IFT 2015 Hiver 2009

Structures de données

Miklós Csűrös

André-Aisenstadt 3149
CSUROS@IRO.UMONTREAL.CA

http ://www.iro.umontreal.ca/~csuros/IFT2015/

Déscription du cours

Université de Montréal

Cours

Répertoire des cours

IFT2015 Structures de données

- Crédits: 3 dont 1 de travaux pratiques - labo
- Durée: 1 trimestre(s)
- Généralement offert à l'automne, à l'hiver et à l'été
- 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 (IFT1025 **et** IFT1065) **ou** (IFT1020 **et** IFT1063)

Horaire Automne Hiver Été

Présent dans programmes
1-175-1-0 Baccalauréat en informatique
1-175-2-0 Majeur en informatique
1-175-4-0 Mineur en informatique
1-190-1-0 Baccalauréat en mathématiques
1-191-1-0 Baccalauréat en mathématiques et informatique
1-205-1-0 Baccalauréat en physique et informatique
1-468-1-0 Baccalauréat en bio-informatique

Dernière modification : 06-01-2007 00:19:47

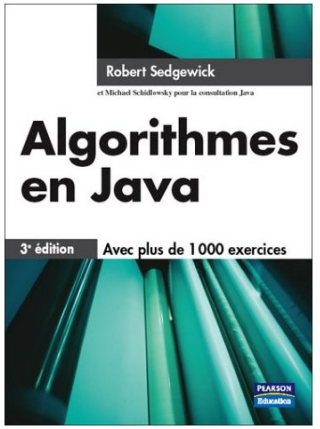
Horaire du cours

Hiver 2009

IFT2015	Structures de données (3 cr.)							
Section								
Activité	Gr.	Jour	De	A	Du	Au	Local	Immeuble
th		Mer.	15:30	16:30	7 janv.	25 févr.	1177	A.-AISENSTADT
th		Lun.	09:30	11:30	12 janv.	9 févr.	1177	A.-AISENSTADT
					23 févr.		1177	A.-AISENSTADT
					9 mars	6 avr.	1177	A.-AISENSTADT
th		Mer.	15:30	16:30	11 mars	15 avr.	1177	A.-AISENSTADT
tp		Mer.	16:30	18:30	7 janv.	25 févr.	1177	A.-AISENSTADT
					11 mars	15 avr.	1177	A.-AISENSTADT
exi		Lun.	09:30	11:30	16 févr.		1360	A.-AISENSTADT
exf		Lun.	09:30	12:30	20 avr.		1360	A.-AISENSTADT
Professeur(e) : Miklos Csuros								

Matériel

Livre principal : [**Sedgewick**]



(80%± du cours vient du livre)

D'autres livres en réserve à Math-Info :
V.O. (en anglais) et Weiss «Data Structures and Algorithm Analysis in Java»

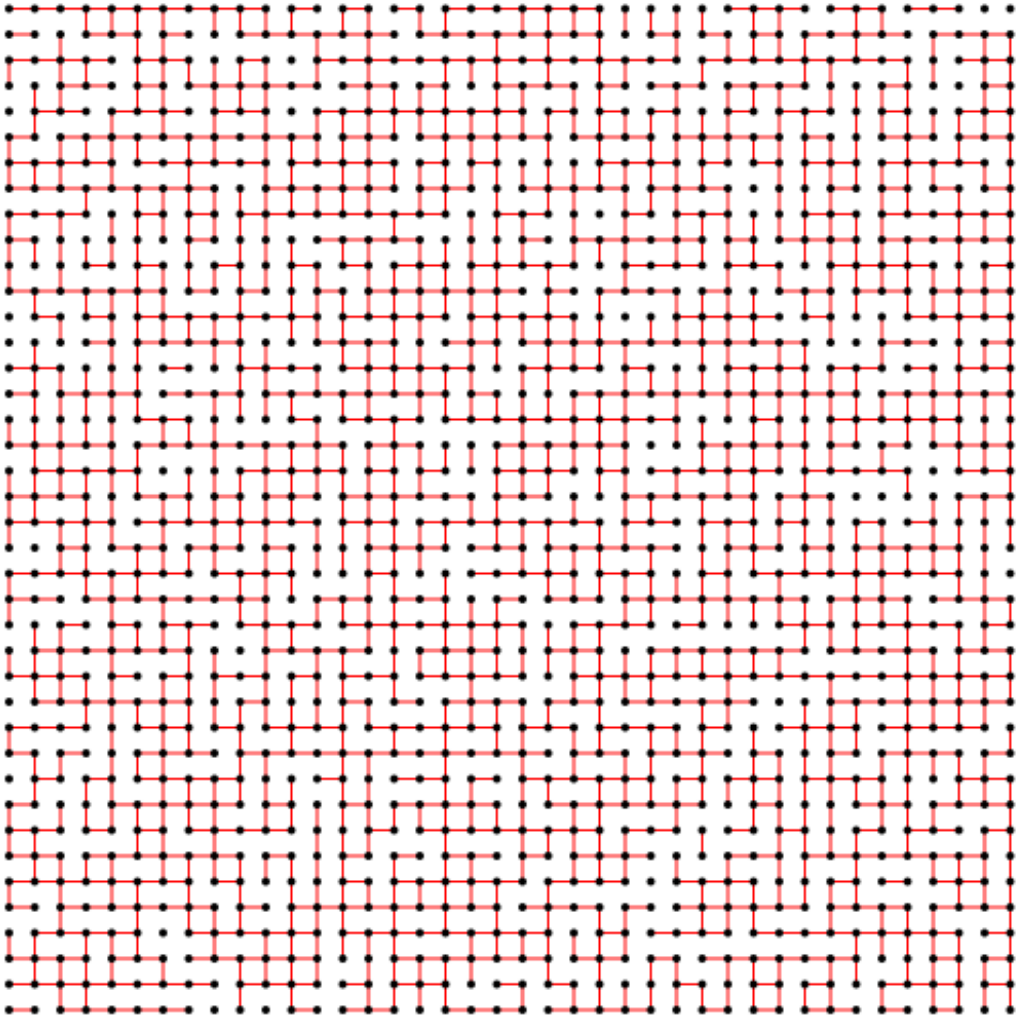
notes de cours affichés avant ou juste après le cours

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

Sujets

- Analyse d'algorithmes, notation grand O [**Ch. 1–2**]
- Listes et piles et files [**Ch. 3–4**]
- Arbres — binaires, equilibrés et *splay* [**Ch. 5, 12–13**]
- Méthodes de tri [**Ch. 6–8**]
- Files à priorités [**Ch. 9**]
- Hachage [**Ch. 14**]
- Graphes — algorithmes de base

Exemple de problème : connexité



Connexité : applications

ordinateurs dans un grand réseau

points de contact dans un réseau éléctrique

équivalence d'éléments (noms de variables)

(Notez qu'on veut seulement savoir si deux éléments sont connexes ou non, et pas les chemins entre eux)

connexité est transitive et réfléxive

→ l'ensemble de tous les éléments peut être décomposé dans des composantes connexes (classes d'équivalence)

Connexité : abstraction

éléments x (p.e., nombres entiers)

opérations abstraites

- FIND(x) trouve l'ensemble contenant x
- UNION(x, y) remplace les ensembles de x et y par leur union
- MAKESET(x) crée un ensemble avec le seul élément x

on réprésente un ensemble par un élément **canonique**

structure de donnée : $\mathsf{id}[x]$ l'ensemble auquel x appartient

```
    MAKESET(x)
1   id[x] ← x
```

QuickF — appartenance rapide

```
    UNION(x, y)
1   si id[x] ≠ id[y] alors
2       pour tout z
3           si id[z] = id[x] alors id[z] = id[y]

    FIND(x)
1   retourner id[x]
```

appartenance rapide — union lente

QuickU — union rapide

idée : on utilise id différemment
1. si $\mathsf{id}[x] = x$ alors x est l'élément canonique de l'ensemble
2. sinon, soit $x \leftarrow \mathsf{id}[x]$ et retourner à 1

(en fait, chaque ensemble forme un arbre)

```
    UNION(x, y)
1   p ← FIND(x) ; q ← FIND(y)
2   si p ≠ q alors
3       id[p] ← q

    FIND(x)
1   tandis que x ≠ id[x] faire x ← id[x]
2   retourner x
```

union rapide — appartenance moins rapide

problème : appels consécutifs $\textsc{union}(1,2)$, $\textsc{union}(2,3)$, $\textsc{union}(3,4)$, ...

QuickUW — union rapide équilibrée

Idée : au lieu de toujours faire $\mathsf{id}[x] \leftarrow y$, lors d'un appel $\textsc{union}(x, y)$, on examine d'abord la taille des deux arbres

Taille stockée dans $\mathsf{taille}[x]$

```
    MAKESET(x)
1   id[x] ← x
2   taille[x] ← 1
```

QuickUW (cont.)

```
    UNION(x, y)
1   p ← FIND(x) ; q ← FIND(y)
2   si p ≠ q alors
3       si taille[p] < taille[q]
4       alors id[p] ← q; taille[q] ← taille[p] + taille[q]
5       sinon id[q] ← p; taille[p] ← taille[p] + taille[q]

    FIND(x)
1   tandis que x ≠ id[x] faire x ← id[x]
2   retourner x
```

avantage : arbres équilibrés (on arrive à la racine en suivant $\leq \lg k$ liens dans un ensemble de k éléments)

Compression de chemin

Si on a n éléments (n appels de MAKESET) et une série de m appels UNION/FIND

QuickF utilise nm opérations
QuickU utilise $nm/2$ opérations
QuickUW utilise $m \lg n$ opérations

Est-ce qu'on peut faire mieux ?

Idée : compression de chemin
quand on monte jusqu'à la racine, faire $\mathsf{id}[x] \leftarrow \mathsf{racine}$ pour tous les membres sur le chemin

(on utilise aussi les tailles des arbres)

Compression de chemin (cont.)

```
    FIND(x)
1   si x ≠ id[x] alors id[x] ← FIND(id[x])
2   retourner id[x]
```

On a besoin de deux passages . . .

autre idée : compression de chemin par réduction à moitié (*path halving*)

```
    FIND(x)
1   tandis que id[id[x]] ≠ id[x] faire id[x] ← id[id[x]] ; x ←
    id[x]
2   retourner id[x]
```

(analyse de performance un peu plus tard. . .)

Analyse de QuickUW

Performance de FIND dépend de la **hauteur** de l'arbre : nombre de liens à suivre jusqu'à ce qu'on arrive à la racine

En maintenant la taille, on contrôle la hauteur des arbres . . .

Principe esquissé : dans le pire cas, on dois joindre deux arbres de la même taille, et la hauteur augmente par un

Est-ce qu'on peut démontrer que hauteur $\leq \lg(\text{taille})$?

(notez : $\lg n = \log_2 n$)

Analyse de QuickUW (cont.)

Théorème Dans la structure QUICKUW, pour chaque arbre, on a

$$\text{hauteur} \leq \mathsf{lg}(\mathsf{taille}). \qquad (\star)$$

Preuve Par induction dans la taille.

Cas de base : quand $\mathsf{taille} = 1$, ($\star$) est vrai.

Hypothèse d'induction : ($\star$) est vrai pour tout $\mathsf{taille} \leq t$.

Cas inductif : Supposons qu'on applique l'opération $\textsc{union}(x, y)$ avec deux arbres de tailles $t_x, t_y \leq t$. (On a $t_x = \mathsf{taille}\big[\textsc{find}(x)\big]$.) h_x, h_y dénotent les hauteurs des arbres. Supposons que $t_x \leq t_y$ (sans perdre la généralité). La hauteur du nouvel arbre est alors $h = \mathsf{max}\{h_y, 1 + h_x\}$.

Analyse de QuickUW (cont2)

Par l'hypothèse d'induction,

$$h \leq \mathsf{max}\{\mathsf{lg}\, t_y, 1 + \mathsf{lg}\, t_x\} = \mathsf{lg}\, \mathsf{max}\{t_y, 2t_x\}.$$

Or, la taille du nouvel arbre est $t_x + t_y \geq \mathsf{max}\{t_y, 2t_x\}$, Donc $h \leq \mathsf{lg}(t_x + t_y)$.

En particulier, un arbre de taille $t + 1$ est crée par UNION de deux arbres de tailles $\leq t$, et donc ($\star$) est vrai pour $\mathsf{taille} = t + 1$. □

Remarque : c'est plus élégant de faire l'induction par le nombre d'opérations UNION.

Vitesse d'exécution

On peut mesurer le temps d'exécution pour comparer la performance des algorithmes

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

Vitesse d'exécution 2

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»

ceci est une fonction de la taille de l'entrée (disons, N)

question typique : quel est l'effet de doubler N ?

Ordre de grandeur des fonctions

- **constant**
- **logarithmique**
- **linéaire**
- $N \log N$
- **quadratique**
- **cubique**
- **[polynomial]**
- **exponentiel**
- **superexponentiel** (p.e, factoriel)
- **non-calculable** (p.e., fonction de Radó)

Logarithmes

$\lg x = \log_2 x$ et $\ln x = \log_e x$ $(x > 0)$

$\log(xy) = \log x + \log y; \log(x^a) = a \log x$

$2^{\lg n} = n; n^n = 2^{n \lg n}; \log_a n = \frac{\lg n}{\lg a}; a^{\lg n} = n^{\lg a}$

Factoriel

$$n! = 1 \cdot 2 \cdots n$$

approximation de Stirling :

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

donc $n!$ est superexponentiel

on voit aussi que $\ln(n!) \approx n \ln n - n + \frac{1}{2} \ln n + \ln \sqrt{2\pi}$, essentiellement $n \ln n$

Ackerman

Ackerman $A(i,j)$ avec $i,j \geq 1$

$$A(i,j) = \begin{cases} 2^j & \text{si } i = 1; \\ A(i-1,2) & \text{si } j = 1 \text{ et } i \geq 2 \\ A(i-1,A(i,j-1)) & \text{si } i,j \geq 2 \end{cases}$$

(définition de Tarjan)

Ackerman inverse ($m \geq n \geq 1$)

$$\alpha(m,n) = \min\left\{i : A(i,\lfloor m/n \rfloor) \geq \lg n\right\}$$

c'est une fonction de croissance très lente
$\alpha(m,n) \leq 3$ pour tout $n < 65536$ et $\alpha(m,n) \leq 4$ pour tout $n < 2^{2^{2^{\cdots^2}}}$
(exponentiation 16 fois)

Logarithme itéré

composition :

$$\lg \lg n = \lg(\lg(n))$$

logarithme itéré

$$\lg^* n = \begin{cases} 0 & \text{si } n \leq 1 \\ 1 + \lg^*(\lg n) & \text{si } n > 1 \end{cases}$$

c'est une fonction très lente (mais à croissance monotone)
$\lg^* n \leq 5$ pour tout $n \leq 2^{65536}$

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 $n \geq N$,

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

La notation $\ldots O(f(n)) \ldots = \ldots O(g(n)) \ldots$ veut dire qu'en remplaçant «$O(f(n))$» à la gauche par n'importe quelle fonction $h(n) \in O(f(n))$, il existe une fonction $h'(n) \in O(g(n))$ telle qu'elle peut remplacer «$O(g(n))$» à la droite pour que l'équation soit vraie

exemple : $n(n+1)/2 + O(\log n) = n^2/2 + O(n)$

Exemples. . .

$2n + 1$: linéaire $O(n)$
$3n^2 + \lg n$: quadratique $O(n^2)$
$2^n + n$: exponentiel (2^n)
$7 \cdot 3^n$: exponentiel $O(3^n)$ mais pas $O(2^n)$
$4n \log_7 n + n$: $O(n \log n)$
$3 \cdot (\log_5 n)^2$: $O(\log^2 n)$

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.

Calculer avec grand O

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 $f(n) + g(n) \in O(g(n))$.

Exemple :

$$\Big(n + O(\sqrt{n})\Big) \cdot \Big(\sqrt{n} + O(1)\Big)$$
$$= n\sqrt{n} + O(n) + O(n) + O(\sqrt{n}) = n^{3/2} + O(n) = O(n^{3/2})$$

D'autres notions asymptotiques

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 $n \geq N(c)$,

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

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

Une astuce avec lim

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

Exemple : récurrence simple

Pour trier la vecteur $v[1..n]$:

1. rechercher le max
2. récurrence sur la reste

Nombre de comparaisons : $C(n) = C(n-1) + n - 1$ si $n > 1$, et $C(1) = 0$

Solution :

$$C(n) = C(n-1) + (n-1) = C(n-2) + (n-2) + (n-1) = \cdots$$
$$= \sum_{i=0}^{n-1} i = n(n-1)/2$$

Temps de calcul : $T(n) = T(n-1) + O(n)$; solution $T(n) = O(n^2)$

Preuve : par induction, en utilisant la définition de O.

Exemple : récurrence 2

$T(n) = T(n/2) + O(1)$; solution $T(n) = O(\log n)$

Exemple : récurrence pour tri

Pour trier la vecteur $v[1..n]$: 1. trier les deux moitiés
2. fusionner les deux vecteurs triées

Temps de calcul : $T(n) = 2 \cdot T(n/2) + O(n)$

Récurrence de base : $C_n = 2C_{n/2} + n; C_1 = 1$
solution : $n(1 + \lg n)$ quand $n = 2^k$

Donc on veut démontrer $T(n) = O(n \log n)$
Preuve : par induction, en utilisant la définition de O