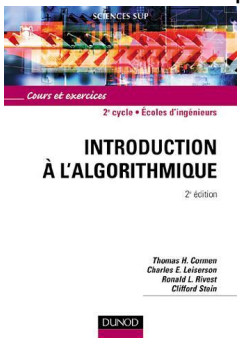
Algorithmique

Alexandre Duret-Lutz
adl@lrde.epita.fr

7 novembre 2014

Références

Comme beaucoup de cours d'algorithmes, celui-ci s'appuie sur :



Introduction à l'algorithmique, par Thomas Cormen, Charles Leiserson, Ronald Rivest et Clifford Stein.

- Une grosse partie de ces transparents est dérivée de ceux de Sylvain Peyronnet (donnés à l'Épita en 2005).
- D'autres sont inspirés des supports de cours de Erik D. Demaine et Charles E. Leiserson au MIT : `http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-046JFall-2005/LectureNotes/`

Plan du cours

- Partie 1 : Notion d'algorithme, complexité
- Partie 2 : Tris et rangs
- Partie 3 : Structures de données
- Partie 4 : Principaux paradigmes

Première partie I

Introduction : algorithmes et complexité

1. Notion d'algorithme
2. Mesure de complexité
   - Comment mesurer ?
   - Exemple du tri par insertion
   - Analyse asymptotique et ordres de grandeur
   - Rappel sur les arbres
   - Tri fusion
   - Théorème général pour les équations de récurrence

Notion d'algorithme

1. Notion d'algorithme
2. Mesure de complexité
   - Comment mesurer ?
   - Exemple du tri par insertion
   - Analyse asymptotique et ordres de grandeur
   - Rappel sur les arbres
   - Tri fusion
   - Théorème général pour les équations de récurrence

Une définition d'algorithme informelle

Un ensemble d'opérations de calcul élémentaires qui sont organisées de façon à produire un ensemble de valeurs en sortie (output) à partir d'un ensemble de valeurs fournies en entrée (input).

Un algorithme résout toujours un problème de calcul. L'énoncé du problème spécifie la relation entrée/sortie souhaitée.

Opérations de calcul élémentaires :

- opérations arithmétiques,
- lecture/affectation de variables,
- comparaisons...

Organisation : (aussi appellé *structure de contrôle*)

- tests,
- boucles...

Exemples

1. **Multiplication**
   Entrée : deux entiers a et b
   Sortie : leur produit ab
   Algorithme : celui de l'école
2. **Plus Grand Commun Diviseur (PGCD)**
   Entrée : deux entiers a et b
   Sortie : $pgcd(a, b)$
   Algorithme : celui d'Euclide
3. **Test de primalité** Entrée : un entier a
   Sortie : vrai ssi a est premier
   Algorithme : boucle pour tester tous les diviseurs ?

On parle plutôt de problème de décision dans le dernier cas : on veut décider (plutôt que calculer) si a est premier.

Une définition formelle ?

Donner une définition précise de la notion d'algorithme est assez difficile et complexe.
Il existe plusieurs définitions mathématiques depuis les années 30.
Objectifs : pouvoir raisonner facilement sur les algorithmes, les composer, les prouver, calculer le coût de leur exécution (temps, espace)

- fonctions récursives
- λ-calcul
- machines de Turing
- RAM (ici : « Random Access **Machine** »)

Calculabilité

Fait :
: Toutes ces définitions sont équivalentes

Thèse de Church :
: Tout ce qui est calculable intuitivement est aussi calculable par les objets formels précédemment cités

Mais :
: Il existe des problèmes non calculables (indécidables)

Exemple : problème de l'arrêt
Entrée : Un algorithme (programme) A et une entrée I
Sortie : A termine-t-il pour I ?

Restriction des problèmes envisagés

Dans ce cours, on étudie seulement des problèmes pour lesquels il existe des algorithmes.

En fait, on s'intéressera aux problèmes pour lesquels il existe des algorithmes efficaces.

Un exemple typique de problème décidable pour lequel il n'y a pas d'algorithme efficace : le jeu d'échec (le nombre de configurations est environ 10^{50}).

Mesure de complexité

1 Notion d'algorithme

2 Mesure de complexité
- Comment mesurer ?
- Exemple du tri par insertion
- Analyse asymptotique et ordres de grandeur
- Rappel sur les arbres
- Tri fusion
- Théorème général pour les équations de récurrence

Mesure de complexité

On veut une mesure pour comparer des algorithmes qui résolvent le même problème (ou non !). On voudrait que la complexité :

- ne dépende pas de l'ordinateur
- ne dépende pas du langage de programmation
- ne dépende pas du programmeur
- soit indépendante des détails l'implémentation
- etc.

Random Access Machines

On choisit de travailler sur un modèle d'ordinateur idéalisé.
RAM :

- processeur unique
- instructions exécutées séquentiellement
- mémoire infinie
- accès aléatoires à la mémoire en temps constant

Pour mesurer l'efficacité d'un algorithme, on va mesurer le temps qu'il utilise. Il suffit de compter les instructions et de les pondérer par leur coût.

On peut aussi vouloir mesurer la mémoire utilisée.

(Concentrons-nous sur le temps pour l'instant.)

Tri par insertion

Problème : Trier un tableau d'entiers
Entrée : un tableau A d'entiers
Sortie : le tableau A trié par ordre croissant

```
InsertionSort(A)
1  for j ← 2 to length(A) do
2        key ← A[j]
3        i ← j − 1
4        while i > 0 and A[i] > key do
5              A[i + 1] ← A[i]
6              i ← i − 1
7        A[i + 1] ← key
```


Tri par insertion : coût en temps

InsertionSort(A)	coût	fois
1 for $j \leftarrow 2$ to $length(A)$ do	c_1	n
2 $key \leftarrow A[j]$	c_2	$n-1$
3 $i \leftarrow j-1$	c_3	$n-1$
4 while $i > 0$ and $A[i] > key$ do	c_4	$\sum_{j=2}^{n} t_j$
5 $A[i+1] \leftarrow A[i]$	c_5	$\sum_{j=2}^{n}(t_j-1)$
6 $i \leftarrow i-1$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $A[i+1] \leftarrow key$	c_7	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n}(t_j-1) + c_6 \sum_{j=2}^{n}(t_j-1) + c_7(n-1)$$

Cas le plus favorable ? Cas le plus défavorable ?

Tri par insertion : cas favorable et défavorable

Cas favorable : le tableau est trié. $t_j = 1$.

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$$

C'est une fonction linéraire du style $an + b$.

Cas défavorable : le tableau est trié par ordre décroissant. $t_j = j$.

Rappelons que $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ et $\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$.

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

C'est une fonction quadratique du style $an^2 + bn + c$.

Cas moyen ?

- Le cas le plus favorable est la borne inférieure
- Le cas le plus défavorable est la borne supérieure
- Le cas général se situe entre les deux (en toute logique !)
- On fait parfois des analyses de « cas moyen » :
  Tableau d'entrée = n nombres tirés au hasard (on suppose une loi uniforme). Pour une cléf choisie ligne 2 en moyenne la moitié du tableaux lui est supérieur et l'autre moitié lui est inférieure.
  Donc $t_j = \frac{t}{2}$.
  On a $\sum_{j=2}^{n} \frac{t}{2} = \frac{n(n+1)-2}{4}$ et $\sum_{j=2}^{n} \frac{t}{2} - 1 = \frac{n(n-3)+2}{4}$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\frac{n(n+1)-2}{4} + c_5\frac{n(n-3)+2}{4} + c_6\frac{n(n-3)+2}{4} + c_7(n-1)$$

C'est encore une fonction quadratique du style $an^2 + bn + c$.

Indépendance vis-à-vis de la machine

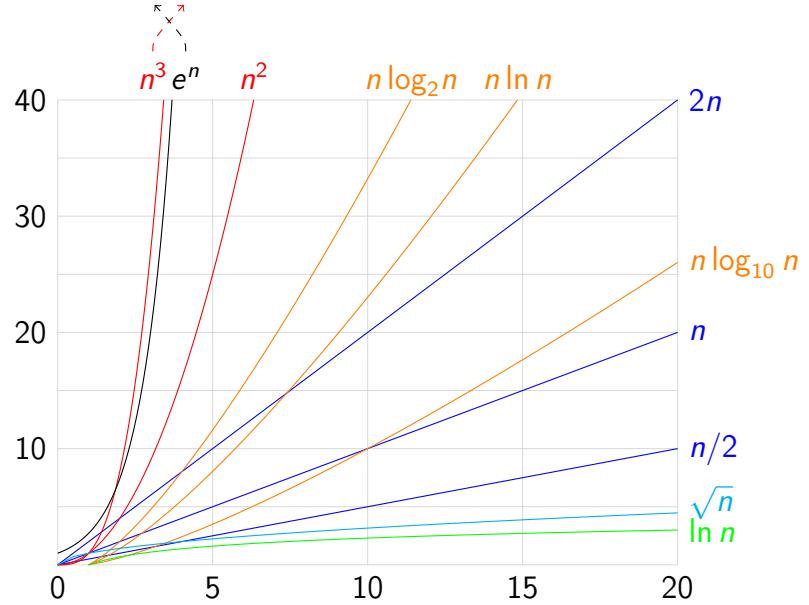
Dans nos formules de $T(n)$, les coefficients c_1, c_2, ..., c_7 dépendent de la machine utilisée.

On souhaite :

- ignorer les constantes qui dépendent des machines,
- étudier la croissance de $T(n)$ lorsque $n \to \infty$.

$\Longrightarrow$ On fait une analyse asymptotique du temps de calcul

Comparaison de fonctions



Concrètement...

Supposons 10^6 opérations par seconde.

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
10^2	6.6 μs	0.1 ms	0.6 ms	10 ms	1 s	$4 \cdot 10^6$ ans
10^3	9.9 μs	1 ms	10 ms	1 s	16.6 min	
10^4	13.3 μs	10 ms	0.1 s	1.6 min	11.6 j	
10^5	16.6 μs	0.1 s	1.6 s	2.7 h	317 ans	
10^6	19.9 μs	1 s	19.9 s	11.6 j	10^6 ans	
10^7	23.3 μs	10 s	3.9 min	3.17 ans		
10^8	26.6 μs	1.6 min	44.3 min	317 ans		
10^9	29.9 μs	16.6min	8.3 h	31709 ans		

Équivalence asymptotique de fonctions

$$\Theta(g(n)) = \{f(n) \mid \exists c_1 \in \mathbb{R}^{+\star}, \exists c_2 \in \mathbb{R}^{+\star}, \exists n_0 \in \mathbb{N}, \\ \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

Impose que f et g soient positives asymptotiquement.

Par convention on note « $f(n) = \Theta(g(n))$ » au lieu de « $f(n) \in \Theta(g(n))$ »

Si $a_2 > 0$, on a $a_2 n^2 + a_1 n + a_0 = \Theta(n^2)$.
Par exemple on pose $c_1 = \frac{a_2}{2}$ et $c_2 = \frac{3a_2}{2}$, et on montre

$$\frac{a_2}{2} \leq a_2 + \underbrace{\frac{a_1}{n} + \frac{a_0}{n^2}}_{\to 0 \text{ quand } n \to \infty} \leq \frac{3a_2}{2}$$

Complexité asymptotique du tri par insertion

Cas favorable

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) \\ = \Theta(n)$$

Cas défavorable

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

Cas moyen

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\frac{n(n+1)-2}{4} \\ + c_5\frac{n(n-3)+2}{4} + c_6\frac{n(n-3)+2}{4} + c_7(n-1) = \Theta(n^2)$$

Simplifions nous les calculs

Pour mesurer la complexité en temps on choisit une opération fondamentale pour le problème de calcul particulier, et on calcule le nombre d'opérations fondamentales exécutées par l'algorithme.
Le choix est bon si le nombre total d'opérations est proportionnel au nombre d'opérations fondamentales.

Exemples :

problème	opération fondamentale
addition des entiers binaires	opération binaire
multiplication de matrices	multiplication scalaire
tri d'un ensemble d'éléments	comparaison des éléments

Calculs simplifiés

InsertionSort(A)	coût	fois
1 for $j \leftarrow 2$ to $length(A)$ do	c_1	n
2 $key \leftarrow A[j]$	c_2	$n-1$
3 $i \leftarrow j-1$	c_3	$n-1$
4 while $i > 0$ and $A[i] > key$ do	c_4	$\sum_{j=2}^{n} t_j$
5 $A[i+1] \leftarrow A[i]$	c_5	$\sum_{j=2}^{n}(t_j - 1)$
6 $i \leftarrow i-1$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $A[i+1] \leftarrow key$	c_7	$n-1$

Calculs simplifiés

		fois
InsertionSort(A)		
1	for $j \leftarrow 2$ to $\mathit{length}(A)$ do	
2	$\mathit{key} \leftarrow A[j]$	
3	$i \leftarrow j-1$	
4	while $i > 0$ and $A[i] > \mathit{key}$ do	$\sum_{j=2}^{n} t_j$
5	$A[i+1] \leftarrow A[i]$	
6	$i \leftarrow i-1$	
7	$A[i+1] \leftarrow \mathit{key}$	

Complexité asymptotique du tri par insertion

Cas favorable

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$$
$$= \Theta(n)$$

Cas défavorable

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

Cas moyen

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\frac{n(n+1)-2}{4}$$
$$+ c_5\frac{n(n-3)+2}{4} + c_6\frac{n(n-3)+2}{4} + c_7(n-1) = \Theta(n^2)$$

Complexité asymptotique du tri par insertion

Cas favorable

$$T(n) = n-1$$
$$= \Theta(n)$$

Cas défavorable

$$T(n) = \frac{n(n+1)}{2} - 1$$
$$= \Theta(n^2)$$

Cas moyen

$$T(n) = \frac{n(n+1)-2}{4}$$
$$= \Theta(n^2)$$

Bornes asymptotiques supérieures et inférieures

$$\mathrm{O}(g(n)) = \{f(n) \mid \exists c \in \mathbb{R}^{+\star}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$$
$$\Omega(g(n)) = \{f(n) \mid \exists c \in \mathbb{R}^{+\star}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\}$$

Notons que
$f(n) = \Theta(g(n)) \iff f(n) = \mathrm{O}(g(n))$ et $f(n) = \Omega(g(n))$.

Ou encore $\Theta(g(n)) = \mathrm{O}(g(n)) \cap \Omega(g(n))$.

Comme on a $\Theta(n) \subseteq \mathrm{O}(n^2)$ et $\Theta(n^2) \subseteq \mathrm{O}(n^2)$, on pourra dire que le temps d'exécution du tri par insertion de n éléments est en $\mathrm{O}(n^2)$.

Propriétés

- Si $\lim_{n\to\infty} \frac{g(n)}{f(n)} = c > 0$ alors $g(n) = \Theta(f(n))$.
- Si $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$ alors $g(n) = \mathrm{O}(f(n))$ et $f(n) \neq \mathrm{O}(g(n))$.
- Si $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$ alors $f(n) = \mathrm{O}(g(n))$ et $g(n) \neq \mathrm{O}(f(n))$.
- $f(n) = \mathrm{O}(g(n))$ ssi $g(n) = \Omega(f(n))$.

Exercices

- Trouver deux fonctions f et g telles que $f(n) \neq \mathrm{O}(g(n))$ et $g(n) \neq \mathrm{O}(f(n))$.
  On dit que deux fonctions peuvent être incomparables au sens de O.
- Montrer que $\Theta(f(n) + g(n)) = \Theta(\max(f(n), g(n)))$.
- Si $f(n) = \Theta(g(n))$, a-t-on $2^{f(n)} = \Theta(2^{g(n)})$?
  Justifiez votre réponse (preuve si oui, contre-exemple si non).
- Définissons l'ordre (partiel) suivant :

  $\Theta(f(n)) \leq \Theta(g(n))$ si $f = \mathrm{O}(g(n))$
  $\Theta(f(n)) < \Theta(g(n))$ si $f = \mathrm{O}(g(n))$ et $g \notin \mathrm{O}(f(n))$

  Énumérer par ordre croissant les ensembles $\Theta(\ldots)$ des fonctions suivantes :
  n, 2^n, $n \log n$, $\ln n$, $n + 7n^5$, $\log n$, $\sqrt{n}$, e^n, 2^{n-1}, n^2, $n^2 + \log n$, $\log \log n$, n^3, $(\log n)^2$, $n!$, $n^{3/2}$.

Analyse avec les notations asymptotiques

Considérons la complexité dans un cas quelconque.

```
InsertionSort(A)
1  for j ← 2 to length(A) do                 Θ(n)+
2        do key ← A[j]                       Θ(n)+
3        i ← j − 1                           Θ(n)+
4        while i > 0 and A[i] > key do       O(n²)+
5              do A[i + 1] ← A[i]            O(n²)+
6              i ← i − 1                     O(n²)+
7        A[i + 1] ← key                      Θ(n)
                                             ------
                                             O(n²)
```


Arbres enracinés : définition graphique

Arbre

Un arbre est un graphe non-orienté acyclique et connexe.

Arbre enraciné

Un arbre enraciné est un arbre avec un sommet distingué. Ce sommet distingué s'appelle la racine de l'arbre.
Les sommets sont de façon équivalente appelés nœuds.

Soit x un nœud dans un arbre enraciné T de racine r. Un nœud quelconque y sur l'unique chemin entre x et r est un ancêtre de x ; réciproquement, x est un descendant de y. En parle d'ancêtre propre et de descendant propre si $x \neq y$. Si (y, x) est le dernier arc du chemin entre r et x, alors y est **le** père de x et x est **un** fils de y. Une feuille est un nœud sans fils. (La racine est l'unique nœud sans père.) Le degré d'un nœud est son nombre de fils. La profondeur de x est la longueur du chemin entre r et x. La hauteur de T est la profondeur maximale de ses nœuds.

Arbres enracinés : définition mathématique

Ensemble avec relation de précédence

$A = \{a, b, c, d, e, f, g, h, i\}$ et $\prec_1$ précédence sur A :
$a \prec_1 b$, $a \prec_1 c$, $a \prec_1 d$, $b \prec_1 e$, $b \prec_1 f$, $c \prec_1 g$, $f \prec_1 h$, $f \prec_1 i$.
D'où une représentation arborescente de A.
On note $\preccurlyeq$ la clôture réflexive et transitive.
$\preccurlyeq$ relation d'ordre (non total) sur A.

Ensemble ordonné

Soit $(A, \preccurlyeq)$ un ensemble ordonné. $\prec$ est l'ordre strict associé à A.

$$\mathrm{Maj}(a) = \{x \in A \mid a \preccurlyeq x\} \qquad \mathrm{Min}(a) = \{x \in A \mid x \preccurlyeq a\}$$
$$\mathrm{Maj}^\star(a) = \{x \in A \mid a \prec x\} \qquad \mathrm{Min}^\star(a) = \{x \in A \mid x \prec a\}$$

Arbres « mathématiques »

Un ensemble ordonné $(A, \preccurlyeq)$ est un arbre ssi

- A est fini
- A possède un unique plus petit élément appelé racine
- $\forall a \in A$, $\mathrm{Min}(a)$ est totalement ordonné.

Arbres enracinés : définition récursive

Définition récursive

Un arbre enraciné est un ensemble de un ou plusieurs nœuds tel que

- l'un des nœuds r est désigné comme la racine
- le reste des nœuds est partionné en $n \geq 0$ ensembles disjoints T_1, T_2, …, T_n tels que chacun de ces ensembles est lui-même un arbre. Les arbres T_1, T_2, …, T_n sont appelés sous-arbres de r

Le degré d'un nœud est le nombre de sous-arbres dont il est la racine.

Arbres ordonnés, arbres binaires

Arbres ordonnés

Un arbre ordonné est un arbre enraciné dans lequel les fils d'un nœud sont ordonnés.
On peut parler de premier sous-arbre, troisième fils, etc.

Arbre binaire

Un arbre binaire est un ensemble fini de nœuds qui est soit vide, soit formé d'un nœud racine et de deux arbres disjoints appelés sous-arbre gauche et sous-arbre droit.

Une feuille est un nœud dont les sous-arbres associés sont vides.

Un arbre binaire n'est pas un arbre ordonné dont les nœuds seraient de degré au plus 2. Un arbre binaire peut avoir un sous-arbre gauche vide (i.e., seulement un fils droit).

L'arbre binaire vide, noté ε, n'est pas un arbre. Tous les autres arbres binaires en sont.

Propriétés d'arbre (1/2)

- Tout arbre présentant n sommets possède $n - 1$ arrêtes.

  Preuve par récurrence. Facile de se convaincre en considérant que seule la racine d'un arbre n'a pas de père.

- Un arbre binaire non vide de f feuilles possède $f - 1$ nœuds de degré 2.

  Preuve par récurrence sur le nombre de feuille $f > 0$ de l'arbre A. Trois cas se présentent pour le cas de récurence :

  - Soit A est le regroupement de A_g et A_d avec $f_g + f_d = f$. On a alors $n = 1 + n_g + n_d = 1 + (f_g - 1) + (f_d - 1) = f - 1$.
  - Soit A est le regroupement de A_g et ε, alors forcément $n = n_g = f_g - 1 = f - 1$.
  - Soit A est le regroupement de ε et A_d, idem.

Propriétés d'arbre (2/2)

- Le nombre de feuilles d'un arbre binaire de hauteur h est d'au plus 2^h. (Preuve par récurrence sur h.)
- Un arbre binaire de $f > 0$ feuilles possède une hauteur d'au moins $\lceil \log f \rceil$. (Conséquence directe.)

- Le nombre de nœud d'un arbre binaire de hauteur h est d'au plus $2^{h+1} - 1$. (Preuve par récurrence sur h.)
- Un arbre binaire de $n > 0$ nœuds possède une hauteur d'au moins $\lceil \log(n+1) - 1 \rceil = \lfloor \log n \rfloor$. (Conséquence directe.)

Arbres équilibrés

Arbre binaire complet

Un arbre binaire non vide est complet si tous ses nœuds sont de degré 2 ou 0.
Les nombres de nœuds et de feuilles d'un arbre binaire complet sont liés par $n = 2f - 1$. (Preuve : on a vu qu'un arbre binaire non vide possède $f - 1$ nœuds de degré 2.)

Arbre binaire équilibré

Un arbre binaire non vide A est équilibré si les longueurs de deux de ses branches ne peuvent différer que d'1 au plus.
Les feuilles d'un tel arbre se situent soit à la profondeur $p = \lfloor \log(n+1) - 1 \rfloor$ soit à la profondeur $p' = \lceil \log(n+1) - 1 \rceil$.
(Un tel arbre n'est pas forcément complet.)

On retient $h = \lfloor \log n \rfloor$ pour les arbres binaires équilibrés.

Tri fusion

Problème : Trier un tableau d'entier
Entrée : un tableau A d'entier, premier indice l, dernier indice r
Sortie : le tableau A trié par ordre croissant

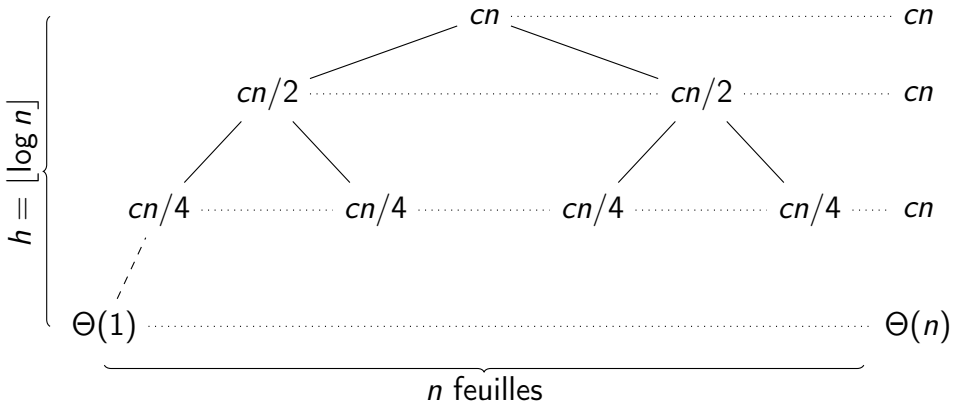
Merge-Sort(A, l, r)	$T(1)$	$T(n)$
1 if $l < r$ then	$\Theta(1)$	$\Theta(1)$
2 $q \leftarrow \lfloor (l+r)/2 \rfloor$		$\Theta(1)$
3 Merge-Sort(A, l, q)		$T(\lfloor n/2 \rfloor)$
3 Merge-Sort$(A, q+1, r)$		$T(\lceil n/2 \rceil)$
4 Merge(A, l, q, r)		$\Theta(n)$

On a donc la relation de récurrence

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T(n/2) + \Theta(n) & \text{si } n > 1 \end{cases}$$

Arbre de récursion

Résolvons $T(n) = 2T(n/2) + cn$ pour une constante $c > 0$.



On a donc $T(n) = \Theta(n \log n)$.

Résolution par dépliage

Une deuxième façon de résoudre $T(n) = 2T(n/2) + cn$ est par dépliage.

$$\begin{aligned} T(n) &= cn + 2T(n/2) \\ &= cn + 2(cn/2) + 4T(n/4) = 2cn + 4T(n/4) \\ &= 2cn + 4(cn/4) + 8T(n/8) = 3cn + 8T(n/8) \\ &\vdots \\ &= kcn + 2^k T(n/2^k) \end{aligned}$$

On peut continuer de déplier jusqu'à atteindre $T(1) = \Theta(1)$, ce qui arrive quand $2^k = n$. En substituant $k = \log_2 n$, on obtient donc :

$$\begin{aligned} T(n) &= cn \log_2 n + n \times T(1) \\ &= \Theta(n \log n) + \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

Théorème général

Soit à résoudre

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq n_0 \\ aT(n/b) + f(n) & \text{si } n > n_0 \end{cases}$$

avec $a \geq 1$, $b > 1$, $n_0 \in \mathbb{N}$.
Alors

- Si $f(n) = \mathrm{O}(n^{\log_b a - \varepsilon})$ pour un $\varepsilon > 0$, alors $T(n) = \Theta(n^{\log_b a})$.
- Si $f(n) = \Theta(n^{\log_b a})$, alors $T(n) = \Theta(n^{\log_b a} \log n)$.
- Si $f(n) = \Omega(n^{\log_b a + \varepsilon})$ pour un $\varepsilon > 0$, et si $af(n/b) \leq cf(n)$ pour une constante $c < 1$ et tous les n suffisament grands, alors $T(n) = \Theta(f(n))$.

Attention, il y a des trous dans ce théorème : une fonction $f(n)$ peut n'être prise en compte par aucun des trois cas. Le ε force la fonction $f(n)$ a être polynomialement plus grande ou plus petite que $n^{\log_b a}$.

Applications du théorème

- $T(n) = 2T(n/2) + \Theta(n)$
  $a = b = 2$, $n^{\log_b a} = n$. On a donc $T(n) = \Theta(n \log n)$.
- $T(n) = 4T(n/2) + \sqrt{n}$
  $a = 4$, $b = 2$, $n^{\log_b a} = n^2$. $\epsilon = 1$ et $\sqrt{n} = \mathrm{O}(n^{2-1})$, on a donc $T(n) = \Theta(n^2)$
- $T(n) = 3T(n/3) + n^2$
  $a = 3$, $b = 3$, $n^{\log_b a} = n$. $\epsilon = 1$ et $n^2 = \Omega(n^{1+1})$, d'autre part $3(n/3)^2 \leq cn^2$ pour $c = 1/3$, a donc $T(n) = \Theta(n^2)$.
- $T(n) = 4T(n/2) + n^2/\log n$
  $a = 4$, $b = 2$, $n^{\log_b a} = n^2$. Mais on ne peut pas trouver d'$\varepsilon > 0$ tel que $n^2/\log n = \mathrm{O}(n^{2-\varepsilon})$. En effet $n^2/\log n \leq cn^{2-\varepsilon}$ implique $n^\varepsilon \leq c \log n$. Le théorème ne s'applique pas.

Deuxième partie II

Tris et rangs

3 Tris comparatifs
- Définition
- Rappel des tris par insertion et fusion
- Tri par sélection
- Tri par tas
  - Structure de tas
  - Opérations sur le tas
- Tri rapide
- Tri introspectif
- Bornes de complexité des tris comparatifs

4 Tris linéaires
- Tri par dénombrement
- Tri par paquets

5 Mélange aléatoire

6 Recherche dans un tableau trié

7 Rangs et médians
- Minimum
- Sélection
- Sélection stochastique
- Sélection en $\mathrm{O}(n)$

Tris comparatifs

3 Tris comparatifs
- Définition
- Rappel des tris par insertion et fusion
- Tri par sélection
- Tri par tas
  - Structure de tas
  - Opérations sur le tas
- Tri rapide
- Tri introspectif
- Bornes de complexité des tris comparatifs

4 Tris linéaires
- Tri par dénombrement
- Tri par paquets

5 Mélange aléatoire

6 Recherche dans un tableau trié

7 Rangs et médians
- Minimum
- Sélection
- Sélection stochastique
- Sélection en $\mathrm{O}(n)$

Le problème

Entrée : Une séquence de n nombres $\langle a_1, a_2, \dots a_n \rangle$
Sortie : Une permutation $\langle a'_1, a'_2, \dots a'_n \rangle$ de la séquence d'entrée telle que $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Souvent les entiers sont une partie d'une donnée plus complexe, l'entier est alors la clé de la donnée. Et on triera les données d'après les clés.
La structure représentant A est généralement un tableau.
Pour l'instant on ne s'intéresse qu'aux tris comparatifs : l'ordre des valeurs ne peut être déterminé qu'en les comparant entre elles.

Complexité d'un problème

Définition
La complexité $C(n)$ d'un problème P est la complexité du meilleur algorithme qui résout P.

Conséquences
- Si un algorithme A résout P en $\mathrm{O}(f(n))$, alors $C(n) = \mathrm{O}(f(n))$.
- Si l'on prouve que tous les algorithmes qui résolvent P ont une complexité en $\Omega(g(n))$, alors $C(n) = \Omega(g(n))$.
- Si ces bornes sont de grandeur équivalentes (i.e., $f(n) = \Theta(g(n))$) alors $C(n) = \Theta(f(n)) = \Theta(g(n))$, c'est la complexité du problème.

Pour l'instant, on a prouvé que la complexité du tri de n éléments est dans $\mathrm{O}(n^2)$.
- cela ne veut pas dire qu'il est impossible de faire mieux
- il est toujours possible de faire pire

Résumé des épisodes précédents

Tri par insertion
En $\mathrm{O}(n^2)$.
On en déduit que la complexité du problème du tri comparatif est en $\mathrm{O}(n^2)$.

Tri fusion
En $\Theta(n \log n)$.
On en déduit que la complexité du problème du tri comparatif est en $\mathrm{O}(n \log n)$.

Ce n'est pas incompatible puisque $\mathrm{O}(n \log n) \subset \mathrm{O}(n^2)$.

Le tri par insertion se fait en place (nombre de variables auxiliaires indépendant de n). Le tri fusion demande un tableau temporaire pour la fusion, il n'est pas en place.

Ces deux tris sont stables : l'ordre des éléments égaux est préservé.

Tri par sélection

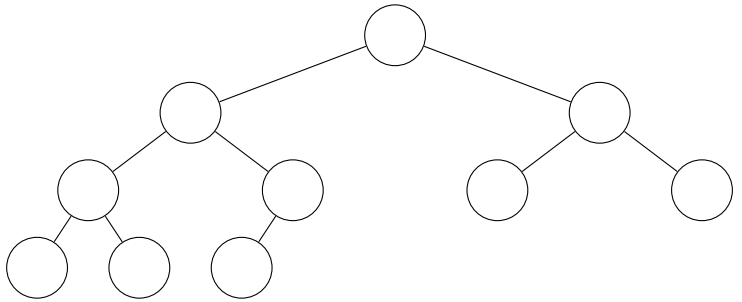
On cherche le minimum du tableau $A[1..n]$ et on l'échange avec la première position. On cherche le minimum du tableau $A[2..n]$ et on l'échange avec la deuxième position, etc. Après k recherches/échanges, $A[1..k]$ est trié.

Select-Sort(A)		$T(n)$
1	for i from 1 to n do	$\Theta(n)$
2	$min \leftarrow i$	$\Theta(n)$
3	for j from i to n do	$\Theta(n^2)$
4	if $A[j] < A[min]$ then $min \leftarrow j$	$\Theta(n^2)$
5	$A[min] \leftrightarrow A[i]$	$\Theta(n)$
		$\Theta(n^2)$

C'est pire que InsertionSort qui est dans $\mathrm{O}(n^2)$ mais pas dans $\Omega(n^2)$.
Tri en place, mais instable (à cause de l'échange).

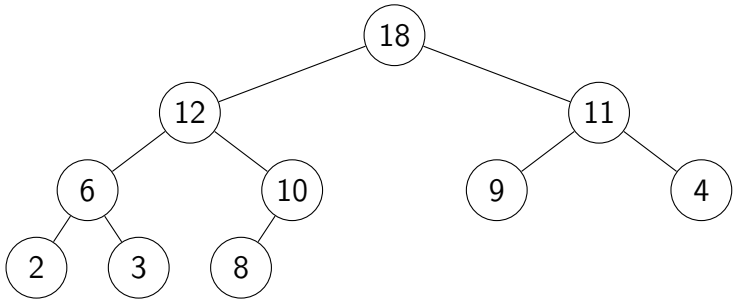
Arbre parfait

Un arbre parfait est un arbre binaire équilibré dont les feuilles du dernier niveau sont toutes à gauche.



Arbre parfait partiellement ordonné et tas

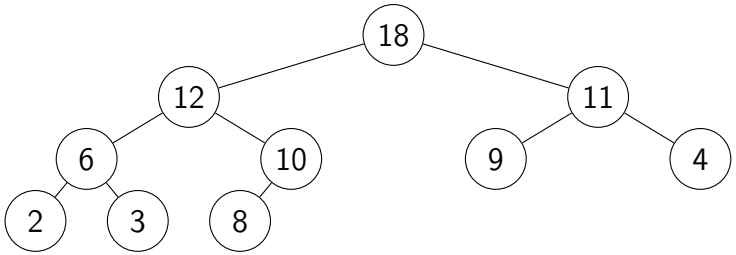
Un arbre parfait partiellement ordonné est un arbre parfait étiqueté de façon à ce que l'étiquette d'un nœud soit supérieure à celles de ses fils.



Un tas est un tableau qui peut s'interpréter comme un arbre parfait partiellement ordonné.

18	12	11	6	10	9	4	2	3	8

Propriétés d'un tas



$A =$

18	12	11	6	10	9	4	2	3	8

L'indice 1 correspond à la racine

Parent(i) $= \lfloor i/2 \rfloor$ (si $i > 0$)

LeftChild(i) $= 2i$ (s'il existe)

RightChild(i) $= 2i + 1$ (s'il existe)

La propriété du tas : $\forall i > 0$, $A[\text{Father}(i)] \geq A[i]$.

Opérations sur le tas (1) : Heapify

Entrée : un tableau A et deux indices i et m tels que $A[\text{LeftChild}(i)]$ et $A[\text{RightChild}(i)]$ soient des racines de tas dans $A[1..m]$,
Sortie : le tableau A tel que $A[i]$ soit une racine de tas.

```
Heapify(A, i, m)
1   l ←LeftChild(i)
2   r ←RightChild(i)
3   if l ≤ m and A[l] > A[i]
4       then largest ← l
5       else largest ← i
6   if r ≤ m and A[r] > A[largest]
7       then largest ← r
8   if largest ≠ i then
9       A[i] ↔ A[largest]
10      Heapify(A, largest, m)
```

$T(n) \leq T(2n/3) + \Theta(1)$ avec n la taille du sous-arbre commençant en et $2n/3$ le nombre maximum de nœuds du sous-arbre recursivement exploré.

On en déduit $T(n) = \mathrm{O}(\log n)$

Aussi $T(h) = \mathrm{O}(h)$ avec h la hauteu de l'arbre commençant à i.

Opérations sur le tas (2) : Build-Heap

Entrée : un tableau A quelconque.
Sortie : le tableau A réordonné en tas.

```
Build-Heap(A)
1   for i ← ⌊length(A)/2⌋ down to 1 do
2       Heapify(A, i, length(A))
```

Les éléments entre $\mathit{length}(A)/2 + 1$ et la fin du tableau sont les feuilles de l'arbre est sont déjà des tas. On corrige le reste du tableau en remontant.

Naïvement si $n = \mathit{length}(A)$, $T(n) = \underbrace{\Theta(n)}_{\text{la boucle}} \underbrace{\mathrm{O}(\log n)}_{\text{Heapify}} = \mathrm{O}(n \log n)$.

Mais en réalité le temps passé dans Heapify dépend de la taille du sous-arbre.

Opérations sur le tas (3) : Build-Heap

Calculons $T(n)$ pour Build-Heap de façon plus précise.

La hauteur d'un tas de n nœuds est $\lfloor \log n \rfloor$.
Le nombre de sous arbres de hauteur h dans un tas de n nœuds est au plus $\lceil n/2^{h+1} \rceil$.
La complexité d'Heapify sur une hauteur h est $\mathrm{O}(h)$.

On en déduit :

$$T(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \mathrm{O}(h) = \mathrm{O}\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^{h+1}} \right) = \mathrm{O}\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^{h}} \right)$$

Comme $\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$ on a $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$.
On obtient $T(n) = \mathrm{O}(n)$.

Algorithme du tri par tas

```
Heap-Sort(A)
1   Build-Heap(A)
2   for i ← length(A) down to 2 do
3       A[1] ↔ A[i]
4       Heapify(A, 1, i − 1)
```

On a bien sûr :

$$T(n) = \underbrace{\mathrm{O}(n)}_{\text{Build-Heap}} + \mathrm{O}(\underbrace{n}_{\text{boucle}} \underbrace{\log n}_{\text{Heapify}}) = \mathrm{O}(n \log n)$$

Tri rapide

Origine

Sir Charles Antony Richard Hoare, 1962.
Tri en place[1] et instable.

Idée générale

On partage le tableau en deux sous-tableaux tels que tous les éléments du premier soient plus petits que ceux du second. On trie ces deux tableaux recursivement.

Comment partager ?

On choisit une clé, qui sert de pivot. Par échanges successifs, on ordonne le tableau en deux blocs :

- au début, les éléments inférieurs (ou égaux) au pivot
- à la fin, les éléments supérieurs (ou égaux) au pivot

Notre choix : le pivot est le premier élément.

1. prétendument, car la récursion consomme de l'espace en $\mathrm{O}(\log n)$

Tri rapide : algorithme

Entrée : un tableau A d'entiers, premier indice l, dernier indice r
Sortie : le tableau A trié par ordre croissant

```
Quick-Sort(A, l, r)
1  if l < r then
2      p ← Partition(A, l, r)
3      Quick-Sort(A, l, p)
4      Quick-Sort(A, p + 1, r)
```

Entrée : un tableau A d'entiers, premier indice l, dernier indice r
Sortie : un entier p et le tableau A réordonné tq $A[l..p] \leq A[p+1..r]$.

```
Partition(A, l, r)
1  x ← A[l] ; i ← l − 1 ; j ← r + 1
2  repeat forever
3      do i ← i + 1 until A[i] ≥ x
4      do j ← j − 1 until A[j] ≤ x
5      if i < j then
6          A[i] ↔ A[j]
7      else
8          return j
```

$T_{\text{Partition}}(n) = \Theta(n)$ si $n = r - l + 1$.

Complexité du tri rapide

Cas défavorable

Le partionnement crée un arbre de découpe très déséquilibré.
Cela se produit si l'entrée est déjà triée (dans un sens ou l'autre).

$$\begin{aligned} T(n) &= \Theta(n) + \Theta(1) + T(n-1) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

Cas favorable

Le partionnement crée un arbre de découpe équilibré.

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

C'est la même équation que pour le tri fusion. On sait que la solution est $T(n) = \Theta(n \log n)$.

Intuition de la complexité moyenne (1)

Supposons un arbre déséquilibré « 1/10 : 9/10 »

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + \Theta(n)$$

On dessine l'arbre de récursion. La branche la plus courte mesure $\log_{10} n = \Theta(\log n)$: la complexité de l'abre limité à ce niveau est $\Theta(n \log n)$. On en déduit que $T(n) = \Omega(n \log n)$. La plus longue branche à une hauteur de $\log_{10/9} n$ et le coût à chaque niveau est $\leq n$. En en déduit que $T(n) \leq n \log_{10/9} n = \mathrm{O}(n \log n)$. Finalement $T(n) = \Theta(n \log n)$.

Toute partition selon un facteur constant implique $T(n) = \Theta(n \log n)$.

Intuition de la complexité moyenne (2)

Dans le cas général, on peut imaginer que les partitions vont être aléatoirement entre « bonnes » et « mauvaises ».

Exemple extrême

On suppose qu'on alterne une fois sur deux une *M*auvaise partition $(1 : n-1)$ et une *B*onne partition $(\lceil n/2 \rceil : \lfloor n/2 \rfloor)$.

$$M(n) = B(n-1) + \Theta(n)$$
$$B(n) = 2M(n/2) + \Theta(n)$$

On en déduit

$$\begin{aligned} M(n) &= 2M((n-1)/2) + \Theta(n/2) + \Theta(n) \\ &= 2M((n-1)/2) + \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

Calcul de la complexité moyenne (1)

On suppose une distribution uniforme des partitions.
La partition coupe $A[1..n]$ en $A[1..i]$ et $A[i+1..n]$ avec $n-1$ choix possibles pour i.

$$T(n) = \frac{1}{n-1}\sum_{i=1}^{n-1}(T(i) + T(n-i)) + \Theta(n) = \frac{2}{n-1}\sum_{i=1}^{n-1} T(i) + cn$$

D'autre part :

$$T(n-1) = \frac{2}{n-2}\sum_{i=1}^{n-2} T(i) + c(n-1)$$

Cherchons à faire apparaître $T(n-1)$ dans $T(n)$:

$$T(n) = \frac{2(n-2)}{(n-1)(n-2)}\left(T(n-1) + \sum_{i=1}^{n-2} T(i)\right) + c(n-1+1)$$

Calcul de la complexité moyenne (2)

$$\begin{aligned} T(n) &= \frac{2(n-2)}{(n-1)(n-2)}\left(T(n-1) + \sum_{i=1}^{n-2} T(i)\right) + c(n-1+1) \\ &= \frac{2}{n-1}T(n-1) + \frac{n-2}{n-1}T(n-1) + c(n-1)\left(1 - \frac{n-2}{n-1}\right) + c \\ &= \frac{n}{n-1}T(n-1) + 2c \end{aligned}$$

On divise à droite et à gauche par n :

$$\frac{T(n)}{n} = \frac{T(n-1)}{n-1} + \frac{2c}{n}$$

Calcul de la complexité moyenne (3)

$$\frac{T(n)}{n} = \frac{T(n-1)}{n-1} + \frac{2c}{n}$$

on pose $Y(n) = \frac{T(n)}{n}$

$$Y(n) = Y(n-1) + \frac{2c}{n} = 2c\sum_{i=1}^{n}\frac{1}{i}$$

$$T(n) = 2cn\sum_{i=1}^{n}\frac{1}{i}$$

La formule d'Euler : $\sum_{i=1}^{n}\frac{1}{i} = \ln n + \gamma + o(1)$.
On en déduit

$$T(n) = \Theta(n)\Theta(\log n) = \Theta(n \log n)$$

Questions

- Que se passe-t-il si tous les éléments du tableau à trier ont la même valeur ?
- On a l'impression qu'un tableau aléatoire sera trié plus rapidement qu'un tableau trié. Comment faire en sorte que l'algorithme aie la même complexité (en moyenne) sur les tableaux aléatoires que sur les tableaux triés ?

Tri rapide stochastique

L'idée est simple : on choisit le pivot aléatoirement dans le tableau.

```
RandomizedPartition(A, l, r)
1  x ← A[Random(l, r)] ; i ← l − 1 ; j ← r + 1
2  repeat forever
4     do j ← j − 1 until A[j] ≤ x
3     do i ← i + 1 until A[i] ≥ x
5     if i < j then
6        A[i] ↔ A[j]
7     else
8        return j
```

L'effet est le même que si l'on avait mélangé le tableau avant de la trier. Au lieu de *supposer* la distribution d'entrée, on l'a *imposée*.

Avantage : aucune entrée particulière ne provoque le pire cas ; celui-ci ne dépend que du générateur aléatoire.

Autre idée : pivot médian

(La médiane de $2k+1$ valeurs est la $(k+1)^{\text{e}}$ plus grande valeur.)

L'idée : on choisit comme pivot pour la partition la médiane de quelques valeurs du tableau.

Par exemple la médiane des trois premières valeur.
Ou mieux (pourquoi ?) : la médiane des $A[l]$, $A[\lfloor \frac{l+r}{2} \rfloor]$ et $A[r]$.

Conclusion sur le tri rapide

- On a $T(n) = \mathrm{O}(n^2)$ de façon générale, mais $T(n) = \Theta(n \log n)$ en moyenne sur des entrées distribuées uniformément.
- En pratique le tri rapide est plus rapide que les autre tris présentés jusqu'à présent pour des valeurs de n pas ridiculement petites.
- Pour les petites valeurs, le tri par insertion est un bon choix.
- L'implémentation de `qsort()` dans la GNU Libc (et d'autres) :
  - Version non-récursive du *tri rapide*.
  - Choix du pivot médian des extrémités et du milieu.
  - Tri par insertion dès que la partition possède ≤ 4 éléments.

Tri introspectif

Origine

David Musser, 1997

Utilisé dans la Standard Template Library de SGI. (`std::sort`)

Intérêt

Modification du tri rapide de façon à ce que $T(n) = \Theta(n \log n)$ toujours. Donc : tri en place [2] et instable.

Idée

Détecter les séquences qui posent problème au tri rapide, et effectuer un tri par tas dans ce cas.

En pratique

On borne le nombre d'appels récursifs à $\mathrm{O}(\log n)$.

Musser suggère $2\lfloor \log n \rfloor$.

2. prétendument, comme pour le QuickSort

Tri introspectif : algorithme

```
IntroSort(A, l, r)
1  IntroSort'(A, l, r, 2⌊log(r − l + 1)⌋)

IntroSort'(A, l, r, depth_limit)
1  if depth_limit = 0 then
2      Heap-Sort(A, l, r)
3      return
4  else
5      depth_limit ← depth_limit − 1
6      p ← Partition(A, l, r)
7      IntroSort'(A, l, p, depth_limit)
8      IntroSort'(A, p + 1, r, depth_limit)
```


Résumé des complexités

	complexité	en moyenne
Tri par insertion	$\mathrm{O}(n^2)$	$\Theta(n^2)$
Tri fusion	$\Theta(n \log n)$	
Tri par sélection	$\Theta(n^2)$	
Tri par tas	$\mathrm{O}(n \log n)$	
Tri rapide	$\mathrm{O}(n^2)$	$\Theta(n \log n)$
Tri introspectif	$\mathrm{O}(n \log n)$	

On sait donc que le problème du tri comparatif est en $\mathrm{O}(n \log n)$. On va maintenant montrer que ce problème est en $\Omega(n \log n)$, c'est-à-dire qu'un tri comparatif ne peut pas être moins complexe dans le cas général.

Borne inférieure sur le tri comparatif

Rappel du problème

Entrée : Une séquence de n nombres $\langle a_1, a_2, \ldots a_n \rangle$

Sortie : Une permutation $\langle a'_1, a'_2, \ldots a'_n \rangle$ de la séquence d'entrée telle que $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Argumentation

Les tris peuvent être représentés par un arbre binaire (dit « de décision »). Les nœuds internes sont des comparaisons entre deux éléments : le fils gauche désigne une réponse négative et le fils droit une réponse positive. Les feuilles représentent la permutation à effectuer pour obtenir le tableau trié.

Il existe $n!$ permutations possibles de $\langle a_1, a_2, \ldots a_n \rangle$. Notre arbre binaire possède $n!$ feuilles, sa hauteur est donc au moins $\lceil \log n! \rceil$. On en déduit $T(n) = \Omega(\log(n!)) = \Omega(n \log n)$.

Rappel de la formule de Stirling : $n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$.

Tris linéaires

3 Tris comparatifs
- Définition
- Rappel des tris par insertion et fusion
- Tri par sélection
- Tri par tas
  - Structure de tas
  - Opérations sur le tas
- Tri rapide
- Tri introspectif
- Bornes de complexité des tris comparatifs

4 Tris linéaires
- Tri par dénombrement
- Tri par paquets

5 Mélange aléatoire

6 Recherche dans un tableau trié

7 Rangs et médians
- Minimum
- Sélection
- Sélection stochastique
- Sélection en $\mathrm{O}(n)$

Tri par dénombrement

Caractéristique

Tri stable, mais pas en place.

Utilisable uniquement si les éléments en entrée sont dans un petit intervalle. Ici on les suppose dans $[\![1, k]\!]$.

Algorithme

CountingSort(A, B, k)

1	for $i \leftarrow 1$ to k do $C[i] \leftarrow 0$	$\Theta(k)$
2	for $i \leftarrow 1$ to $\mathit{length}(A)$ do $C[A[j]] \leftarrow C[A[j]] + 1$	$\Theta(n)$
3	for $i \leftarrow 1$ to k do $C[i] \leftarrow C[i] + C[i-1]$	$\Theta(k)$
4	for $i \leftarrow \mathit{length}(A)$ down to 1 do	$\Theta(n)$
5	$\quad B[C[A[i]]] \leftarrow A[i]$	$\Theta(n)$
6	$\quad C[A[i]] \leftarrow C[A[i]] - 1$	$\Theta(n)$
		$\Theta(n) + \Theta(k)$

Complexité

Si $k = \mathrm{O}(n)$, alors $T(n) = \Theta(n)$.

Tri par paquets

Caractéristique

Tri ni stable, ni en place. Suppose les éléments de l'entrée répartis uniformément. Ici on considère l'intervalle $[0, 1[$.

Algorithme

BucketSort(A)

1	$n \leftarrow \mathit{length}(A)$	$\Theta(n)$
2	for $i \leftarrow 1$ to n do	$\Theta(n)$
3	$\quad$ insert $A[i]$ into the list $B[\lfloor n \cdot A[i] \rfloor]$	$\Theta(n)$
4	for $i \leftarrow 0$ to $n-1$ do	$\Theta(n)$
5	$\quad$ sort $B[i]$ with InsertionSort	$\sum_{i=0}^{n-1} \mathrm{O}(n_i^2)$
6	concatenate $B[0]$, $B[1]$, …, $B[n-1]$ together in order	$\Theta(n)$

Complexité

Tout dépend de la taille n_i des $B[i]$ à trier.

Complexité du tri par paquets

Cas favorable

Si les $B[i]$ sont de taille $n_i = 1$, les n appels à InsertionSort de la ligne 5 prennent tous $\Theta(1)$.

On obtient une complexité en $\Theta(n)$.

Cas défavorable

Si (1) tous les éléments se retrouvent dans le même paquet, et (2) se paquet est ordonné à l'envers (pire cas pour le tri par insertion), alors la ligne 5 demande $\Theta(n^2)$ opérations.

On obtient une complexité en $\Theta(n^2)$.

Cas moyen

Que vaut n_i en moyenne ?

Que vaut n_i^2 en moyenne ?

Que vaut $\sum_{i=0}^{n-1} \mathrm{O}(E[n_i^2])$?

Rappels probabilistes

Espérance d'une variable aléatoire

C'est sa valeur attendue, ou moyenne.

$$\mathrm{E}[X] = \sum_x x \Pr\{X = x\}$$

Variance

$$\mathrm{Var}[X] = \mathrm{E}[(X - \mathrm{E}[X])^2] = \mathrm{E}[X^2] - \mathrm{E}^2[X]$$

Loi binomiale

On lance n ballons dans r paniers. Les chutes dans les paniers sont équiprobables ($p = 1/r$). On note X_i le nombre de ballons dans le panier i. On a $\Pr\{X_i = k\} = \mathrm{C}_n^k p^k (1-p)^{n-k}$. On peut montrer $E[X_i] = np$ et $\mathrm{Var}[X_i] = np(1-p)$.

Tri par paquet : étude probabiliste

Notons n_i la taille d'un paquet à trier avec un tri par insertion.

Si les valeurs à triées sont uniformément réparties leur répartitions dans les différents paquets sont équiprobable. On se retrouve dans la situation d'envoyer n ballons dans n paniers ($p = 1/n$).

On a donc $E[n_i] = np = 1$ et $\mathrm{Var}[n_i] = np(1-p) = 1 - \frac{1}{n}$.

Le tri par insertion de n éléments prend $\mathrm{O}(n^2)$. Pour tous les $B[i]$ on a

$$\sum_{i=0}^{n-1} \mathrm{O}(E[n_i^2]) = \mathrm{O}\left(\sum_{i=0}^{n-1} \mathrm{E}[n_i^2]\right) = \mathrm{O}\left(\sum_{i=0}^{n-1} \left(\mathrm{E}^2[n_i] + \mathrm{Var}[n_i]\right)\right)$$
$$= \mathrm{O}\left(\sum_{i=0}^{n-1} \left(1 + 1 - \frac{1}{n}\right)\right) = \mathrm{O}(n)$$

Finalement $T(n) = \mathrm{O}(n) + \Theta(n) = \Theta(n)$ en moyenne.

Mélange aléatoire

3 Tris comparatifs
- Définition
- Rappel des tris par insertion et fusion
- Tri par sélection
- Tri par tas
  - Structure de tas
  - Opérations sur le tas
- Tri rapide
- Tri introspectif
- Bornes de complexité des tris comparatifs

4 Tris linéaires
- Tri par dénombrement
- Tri par paquets

5 Mélange aléatoire

6 Recherche dans un tableau trié

7 Rangs et médians
- Minimum
- Sélection
- Sélection stochastique
- Sélection en $\mathrm{O}(n)$

Mélange naïf

Entrée : un tableau A
Sortie : une copie B mélangée

```
NaiveRandomizeArray(A)
1  for i ← 1 to length(A) do C[i] ← 0
2  for i ← 1 to length(A) do
3      do
4          j ←Random(1, length(A))
5      until C[j] = 0
6      C[j] ← 1
7      B[i] ← A[j]
8  return B
```

Notons t_i le nombre d'exécutions de la ligne 4 au sein d'une itération de la boucle principale pour un i donné, et n la taille de A.

Mélange naïf : étude (1/2)

Si i valeurs ont été prises, la probabilité de choisir un j tel que $C[j] = 0$ est $\frac{i}{n}$. La probabilité de tomber sur un mauvais « j » k fois de suite avant d'en trouver un bon est $\Pr\{t_{i+1} - 1 = k\} = \left(\frac{i}{n}\right)^k \left(1 - \frac{i}{n}\right)$.

On en déduit

$$\mathrm{E}[t_{i+1} - 1] = \sum_{k=0}^{\infty} k \Pr\{t_{i+1} - 1 = k\} = \frac{n-i}{n} \sum_{k=0}^{\infty} k \left(\frac{i}{n}\right)^k$$

Pour $|x| < 1$ on sait $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$. En dérivant et en multipliant par x on obtient $\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$.
Il en découle que

$$\mathrm{E}[t_{i+1} - 1] = \frac{n-i}{n} \frac{\frac{i}{n}}{\left(1 - \frac{i}{n}\right)^2} = \frac{n-i}{n} \frac{\frac{i}{n}}{\frac{(n-i)^2}{n^2}} = \frac{i}{n-i}$$

Mélange naïf : étude (2/2)

$$\mathrm{E}[t_i] = 1 + \frac{i-1}{n-i+1} = \frac{n}{n+1-i}$$

Le nombre moyen d'exécutions de la ligne 4 est

$$\sum_{i=1}^{n} \mathrm{E}[t_i] = \sum_{i=1}^{n} \frac{n}{n+1-i} \underset{k=n+1-i}{=} n \sum_{k=1}^{n} \frac{1}{k}$$
$$= n(\ln n + \Theta(1)) = \Theta(n \log n)$$

On en déduit que la complexité moyenne de ce mélange est $T(n) = \Theta(n \log n)$.

Dans le meilleur cas, on a $T(n) = \Theta(n)$ (le random est toujours le bon !).

Dans le pire cas (le random est toujours le mauvais) l'algorithme ne termine pas. Oups !

Mélange plus mieux

Entrée : un tableau A
Sortie : le tableau A mélangé

```
BetterRandomizeArray(A)
1  for i ← 1 to length(A) − 1 do
2      j ←Random(i, length(A))
3      A[i] ↔ A[j]
```

On a évidement $T(n) = \Theta(n)$ dans tous les cas.

Recherche dans un tableau trié

3 Tris comparatifs
- Définition
- Rappel des tris par insertion et fusion
- Tri par sélection
- Tri par tas
  - Structure de tas
  - Opérations sur le tas
- Tri rapide
- Tri introspectif
- Bornes de complexité des tris comparatifs

4 Tris linéaires
- Tri par dénombrement
- Tri par paquets

5 Mélange aléatoire

6 Recherche dans un tableau trié

7 Rangs et médians
- Minimum
- Sélection
- Sélection stochastique
- Sélection en $\mathrm{O}(n)$

Recherche dans un tableau trié

Entrée : un tableau $A[l..r]$, une valeur v
Sortie : un indice $i \in [\![l, r]\!]$ tel que $A[i] = v$ ou 0 si $v \notin A[l..r]$.

```
BinarySearch(A, l, r, v)
 1  if l ≤ r then
 2      m ← ⌊(l + r)/2⌋
 3      if v = A[m] then
 4          return m
 5      else
 6          if v < A[m] then
 7              return BinarySearch(A, l, m − 1, v)
 8          else
 9              return BinarySearch(A, m + 1, r, v)
10  else
11      return 0
```


Recherche dans un tableau trié (compexité)

$$T(n) \leq T(n/2) + \Theta(1)$$

On en déduit $T(n) = \mathrm{O}(\log(n))$.

On peut montrer que cette complexité est optimale.

Concentrons nous sur le cas $v \notin A$. Une preuve que $v \notin A$ est un énoncé de la forme

- $\exists i \in [\![1, n[\![,\ A[i] < v < A[i+1]$ ou
- $v < A[1]$ ou
- $A[n] < v$

Si $v \notin A$, un algorithme de comparaison ne peut s'arrêter que s'il possède une preuve.

Arbre de décision pour une recherche par comparaison (1/2)

L'arbre de décision pour un algorithme de recherche dans un tableau A est un arbre binaire dont les nœuds internes sont étiquetés par des indices de A de la façon suivante :

- la racine est étiquetée par l'indice du premier élément avec lequel A compare v
- si l'étiquette d'un nœud est i, l'étiquette du fils gauche est l'indice du nœud avec lequel l'algorithme compare v si $v < A[i]$, l'étiquette du fils droit est l'indice du nœud avec lequel l'algorithme compare v si $v > A[i]$.

Les feuilles de A sont étiquetées par les preuves que $v \notin A$.

(Cet arbre peut être étendu en rajoutant une feuille a chaque sommet interne étiqueté par i. Cette feuille correspond au résultat $v = A[i]$.)

Arbre de décision pour une recherche par comparaison (2/2)

La complexité en pire cas de l'algorithme est la longueur de la branche la plus longue de cet arbre de décision.

Supposons par l'absurde qu'il y ait (strictement) moins de n sommet internes. Dans ce cas il existe un indice $i \in [\![1, n]\!]$ qui n'étiquette aucun nœud de l'arbre.
Imaginons deux tableaux A et A' tels que $\forall j \neq i,\ A[j] = A'[j] \neq v$ et $A[i] = v$ et $A'[i] \neq v$.
L'algorithme de comparaison (représenté par l'arbre de décision) ne peut distinguer ces deux tableaux, donc il donne un résultat incorrect.
Le nombre de nœuds internes est donc supérieur à n.

Le nombre de comparaisons effectuées dans le pire cas est donc supérieur à $\lfloor \log n \rfloor$.

On a prouvé $T(n) = \Omega(\log n)$ en pire cas.

Rangs et médians

3 Tris comparatifs
- Définition
- Rappel des tris par insertion et fusion
- Tri par sélection
- Tri par tas
  - Structure de tas
  - Opérations sur le tas
- Tri rapide
- Tri introspectif
- Bornes de complexité des tris comparatifs

4 Tris linéaires
- Tri par dénombrement
- Tri par paquets

5 Mélange aléatoire

6 Recherche dans un tableau trié

7 Rangs et médians
- Minimum
- Sélection
- Sélection stochastique
- Sélection en $\mathrm{O}(n)$

Recherche du minimum

```
Minimum(A)
1  min ← A[1]
2  for i ← 2 to length(A) do
3      if min > A[i] then
4          min ← A[i]
5  return min
```

On a $T(n) = n - 1 = \Theta(n)$ comparaisons.

On en déduit que le problème de la recherche du miminum est en $\mathrm{O}(n)$. Est-il possible de faire mieux ?

La recherche du minimum peut être vu comme un tournoi, remporté par le plus petit élément. Chaque comparaison est un match. Tout élément hormis le vainqueur perdra au moins un match. Il faudrait donc $n - 1$ comparaisons pour trouver le minimum.
Le problème de la recherche du minimum est donc en $\Theta(n)$.

Sélection

Problème de la sélection

Entrée : Un ensemble A de n nombres distincts et un nombre $i \in [\![1, n]\!]$.
Sortie : L'élément $x \in A$ qui est plus grand que $i - 1$ autres éléments de A exactement.

Peut être résolu en $\mathrm{O}(n \log n)$

Il suffit de trier A et retourner $A[i]$.

On peut mieux faire !

Le problème est en $\Theta(n)$.

Sélection stochastique

```
RandomizedSelection(A, l, r, i)
1  if l = r then return A[l]
2  m ← RandomizedPartition(A, l, r)
3  k ← m − l + 1
4  if i ≤ k
5      then return RandomizedSelection(A, l, m, i)
6      else return RandomizedSelection(A, m + 1, r, i − k)
```

Pire cas

La partition crée systématiquement deux ensembles de taille 1 et $r - l$, plus grand contenant l'élément recherché.
$T(n) = \Theta(n) + T(n-1) = \Theta(n^2)$.
Cependant aucune entrée ne donne à coup sûr ce pire cas.

Cas favorable

$T(n) = \Theta(n) + T(9n/10)$. $a = 1$, $b = 10/9$. $n^{\log_{10/9} 1} = n^0$.
Cas 3 du Th. général. La régularité est respectée, et $T(n) = \Theta(n)$.

Sélection stochastique : cas moyen (1/4)

La partition coupe $A[1..n]$ en $A[1..i]$ et $A[i+1..n]$ avec $n-1$ choix possibles pour i.

$$T(n) \leq \frac{1}{n-1}\sum_{i=1}^{n-1} \max(T(i), T(n-i)) + \Theta(n)$$
$$\leq \frac{1}{n-1}\sum_{i=1}^{n-1} T(\max(i, n-i)) + \Theta(n)$$

$$\text{On a } \max(i, n-i) = \begin{cases} i & \text{si } i \geq \lceil n/2 \rceil \\ n-i & \text{si } i \leq \lfloor n/2 \rfloor \end{cases}$$

Si n est pair, tous les termes entre $T(\lceil n/2 \rceil)$ et $T(n)$ apparaissent deux fois. Si n est impair, $T(\lfloor n/2 \rfloor)$ est seul en plus.

Sélection stochastique : cas moyen (2/4)

$$T(n) \leq \frac{2}{n-1}\sum_{i=\lfloor n/2 \rfloor}^{n-1} T(i) + \Theta(n)$$

On veut montrer que $T(n) = \mathrm{O}(n)$. On va faire une démonstration par récurrence. Le principe :

- On suppose que $T(m) \leq cm$ pour une constante c et pour tout m tel que $n_0 \leq m < n$ avec un n_0 choisi. (On peut choisir c et n_0 aussi grands que l'on veut pour aider la preuve.)
- On démontre $T(n) \leq cn$.
- On en déduit $T(n) \leq cn$ pour tous les $n > n_0$ si $T(n) = \mathrm{O}(1)$ pour $n \leq n_0$.

Sélection stochastique : cas moyen (3/4)

Lemme

$$\sum_{k=\lfloor n/2 \rfloor}^{n-1} k \leq \frac{3}{8}n^2$$

Preuve

$$\sum_{k=\lfloor n/2 \rfloor}^{n-1} k = \frac{(n-1+\lfloor n/2 \rfloor)(n-\lfloor n/2 \rfloor)}{2} = \frac{n^2 - n + \lfloor n/2 \rfloor - \lfloor n/2 \rfloor^2}{2}$$

Si n est pair :

$$= \frac{4n^2 - 4n + 2n - n^2}{8} = \frac{3n^2 - 2n}{8} \leq \frac{3n^2}{8}$$

Si n est impair :

$$= \frac{4n^2 - 4n + 2(n-1) - (n-1)^2}{8} = \frac{3n^2 - 3}{8} \leq \frac{3n^2}{8}$$

Sélection stochastique : cas moyen (4/4)

$$T(n) \leq \frac{2}{n-1}\sum_{i=\lfloor n/2 \rfloor}^{n-1} T(i) + \Theta(n)$$

On suppose $T(i) \leq ci$ pour $i \leq n$:

$$T(n) \leq \frac{2}{n-1}\sum_{i=\lfloor n/2 \rfloor}^{n-1} ci + \Theta(n) \quad T(n) \leq \frac{2c}{n-1}\frac{3n^2}{8} + \Theta(n)$$
$$T(n) \leq \frac{3cn(n-1) + 3cn}{4(n-1)} + \Theta(n) \leq \frac{3cn}{4} + \frac{3c(n-1) + 3c}{4(n-1)} + \Theta(n)$$
$$T(n) \leq cn - \frac{cn}{4} + \frac{3c}{4} + \frac{3c}{4(n-1)} + \Theta(n) \leq cn + \left(\Theta(n) - \frac{cn}{4}\right)$$

c peut être choisi assez grand pour que $cn/4$ domine le $\Theta(n)$

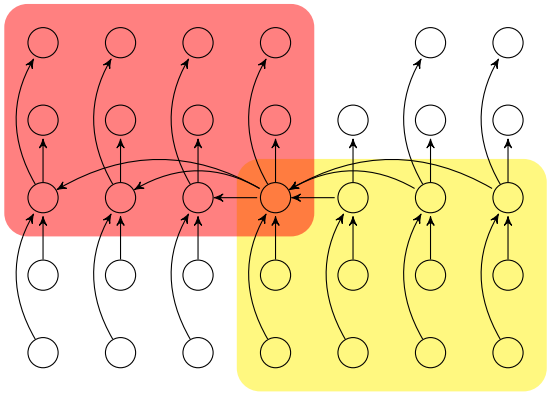
$T(n) \leq cn$ et on en déduit $T(n) = \mathrm{O}(n)$

Sélection en $\mathrm{O}(n)$

Selection :

- Diviser les n éléments en $\lfloor n/5 \rfloor$ groupes de 5 éléments et un groupe de $n \mod 5$ éléments.
- Calculer le médian de chaque groupe.
- Appeler Selection récursivement pour trouver le médian des médians.
- Partitionner le tableau d'entrée autour du médian des médians en utilisant une version modifiée de Partition. Notons k le nombre d'éléments de la région inférieure.
- Appeler Selection récursivement pour trouver le i^{e} plus petit élément de la région inférieure si $i \geq k$, ou le $(i-k)^{\text{e}}$ plus petit élément de la région supérieure si $i > k$.

Sélection en $\mathrm{O}(n)$: étude (1/2)



À la louche... Au moins un quart des éléments sont plus petits que le médian et un quart des éléments sont plus grands. (On peut être plus précis.) Dans le pire cas, la partition retourne donc un sous-tableau de $n/4$ éléments et un autre de $3n/4$ éléments.

Sélection en $\mathrm{O}(n)$: étude (2/2)

$$T(n) \leq \underbrace{\Theta(n)}_{\text{calculs des } \lceil n/5 \rceil \text{ médians}} + \underbrace{T(n/5)}_{\text{médian des médians}} + \underbrace{\Theta(n)}_{\text{partition}} + \underbrace{T(3n/4)}_{\text{sélection récursive}}$$

Sans le terme $T(n/5)$ on pourrait appliquer le théorème général est obtenir $T(n) = \mathrm{O}(n)$. On va montrer cette borne par récurrence : On substitue cn à $T(n)$:

$$T(n) \leq \Theta(n) + \frac{cn}{5} + \frac{3cn}{4} = \Theta(n) + \frac{19cn}{20} \leq cn + \left(\Theta(n) - \frac{cn}{20}\right)$$

c peut être choisi assez grand pour que $\frac{cn}{20}$ domine le $\Theta(n)$

$$T(n) \leq cn$$

Donc $T(n) = \mathrm{O}(n)$.

Conclusion sur la sélection

- Question : Nous avons fait des groupes de 5 valeurs. Aurions-nous pu faire des groupes de 3 ? Des groupes de 7 ?
- L'algorithme de sélection en $\mathrm{O}(n)$ montre que ce problème est en $\mathrm{O}(n)$. C'est intéressant théoriquement, mais en pratique il est *généralement* plus rapide d'utiliser la sélection stochastique.
- L'algorithme de sélection stochastique (en $\mathrm{O}(n^2)$) peut être modifié à la manière du Tri Introspectif pour se transformer en « médian des médians » sur les entrées qui posent problème. (Cherchez « `introselect` » sur l'Internet.)