IFT 2010 Hiver 2006

Structures de données

Miklós Csűrös

André-Aisenstadt 3149
csuros@iro.umontreal.ca

http ://www.iro.umontreal.ca/~csuros/IFT2010/

Plan de cours

Université de Montréal

Cours

Répertoire des cours

IFT2010 Structures de données

- Crédits: 4 dont 1 de travaux pratiques - labo
- Durée: 1 trimestre(s)
- Généralement offert à l'automne et à l'hiver
- Période: le jour

Responsables	Faculté des arts et des sciences - Département ou école: Informatique et recherche opérationnelle
Description	Types abstraits pour les structures de données, arbres, dictionnaires, files avec priorités, graphes, méthodes externes.
Préalables	IFT1020 **et** IFT1063

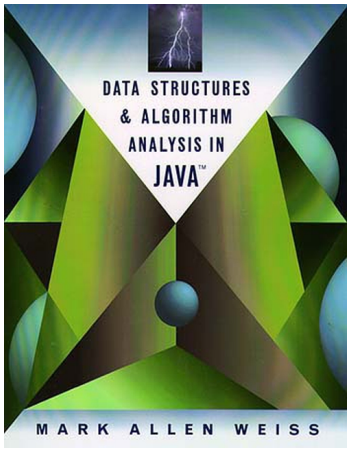
Horaire :

mardi 8 :30–10 :30 AA 1360, jeudi 10 :30–11 :30 AA 1360

mercredi démo 15 :30–17 :30 AA 1411

Matériel

Livre principal : [**Weiss**]



D'autres livres en réserve à Math-Info :

[**AHU**] Aho, Hopcroft, Ullman *Data Structures and Algorithms* («Structures de données et algorithmes»)

[**Drozdek**] *Data Structures and Algorithms in Java*

notes de cours affichés avant ou juste après le cours

devoirs :$4 \times 10\%$ intra :30% final :30%

Sujets

- Analyse d'algorithmes, notation *big-Oh* [**Ch. 2**]
- Listes et piles [**Ch. 3**]
- Arbres — binaires, balancés et *splay* [**Ch. 4**]
- Hachage [**Ch. 5**]
- Tas — *heap* ou *priority queue* [**Ch. 6**]
- Exemple d'algos — tris [**Ch. 7**]
- Exemple de SD : ensemble [**Ch. 8**]
- Graphes — SD et algo [**Ch. 9**]
- Dessert

Exemple

Problème : triage de nombres entiers

Entrée : un ensemble de nombres IN

Sortie : tableau `out` des nombres de IN en ordre croissant

P.e. : $\mathsf{IN} = \{3, 5, 8, 1, 7\}$, `int[] out` $= \{1, 3, 5, 7, 8\}$

Noter : je n'ai pas défini la représentation de IN (par contre, `out` est un tableau d'entiers `int` en Java)

Tri A

on utilise une liste OUT qui préserve l'ordre d'addition des éléments

```
A1 initialiser OUT ← ∅          // OUT est vide au début
A2 tandis que IN ≠ ∅
A3     m ← min IN
A4     enlever m de IN
A5     ajouter m à la fin de OUT
A6 retourner OUT transformé en int[]
```

Questions

1. Implantation de IN : doit supporter
   - (initialisation lors de la lecture de l'entrée)
   - vérification si vide
   - choix de min
   - suppression d'un élément
2. Implantation de OUT : doit supporter
   - ajout d'un élément
   - transformation en `int[]`

Solution simple

Supposons que IN est implanté par `int[] in`

- vide ? : `in.length == 0`
- choix de min : `for (i=0;i<in.length; i++){...}`
- suppression d'un élément : (copier dans un autre tableau)

Solution de l'informaticien/ne

Recursion !

- on peut trier IN avec < 2 éléments sans problème
(exercice à la maison . . .)

- récursion : trier deux moitiés séparamment + fusionner les listes triées

Tri B

implantation de IN : `int[]`

```
B1 si in.length < 2 alors retourner in
B2 int[] in1 ← {in[0],in[2],in[4],...}
B3 int[] in2 ← {in[1],in[3],in[5],...}
B4 int[] out1 ← triB(in1); int[] out2 ← triB(in2)
B5 int[] out ← fusion(out1,out2)
B6 retourner out
```

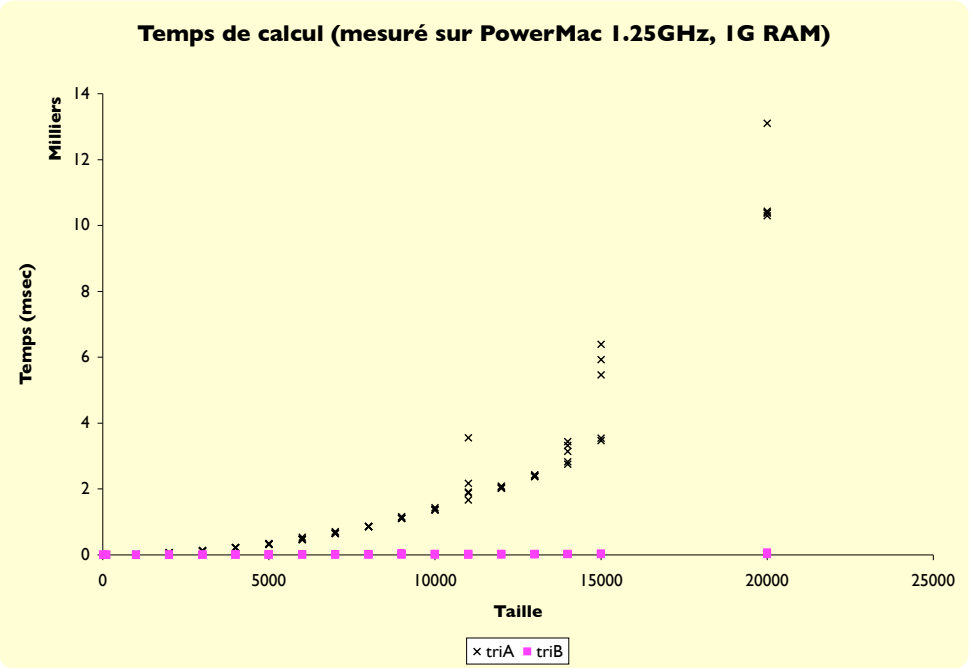
`int[] fusion(int[] arr1, int[] arr2)` prend deux tableaux triés est calcule leur fusion triée

Vitesse d'exécution

On peut mesurer le temps d'exécution des deux approches

```
...
long T0 = System.currentTimeMillis(); // temps de début
...
long dT = System.currentTimeMillis()-T0; // temps (ms) dépassé
```

Vitesse d'exécution 2



triB a l'air d'être meilleur . . .

Vitesse d'exécution 3

Le temps d'exécution dépends de beaucoup de conditions :

- entrée
- ordinateur
- environnement *run-time*
- température
- phase de la lune
- [...]

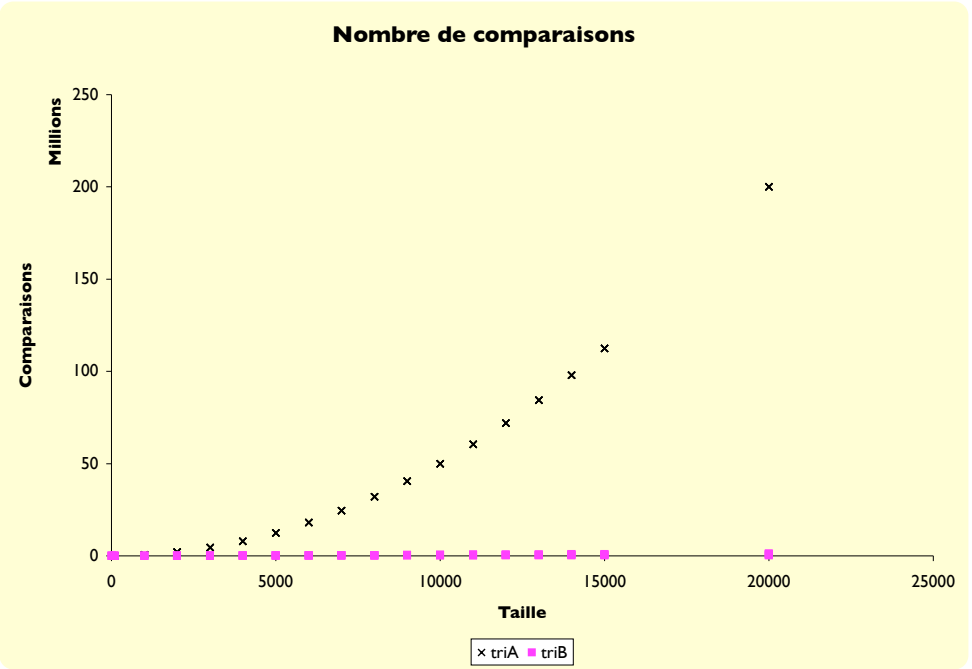
Ce qui nous intéresse, c'est la dépendance de l'entrée

$\Rightarrow$ abstraction de l'implantation

p.e., calculer nombre d'opérations «primitives»

Temps de calcul

mesuré ici en nombre de comparaisons entre éléments de IN



triB a toujours l'air d'être meilleur . . . indépendamment du plate-forme

Temps de calcul 2

But : preuve formelle pour comparer les nombre d'opérations

Thm. Nombre de comparaisons performés par tri A est $A(n) = n(n-1)/2$ si $|\mathsf{IN}| = n$ et on utilise l'implantation par `int[]`.

Preuve. $A(n)$ est déterminé par le nombre de comparaisons utilisés pour min.

```
int min_pos = 0;
for (int i=1; i<in.length; i++)
   if (in[i]<in[min_pos]) min_pos=i;
// maintenant in[min_pos] est le min.
```

Nombre de comparaisons : `in.length` $- 1$

Preuve (cont.) Nombre total de comparaisons :

$$(n-1)+(n-2)+(n-3)+\cdots+(2-1)+0=\frac{n(n-1)}{2}.$$

□

Temps de calcul 3

Théorème similaire pour le nombre de comparaisons par tri B ?

- $A(n)$ ne dépend pas de l'ordre des éléments en `in[]`

- pour tri B, ce nombre dépend de l'ordre

$\Rightarrow$ il faut décider si on veut savoir le pire temps ou le temps moyen ou le meilleur temps ou . . .

Noter : notion du «moyen» dépend de la distribution [probabiliste] des entrées !

Fusion de deux tableaux

```
1  Entrée A1, A2 (de type int[])
2  intialiser Sortie
3  i1 ← 0, i2 ← 0
4  while (i1 < A1.length et i2 < A2.length)
5        if (A1[i1] ≤ A2[i2])
6        then ajouter A1[i1] à la fin de Sortie, i1 ← i1 + 1
7        else ajouter A2[i2] à la fin de Sortie, i2 ← i2 + 1
8  while (i1 < A1.length) ajouter A1[i1] à la fin de Sortie, i1 ← i1 + 1
9  while (i2 < A2.length) ajouter A2[i2] à la fin de Sortie, i2 ← i2 + 1
10 return Sortie
```

Implantation de Sortie par `int[]` :

initialiser par

```
int[] Sortie = new int[A1.length+A2.length]; int sortie_idx=0;
```

ajouter x par

```
Sortie[sortie_idx]=x; sortie_idx++;
```

Temps de calcul 4

Thm. Nombre de comparaisons d'éléments performés par tri B est inférieur à $B(n) = n\lceil \lg n \rceil$. $(n > 0)$

Preuve. Nombre de comparisons pour la fusion de `out1` et `out2` :

$$\leq \texttt{out1.length} + \texttt{out2.length} - 1$$

(Lemme démontré au cours.)

Nombre total de comparaisons sur chaque niveau de récurrences est $\leq n$

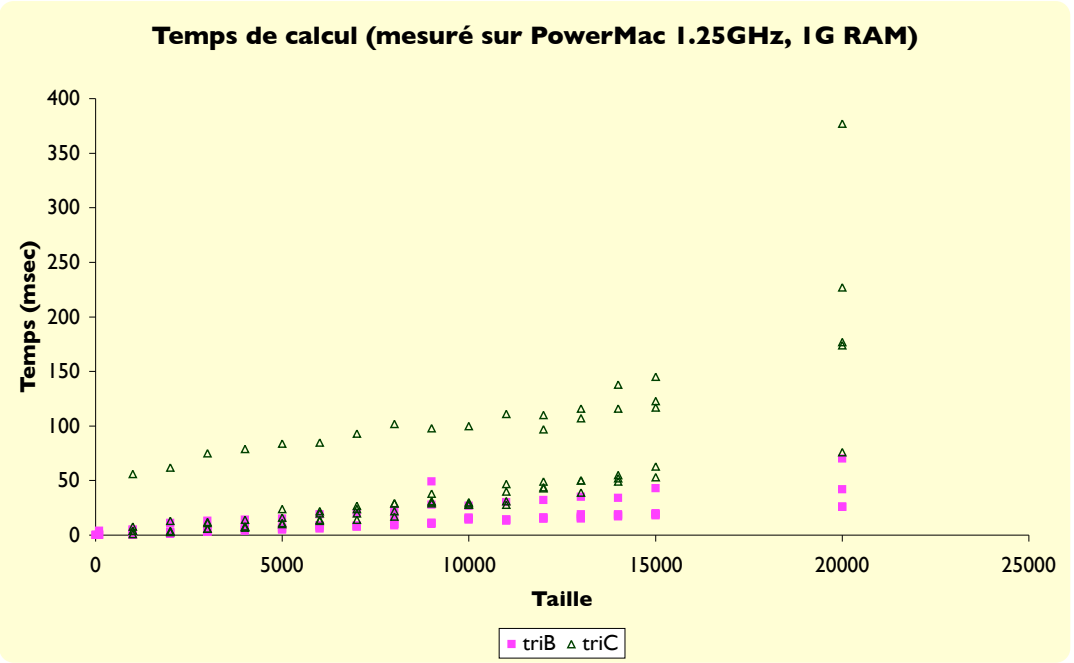
Nombre total de niveaux de récurrences : $\lceil \lg(n) \rceil$.

□

Remarque : on utilise $\lg n$ pour dénoter $\log_2(n)$

Structure de données

tri C : même algorithme que tri A mais on utilise `java.util.TreeSet` pour implanter IN



Structure de données 2

Pourquoi tri B et tri C ont essentiellement le même comportement ?

Parce que `TreeSet.first()` et `TreeSet.remove()` s'exécutent en temps de $f(n) + c \cdot \lg(n)$ avec $c > 0$ et $f(n) \ll \lg(n)$.

(Arbre binaire balancé : on verra dans ce cours comment le faire)

Notation asymptotique

«Croissance essentielle» d'une fonction :

$3n^2 - 2n + 1$ est de la même croissance essentielle que $12n^2 + 3 \lg n$

Déf. $f(n) \in O\Big(g(n)\Big)$ si et seulement si il existe $N \in \mathbb{N}$ et $c \in \mathbb{R}^+$ t.q. pour tout $n \geq N$,

$$f(n) \leq c \cdot g(n).$$

Noter : on écrit souvent $f(n) = O\Big(g(n)\Big)$ mais en vérité $O(g(n))$ est un *ensemble* de fonctions.

Exemples. . .

$2n + 1$ — linéaire
$3n^2 + \lg n$ — quadratique
$2^n + n$ — exponentiel
$4 \cdot 2^n$ et $7 \cdot 3^n$
$4n \log_7 n + n$
$3 \cdot (\log_5 n)^2$

Notez que $2n + 3 \in O(0.0001n + \log_7 n - \sqrt{n})$ est correct par notre définition.

En général, on choisit la fonction «la plus simple» dans les parenthèses de $O(\cdot)$ [ou o, Ω, Θ]
$\Rightarrow$ dans des devoirs, exercices, etc, seulement la forme la plus simple vaut 100% de crédit.

Notation asymptotique 2

Déf. $f(n) \in \Omega\Big(g(n)\Big)$ si et seulement si $g(n) \in O\Big(f(n)\Big)$.

Déf. $f(n) \in \Theta\Big(g(n)\Big)$ si et seulement si $f(n) \in O\Big(g(n)\Big) \cap \Omega\Big(g(n)\Big)$.

Déf. $f(n) \in o\Big(g(n)\Big)$ si et seulement si pour tout $c \in \mathbb{R}^+$ il existe $N(c) \in \mathbb{N}$ t.q. pour tout $n \geq N(c)$,

$$f(n) \leq c \cdot g(n)$$

Notation asymptotique 3

Thm. $f(n) \in o(g(n))$ ssi

$$f(n) \in O(g(n)) \setminus \Theta(g(n)) = O(g(n)) \setminus \Omega(g(n)).$$

Thm. $f(n) \in o(g(n))$ ssi

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

Thm. S'il existe $c \in \mathbb{R}^+$ t.q.

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c,$$

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

Notation asymptotique 4

Thm Si $f_1(n) \in O(g_1(n))$ et $f_2(n) \in O(g_2(n))$, alors

$$f_1(n) + f_2(n) \in O\Big(g_1(n) + g_2(n)\Big)$$

$$f_1(n) \cdot f_2(n) \in O\Big(g_1(n) \cdot g_2(n)\Big).$$

Thm Si $f(n) \in O(g(n))$ alors $f(n) + g(n) \in O(g(n))$.

Thm $\ln n \in o(n^\epsilon)$ pour tout $\epsilon > 0$

Analyse de temps

```
public static int somme_cubes(int n){
    int somme=0;
    for (int i=1; i<n; i++) somme += i*i*i;
    return i;
}
```

Règle 1. Temps de calcul pour boucle : nombre d'exécutions ⋆ temps d'exécution.

$$n \cdot O(1) = O(n)$$

Analyse de temps 2

```
...
for(int i=0; i<n; i++)
   for (int j=0; j<n; j++)
      k+= i*j;
...
```

Règle 1bis. Boucles imbriqués.

Analyse de temps 3

Règle 2. Étapes consécutives.

Règle 3. if/else

Analyse de temps 4

Thm Temps de calcul pour tri A est $O(n^2)$ et pour tri B est de $O(n \log n)$

Analyse de temps 5

Exemples de *subsequence sum*

[Weiss 2.4.3 ou l'article de Jon Bentley : «Algorithm design techniques», *Communications of the ACM*, **27**(9) :865–871 (1984 September).]

Algorithme d'Euclid

```
E1  Algo gcd(a, b)
E2  tandis que (b ≠ 0)
E3      c ← a mod b
E4      a ← b
E5      b ← c
E6  retourner a
```

(on impose $b \leq a$)

Déf. Nombres Fibonacci : $F(0) = 0; F(1) = 1; F(n+1) = F(n) + F(n-1)$ si $n > 0$.

Lemme. Si $x < F(n+1)$ et $y \geq F(n)$ alors $x \bmod y < F(n-1)$.

Analyse de l'algorithme : dénotons les valeurs des variables (juste après ligne E3) par $\langle a_1, b_1, c_1 \rangle$, $\langle a_2, b_2, c_2 \rangle$, ...

Algorithme d'Euclid 2

Lemme. Si $b_i < F(n)$ alors $b_{i+1} < F(n-1)$ ou $b_{i+2} < F(n-2)$.

Si $b_i < F(3) = 2$, alors le boucle ne sera plus executé.

$\Rightarrow$ Si $b < F(n)$, alors le boucle est executé $\leq (n-2)$ fois.

Notez que $n = O(\log b)$ suffit pour cette borne : $F(n) \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n$

$\Rightarrow$ temps de calcul (opérations sur bits) est de

$$O(\log^3 a)$$

donc polynomial dans la taille de l'entrée.

[ici, on suppose que calculer $a \bmod b$ prend $O(\log^2 a)$ temps . . . en fait, on peut le faire plus rapidement]