1$
  - $f(n) = 1 = \boldsymbol{\Theta}(1)$
- Cas n°2 en prenant $k = 0$
- Alors $T(n) = \boldsymbol{\Theta}(n^{\log2(1)} \log^1(n)) = \boldsymbol{\Theta}(\log(n))$

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

Théorème maître : exemple 3

- **$T(n) = 3T(n/2) + n^2$**
  - $a = 3$, $b = 2$, donc $c = \log_b(a) = \log_2(3) \approx 1.58$, $n^c \approx n^{1.58}$
  - $f(n) = n^2 = \boldsymbol{\Theta}(n^2) = \boldsymbol{\Omega}(n^2)$
- Cas n°3 : $\boldsymbol{\Omega}(n^{\log 2(3+\varepsilon)})$ en prenant $\varepsilon = 1$
  - $3\ (n/2)^2 \leq k\ n^2$ en prenant $k = 3/4 < 1$
- Alors $T(n) = \boldsymbol{\Theta}(n^2)$

- Règle 3 : n^c croît moins vite que $f(n)$, et $f(n)$ satisfait une *condition de régularité*
  - Si $f(n) = \boldsymbol{\Omega}(n^{\log b(a+\varepsilon)})$ avec $\varepsilon>0$
  - Et $\exists$ $k<1$ tq. $a\ f(n/b) \leq k\ f(n)$ pour n grand
  - Alors, $T(n) = \boldsymbol{\Theta}(f(n))$

Tris optimaux : Vocabulaire

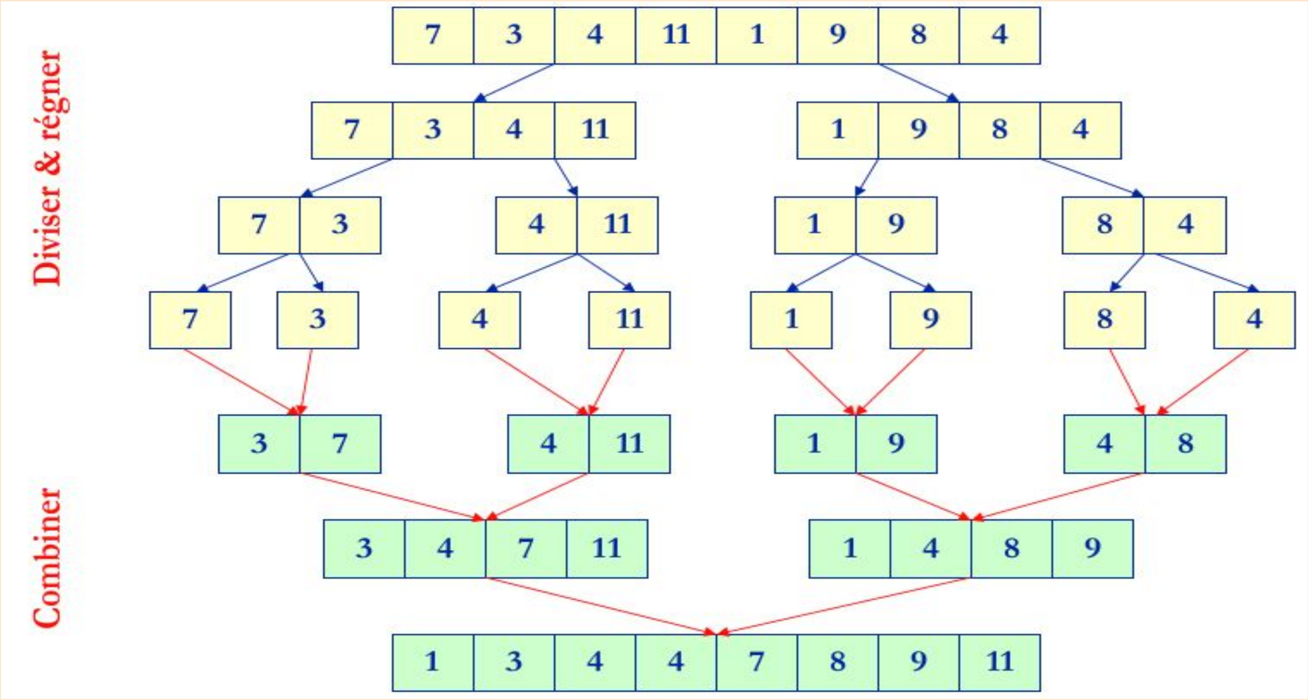
https://fr.wikipedia.org/wiki/Algorithme_de_tri

- 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 $n \cdot \log_2(n)$

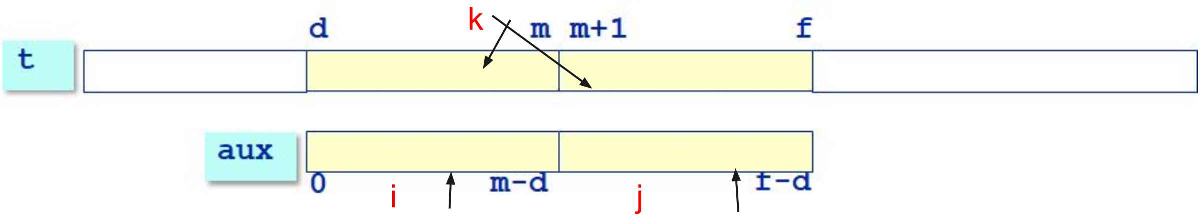
Tri fusion : principe

- 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



Fusionner




```c
void fusionner(T_Elt t [], int d, int m, int f)
  {
     T_Elt aux[f - d + 1]; /* !!! C99 !!! */
     int i, j, k;
     memcpy(aux, &t[d], (f - d + 1) * sizeof(T_Elt));
     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++];
         else
                t[d + k++] = aux[j++];
     for (; i <= m - d; t[d + k++] = aux[i++]);
     for (; j <= f - d; t[d + k++] = aux[j++]);
  }
```

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

Régner

Combiner

Tri fusion : coût

- 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 $T(0) = 0$

Tri fusion : résolution itérative

- $T(n) = 2 \times T(n/2) + \Theta(\ n\ )$, $T(1) = 0$, $T(0) = 0$
  - $T(n) = 2\ T(n/2) + cn$
  - $T(n) = 2\ (2\ T(n/4) + cn/2) + cn = 4\ T(n/4) + 2\ cn$
  - $T(n) = 4\ (2\ T(n/8) + cn/4) + 2\ cn = 2^3\ T(n/2^3) + 3\ cn$
  - $T(n) = 2^i\ T(n/2^i) + i\ cn$
- Quand $i = \lfloor \log_2(n) \rfloor$:
  - $2^i \leq n < 2^{i+1}$ $\quad i \leq \log_2(n) < i + 1 \qquad\qquad i = \lfloor \log_2(n) \rfloor$
  - $T(n) = n\ T(1) + \lfloor \log_2(n) \rfloor\ cn = \lfloor \log_2(n) \rfloor\ cn$
- $T(n) = \Theta(\ n \log_2(n)\ )$

Tri fusion : Théorème Maître

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

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

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 (in stitu) 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

Tri rapide : principe

- 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.
  - et 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

```
    int iPivot;

     if (fin > debut)
    {
        iPivot = Partitionner(T, debut, fin);
        Tri_rapide(t, debut, iPivot - 1);
        Tri_rapide(t,        iPivot + 1, fin);
    }
}
```

Diviser

Régne

Régner

centralelille IG2I

Partitionnement

```
static int partitionner (long [] t, int iDebut, int iFin)
{
   int i = iDebut - 1, j = iFin;
   long pivot = t[iFin];//choix du pivot, ici le dernier

   While (1){ // traitement itératif

      while (t[++i] < pivot);

      while (pivot < t[--j])
         if (j <= i) break;

      if (i >= j) break; //croisement

      long aux = t[i];
      t[i] = t[j];
      t[j] =
      aux;
   }

   t[iFin] = t[i];
   t[i] = pivot;
   return i;
}
```

centralelille IG2I

Tri rapide : coût

- Soit *n* la taille de l'ensemble à trier : $n = fin - debut + 1$
- Remarque : le coût de partitionner est en **Θ(n)**
  - Le pivot ayant été choisi, il faut le comparer à tous les autres éléments de l'ensemble (n-1 comparaisons)
- Après le partitionnement, l'élément pivot est placé à sa place définitive *iPivot*, il délimite deux partitions
  - l'un de n1 = (*iPivot - debut) éléments*
  - l'un de n1 = (*ifin - iPivot) éléments*

Tri rapide : coûts

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

Tri rapide : coûts

- Cas favorable : chaque partie contient n/2 éléments (à ± 1)
  - $T(n) = 2\,T(n/2) + \Theta(\,n\,)$ 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 : $T(n) = O(\,n \log(n)\,)$
  - $T(0) = 0$, $T(1) = 0$
  - $T(n) = n-1 + 1/n \sum_{i=0 \to n-1}(\,T(i) + T(n-i-1)\,)$ $n>1$

Partitionner (1)

```
ipivot = Partitionner(T, d, f);
```

- ⇨ *Choix du pivot : par exemple le dernier*
  - pivot = 6   iPivot = f;
- ⇨ *Deux indices:*
  - L'indice i qui remonte dans T jusqu'à trouver un élément qui n'est pas inférieur à pivot;
    i est initialisé à d
  - L'indice j qui 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
  - Incrémentation de l'indice i, ET
  - Décrémentation de l'indice j

Partitionner (2)

```
ipivot = Partitionner(T, d, f);
```

- ⇨ *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
  - Décrémentation de l'indice j.
- ⇨ *La procédure s'arrête lorsque les indices i et j se croisent.*

Partitionner (3)

```
ipivot = Partitionner(T, d, f);
```

- ⇨ *Il ne reste plus qu'à permuter l'élément d'indice i avec le pivot (d'indice f)*

d +0	+1	+2	+3	+4	+5	+6	+7	+8	d +9 = f
2	3	1	4	4	9	6	7	7	6
				j	i				f

- ⇨ *… et à retourner l'indice de l'élément contenant la valeur pivot, c'est-à-dire l'indice i.*

d +0	+1	+2	+3	+4	+5	+6	+7	+8	d +9 = f
2	3	1	4	4	6	6	7	7	9
					i				

Il faut garantir que :

- ⇨ *L'indice j ne devienne pas inférieur à d : cas où le pivot est supérieur ou égal à tous les autres éléments entre f - 1 et d*
- ⇨ *L'indice i ne devienne pas supérieur à f : cas où le pivot est inférieur à tous les autres éléments entre d et f*
  - Ce cas est correctement traité car lorsque i = f, alors T[i] n'est plus strictement inférieur à pivot, car égal à pivot.

Tri rapide : conclusion

- Tri qui peut dégénérer et devient en (O (n^2))
- Complexité minimale et moyenne : O(n.log(n))
- 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
- peu adapté aux tableaux de petite taille
  - en dessous d'une certain seuil, appliquer un tri par insertion

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