

Algorithmique Avancée

Antoine Genitrini

`Antoine.Genitrini@upmc.fr`

Master Informatique 1

`http://www-master.ufr-info-p6.jussieu.fr/2017`

`http://www-master.ufr-info-p6.jussieu.fr/2017/algav`

Année 2017-2018

Organisation

- **Équipe pédagogique :**

- **Cours** : Antoine Genitrini (mardi 10h45, amphi Herpin)
- **TD-TME** : Philippe Aubry (jeudi matin), Élie de Panafieu (jeudi matin), Maryse Pelletier (mardi après-midi)

- **Planning :**

- 1er cours le mardi 19 septembre
TD : à partir du **26 septembre**.

- **Contrôle Connaissances :**

Session 1

- Examen Réparti 1 (E1) : 13–17 Novembre 2017
- Devoir de Programmation (D) : distribué en octobre, soutenu 19–21 Décembre 2017
- Fin des enseignements le 21 Décembre 2017
- Examen Réparti 2 (E2) : 15–19 Janvier 2018
- Note de Session 1 = $0.2 E1 + 0.2 D + 0.6 E2$

Session 2

- Examen Session 2 (SS) : Juin 2018
- Note Session 2 = SS

Plan du cours

Objectifs : Complexité des algorithmes → Comparer, Optimiser

- **Structures Arborescentes : Files de priorité**
 - Files Binomiales et Files de Fibonacci
 - Coût amorti et Coût moyen
- **Structures Arborescentes pour la Recherche**
 - Arbres de Recherche équilibrés
 - Recherche externe
 - Tries et Arbres Digitaux
- **Classes de complexité et Approximations**
par l'exemple
 - Réduction
 - NP-Complétude
- **Méthodes de Hachage**
 - Hachage interne et externe
 - Hachage universel
 - Hachage cryptographique

Bibliographie

- T. Cormen, C. Leiserson, R. Rivest, C. Stein
Introduction à l'algorithmique
- C. Froidevaux, M-C. Gaudel, M. Soria
Types de données et algorithmes
- D. Beauquier, J. Berstel, P. Chrétienne
Éléments d'algorithmique
- M. Crochemore, C. Hancart, T. Lecroq
Algorithmique du texte

CHAPITRE 0

INTRODUCTION COMPLEXITÉ

- Théorie de la complexité et classification de problèmes :
 - P : ce qui se calcule en temps polynomial $O(n^k)$
 - EXP : ce qui se calcule en temps exponentiel $O(2^n)$
 - NP : intermédiaire (P=NP ?)
- Problèmes exponentiels : optimisation combinatoire, systèmes cryptographiques, ...
- Problèmes polynômiaux : tri, recherche, géométrie, texte, arithmétique, ...

Affiner la classification, comparer les algorithmes, optimiser.

Analyse d'algorithmes

Algorithme \mathcal{A} opère sur des données de \mathcal{E} (mots, arbres, graphes)

taille des données : $\mathcal{E} \rightarrow \mathbb{N}$ (longueur mot, nombre sommet, ...)

Mesure de la complexité de \mathcal{A}

$$\tau_{\mathcal{A}} : \mathcal{E} \rightarrow \mathbb{N}$$

*place mémoire, nombre d'**opérations fondamentales** effectuées*

Analyse de la complexité sur les données de taille n

$$\tau_{\mathcal{A}} : \mathcal{E}_n \rightarrow \mathbb{N}$$

pour comparer les méthodes de résolutions

Ex : multiplication de matrices, tri, recherche, ...

- complexité dans le meilleur des cas (minimale) : $\min\{\tau_{\mathcal{A}}(e); e \in \mathcal{E}_n\}$

- complexité dans le pire cas (maximale) : $\max\{\tau_{\mathcal{A}}(e); e \in \mathcal{E}_n\}$

temps réel, systèmes embarqués

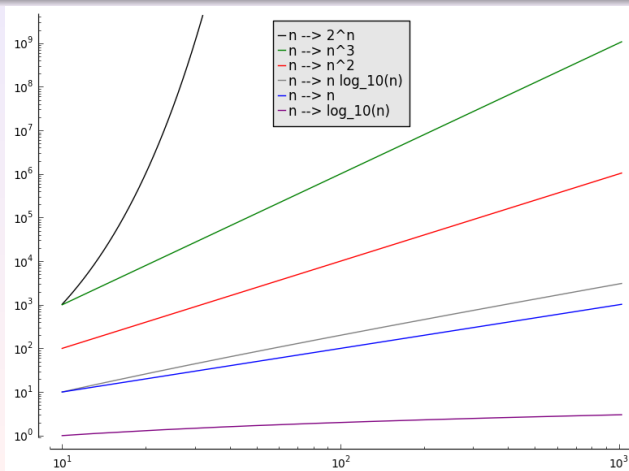
- complexité en moyenne (uniforme) : $\frac{1}{|\mathcal{E}_n|} \sum_{e \in \mathcal{E}_n} \tau_{\mathcal{A}}(e)$

cas typique \rightarrow probabilité des données

Analyse amortie : coût d'une suite d'opérations

Ordre de grandeur asymptotique

$$\begin{array}{llll}
 f \text{ et } g : \mathbb{N} \mapsto \mathbb{R}^+ & f = o(g) & \iff & \lim_{n \rightarrow \infty} f(n)/g(n) = 0 \\
 & f = O(g) & \iff & \exists \alpha \in \mathbb{R}^{+*} \mid \forall n_0 > n, f(n) \leq \alpha \cdot g(n) \\
 & f = \Theta(g) & \iff & f = O(g) \text{ et } g = O(f)
 \end{array}$$



Comparaisons d'ordres de grandeur

Machine faisant de l'ordre de 10^9 opérations/secondes :

	$n = 10$	$n = 10^3$	$n = 10^6$
$\log n$	$\ll \text{sec}$	$\ll \text{sec}$	$\ll \text{sec}$
$10 \log n$	$\ll \text{sec}$	$\ll \text{sec}$	$\ll \text{sec}$
$100 \log n$	$\ll \text{sec}$	$\ll \text{sec}$	$\ll \text{sec}$
n	$\ll \text{sec}$	$\ll \text{sec}$	$\ll \text{sec}$
$10 \cdot n$	$\ll \text{sec}$	$\ll \text{sec}$	$\ll \text{sec}$
$100 \cdot n$	$\ll \text{sec}$	$\ll \text{sec}$	$\ll \text{sec}$
n^2	$\ll \text{sec}$	$\ll \text{sec}$	$\Theta \text{ min}$
$10 \cdot n^2$	$\ll \text{sec}$	$\ll \text{sec}$	$\Theta \text{ heure}$
$100 \cdot n^2$	$\ll \text{sec}$	$\ll \text{sec}$	$\Theta \text{ jour}$
n^3	$\ll \text{sec}$	$\Theta \text{ sec}$	$\Theta \text{ année}$
$10 \cdot n^3$	$\ll \text{sec}$	$\Theta \text{ sec}$	∞
$100 \cdot n^3$	$\ll \text{sec}$	$\Theta \text{ min}$	∞
2^n	$\ll \text{sec}$	∞	∞
$10 \cdot 2^n$	$\ll \text{sec}$	∞	∞
$100 \cdot 2^n$	$\ll \text{sec}$	∞	∞

CHAPITRE 1

FILES de PRIORITÉ

Files binomiales et files de Fibonacci

- 1 Opérations sur les files de priorité
- 2 Arbres binomiaux : *définition et propriétés*
- 3 Files binomiales : *définition et propriétés*
- 4 Union de 2 files binomiales *en temps logarithmique*
- 5 Autres opérations sur les files binomiales
- 6 Analyse en Coût amorti

Opérations sur les files de priorité

Les files de priorité *min* :

- Ensemble d'éléments

- Chaque élément identifié par une clé
- Ordre total sur les clés

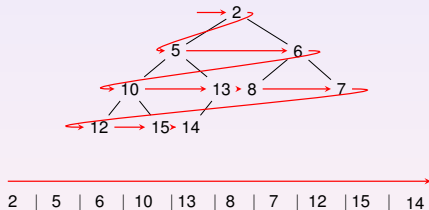
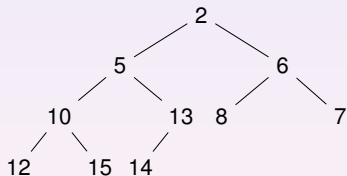
- **Opérations**

- Ajouter un élément
- Supprimer l'élément de clé minimale
- Construction
- Union de 2 files de priorité *min*
- Modification d'une clé

De façon analogue, on peut définir les files de priorité *max*.

Tas

Exemple d'un tas *min* :



Représentations et Efficacité

Nombre de comparaisons dans le pire des cas :

	Liste triée	Tas	File Binomiale
Supp Min (1 élt parmi n)	$O(1)$	$O(\log n)$	$O(\log n)$
Ajout (1 élt parmi n)	$O(n)$	$O(\log n)$	$O(\log n)$
Construction (n élts)	$O(n^2)$	$O(n)$	$O(n)$
Union (n élts et m élts)	$O(n + m)$	$O(n + m)$	$O(\log(n + m))$

Applications des Files de priorité

- Tri par tas (*heapsort*)
- Sur les graphes
 - plus court chemin à partir d'une source (Dijkstra)
 - plus court chemin entre tous les couples de sommets (Johnson)
 - arbre couvrant minimal (Prim)
- Interclassement de listes triées
- Code de Huffman (compression)

Arbre binomial – Définition

Pour chaque puissance de 2, il existe une structure d'arbre binomial dont la taille est cette puissance de 2.

Un arbre binomial est une structure ne contenant pas d'information.

Définition par récurrence

- B_0 est l'arbre réduit à un seul nœud,
- Étant donnés 2 arbres binomiaux B_k , on obtient B_{k+1} en faisant de l'un des B_k le premier fils à la racine de l'autre B_k .

Exemples : dessiner B_0, B_1, B_2, B_3, B_4

Arbre binomial – Propriétés

Propriétés de B_k , ($k \geq 0$)

- 1 B_k a 2^k nœuds
- 2 B_k a $(2^k - 1)$ arêtes
- 3 B_k a hauteur k
- 4 Le degré à la racine est k
- 5 Le nombre de nœuds à profondeur i est $\binom{k}{i}$
- 6 La forêt reliée à la racine de B_k est

$$< B_{k-1}, B_{k-2}, \dots, B_1, B_0 >$$

Arbre binomial – Idées de preuves

- ❶ $n_0 = 1$ et $n_k = 2n_{k-1}$
- ❷ arbre : n nœuds $\Rightarrow n - 1$ arêtes
- ❸ $h_0 = 0$ et $h_k = 1 + h_{k-1}$
- ❹ $d_0 = 0$ et $d_k = 1 + d_{k-1}$
- ❺ $n_{k,0} = 1$, $n_{k,l} = 0$ pour $l > k$, et $n_{k,i} = n_{k-1,i} + n_{k-1,i-1}$,
pour $i = 1, \dots, k$
- ❻ propriété de décomposition, par récurrence sur k

File Binomiale

Tournoi Binomial (ou tas binomial)

Un *tournoi binomial* est un arbre binomial étiqueté croissant (croissance sur tout chemin de la racine aux feuilles)

File Binomiale

Une *file binomiale* est une suite de tournois binomiaux de tailles strictement décroissantes

Exemples :

- $FB_{12} = \langle TB_3, TB_2 \rangle$,
- $FB_7 = \langle TB_2, TB_1, TB_0 \rangle$

Représentation d'une file de priorité

Une file de priorité \mathcal{P} de n éléments

- si $n = 2^k$, \mathcal{P} est un tournoi binomial
- sinon la file binomiale \mathcal{P} est une suite de tournois correspondants aux bits égaux à 1 dans la représentation binaire de n .

Représentation binaire de n

$$n = \sum_{i=0}^{\lfloor \log_2 n \rfloor} b_i \cdot 2^i, \quad \text{avec} \quad b_i \in \{0, 1\}, \quad b_{\lfloor \log_2 n \rfloor} = 1$$

Le poids de Hamming de n vaut :

$\nu(n) = \sum_i b_i$: # bits à 1 dans représentation binaire de n .

File binomiale – Propriétés

Propriétés de FB_n

- ➊ FB_n a n nœuds
- ➋ FB_n a $(n - \nu(n))$ arêtes
- ➌ Le plus grand arbre de la file est $B_{\lfloor \log_2 n \rfloor}$
(hauteur $\lfloor \log_2 n \rfloor$ et nombre de nœuds $2^{\lfloor \log_2 n \rfloor}$)
- ➍ Le nombre d'arbres de la file est $\nu(n)$
(avec $\nu(n) \leq 1 + \lfloor \log_2 n \rfloor$)
- ➎ Le minimum de la file est à la racine de l'un des arbres

- ➏ $n = \sum_i b_i \cdot 2^i,$
- ➐ $n - \nu(n) = \sum_i b_i \cdot (2^i - 1),$
- ➑ $\nu(n) = \sum_i b_i$

Union de files binomiales

1 Union de 2 tournois de tailles différentes :

$$TB_k \cup TB_{k'}, k > k' \longrightarrow F_{2^k+2^{k'}} = < TB_k, TB_{k'} >$$

Exemple : $TB_1 \cup TB_2$

2 Union de 2 tournois de même taille :

$$TB_k \cup TB'_k \longrightarrow TB_{k+1},$$

avec $rac(TB_{k+1}) = \min(rac(TB_k), (rac(TB_{k'})))$

Exemple : $TB_2 \cup TB'_2$

3 Union de 2 files binomiales \equiv addition binaire

Exemple : $FB_5 \cup FB_7$

Union de deux files

- 1 Interclasser les 2 files en partant des tournois de degré minimum
- 2 Lorsque 2 tournois de la même taille k , on engendre un tournoi de taille $k + 1$
- 3 À chaque étape au plus 3 tournois de même taille sont à fusionner (1 dans chacune des files + 1 retenue de la fusion à l'étape précédente)
- 4 Lorsque 3 tournois de la même taille k , on en retient 2 pour engendrer un tournoi de taille $k + 1$, et l'on garde le troisième comme tournoi de taille k .

Primitives sur les tournois binomiaux

```
def EstVide(T):  
    """TournoiB -> booléen  
    Renvoie vrai ssi le tournoi est vide."""
```

```
def Degre(T):  
    """TournoiB -> entier  
    Renvoie le degre de la racine du tournoi."""
```

```
def Union2Tid(T):  
    """TournoiB * TournoiB -> TournoiB  
    Renvoie l'union de 2 tournois de meme taille."""
```

```
def Decapite(T):  
    """TournoiB -> FileB  
    Renvoie la file binomiale obtenue en supprimant la racine  
    du tournoi T_k -> <T_{k-1}, T_{k-2}, ... , T_1, T_0>."""
```

```
def File(T):  
    """TournoiB -> FileB  
    Renvoie la file binomiale reduite au tournoi  
    T_k -> <T_k>."""
```

Primitives sur les files binomiales

```
def EstVide(F):  
    """FileB -> booleen  
    Renvoie vrai ssi la file est vide."""
```

```
def MinDeg(F):  
    """FileB -> TournoiB  
    Renvoie le tournoi de degre minimal dans la file."""
```

```
def Reste(F):  
    """FileB -> FileB  
    Renvoie la file privée de son tournoi de degre minimal."""
```

```
def AjoutMin(T, F):  
    """Tournoi * FileB -> FileB  
    Hypothese : le tournoi est de degre inferieur au MinDeg de la file  
    Renvoie la file obtenue en ajoutant le tournoi comme  
    tournoi de degre minimal de la file initiale."""
```

Algorithme d'Union

```
def UnionFile(F1, F2):  
    """FileB * FileB -> FileB  
    Renvoie la file binomiale union des deux files F1 et F2."""  
  
    return UFret(F1, F2, vide)
```

```
def UFret(F1, F2, T):  
    """FileB * FileB * TournoiB -> FileB  
    Renvoie la file binomiale union de deux files et d'un tournoi."""  
  
    if EstVide(T): #pas de tournoi en retenue  
        if EstVide(F1):  
            return F2  
        if EstVide(F2):  
            return F1  
  
        T1 = MinDeg(F1)  
        T2 = MinDeg(F2)  
        if Degre(T1) < Degre(T2):  
            return AjoutMin(T1, UnionFile(Reste(F1), F2))  
        if Degre(T2) < Degre(T1):  
            return AjoutMin(T2, UnionFile(Reste(F2), F1))  
        if Degre(T1) == Degre(T2):  
            return UFret(Reste(F1), Reste(F2), Union2Tid(T1, T2))  
        ...
```

Algorithme d'Union

```

...
else:    #T tournoi en retenue
    if EstVide(F1):
        return UnionFile(File(T), F2)
    if EstVide(F2):
        return UnionFile(File(T), F1)

    T1 = MinDeg(F1)
    T2 = MinDeg(F2)
    if Degre(T) < Degre(T1) and Degre(T) < Degre(T2):
        return AjoutMin(T, UnionFile(F1, F2))
    if Degre(T) == Degre(T1) and Degre(T) == Degre(T2):
        return AjoutMin(T, UFret(Reste(F1), Reste(F2), \
                                   Union2Tid(T1, T2)))
    if Degre(T) == Degre(T1) and Degre(T) < Degre(T2):
        return UFret(Reste(F1), F2, Union2Tid(T1, T))
    if Degre(T) == Degre(T2) and Degre(T) < Degre(T1):
        return UFret(Reste(F2), F1, Union2Tid(T2, T))

```

Analyse de complexité

Union de 2 files binomiales FB_n et FB_m en $O(\log_2(n + m))$

- Hypothèse :
toutes les primitives ont une complexité en $O(1)$
- Critère de complexité : *nombre de comparaisons entre clés*
- Complexité dans le *pire des cas*
- Idée :
l'union de 2 tournois de même taille nécessite 1 comparaison entre clés et ajoute 1 arête dans la file résultat.
- Conséquence : Le nombre de comparaisons pour faire l'union de 2 files c'est le nombre d'arêtes de la file union diminué du nombre d'arêtes des files de départ.

Calcul

Nombre de comparaisons pour faire l'union d'une file binomiale de n éléments et d'une file binomiale de m éléments.

$$\begin{aligned}\#cp(FB_n \cup FB_m) &= n + m - \nu(n + m) - (n - \nu(n)) - (m - \nu(m)) \\ &= \nu(n) + \nu(m) - \nu(n + m) \\ &< \lfloor \log_2 n \rfloor + 1 + \lfloor \log_2 m \rfloor + 1 \\ &\leq 2\lfloor \log_2(n + m) \rfloor + 2 \\ &\underset{n \rightarrow \infty}{=} O(\log_2(n + m)).\end{aligned}$$

Exemples :

- $FB_{21} \cup FB_{10}$
- $FB_{21} \cup FB_{11}$

Ajout d'un élément x à une file FB_n

Algorithme

Créer une file binomiale FB_1 contenant uniquement x .
Puis faire l'union de FB_1 et FB_n .

Complexité : $\nu(n) + 1 - \nu(n + 1) \longrightarrow$ entre 0 et $\nu(n)$

Exemples :

- $FB_1 \cup FB_8$
- $FB_1 \cup FB_7$

Construction

Complexité de la construction d'une file binomiale par **adjonctions successives** de ses n éléments.

$$\begin{aligned}
 \#cp(FB_n) &= \nu(n-1) + 1 - \nu(n) \\
 &\quad + \nu(n-2) + 1 - \nu(n-1) \\
 &\quad + \dots \\
 &\quad + \nu(1) + 1 - \nu(2) \\
 &= n - \nu(n).
 \end{aligned}$$

Donc le nombre moyen de comparaisons pour 1 ajout est $1 - \frac{\nu(n)}{n} < 1$.

Coût amorti d'une opération dans une série d'opérations :

$$\frac{\text{coût total}}{\#opérations}.$$

Suppression du minimum de FB_n

Recherche du minimum

Le minimum de la file est à la racine d'un des tournois la composant.

Complexité : $\nu(n) - 1$ comparaisons = $O(\log n)$

Suppression du minimum

- Déterminer l'arbre B_k de racine minimale
- Supprimer la racine de $B_k \rightarrow$ File $\langle B_{k-1}, \dots, B_0 \rangle$
- Faire l'union des files $FB_n \setminus B_k$ et $\langle B_{k-1}, \dots, B_0 \rangle$

Complexité : $O(\log n)$.

Diminuer une clé

Hypothèse : **Accès direct au nœud dont il faut diminuer la clé**

- modifier la clé
- échanger le nœud avec son père jusqu'à vérifier l'hypothèse de croissance (\equiv tas)

Le nombre maximum de comparaisons est la hauteur de l'arbre.

Complexité : $O(\log n)$.

Coût amorti

Files Binomiales :

- ajout d'un élément et recherche du minimum en $O(\log n)$
- suppression du minimum et union de 2 files en $O(\log n)$

Files de Fibonacci (la réorganisation en file binomiale est faite uniquement lors de la suppression -ajout sans contrainte-) :

- ajout d'un élément et union de 2 files en $O(1)$
- suppression du minimum en $O(\log n)$

Remarque : on ne peut pas espérer avoir $O(1)$ pour ajout et suppression du minimum, car alors on serait en contradiction avec les résultats de borne inférieure en $O(n \log n)$ pour le tri par comparaisons.

Coût amorti

Définition

- *Coût amorti* d'une opération dans une *suite d'opérations*
= coût moyen d'une opération dans le pire cas, quelle que soit la suite d'opérations.
- ne dit rien sur le coût d'une opération particulière, qui, prise indépendamment, pourrait avoir un coût pire supérieur.

Méthodes

- méthode par agrégat :
coût amorti = coût total / # d'opérations
- méthode du potentiel
- autres...

Coût amorti : méthode par agrégat

- **Principe** : majorer le coût total d'une suite de n opérations et diviser par n .
- **Exemple** : opérations sur les piles
 - `empiler(S, x)` → coût 1
 - `dépiler(S)` → coût 1
 - `multidépiler(S, k)` → coût $\leq k$

Suite de n opérations :

- coût maximal d'une opération $O(n)$
- mais coût amorti de chaque opération en $O(1)$:

(on ne dépile que les éléments empilés → coût de n opérations en $O(n)$).

Coût amorti : méthode du potentiel

- **Principe** : à chaque structure de données est associé un *potentiel*, qui peut être libéré pour payer des opérations futures
 - structure de données D_i ,
 - fonction *potentiel* $\Phi : \mathcal{D} \rightarrow \mathcal{R}^+$, vérifiant $\Phi(D_i) \geq \Phi(D_0)$
 - *coût amorti* de la i -ème opération :
 $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$ (c_i coût réel de la i -ème opération)
 - coût amorti total : $\sum_{i=1}^n \hat{c}_i = \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0)$
 Borne sup du coût réel total car $\Phi(D_n) \geq \Phi(D_0)$
- **Exemple** : opérations sur les piles
 - $\Phi(D_i)$ = nombre d'objets de D_i
 - coût amorti de chaque opération en $O(1)$:
 - empiler : $\Phi(D_i) - \Phi(D_{i-1}) = (s+1) - s$ donc $\hat{c}_i = 1 + 1 = 2$
 - depiler : $\Phi(D_i) - \Phi(D_{i-1}) = -1$ donc $\hat{c}_i = 1 - 1 = 0$
 - Multidepiler : $\Phi(D_i) - \Phi(D_{i-1}) = -\min(s, k)$ donc $\hat{c}_i = 0$

coût amorti total $\sum_{i=1}^n \hat{c}_i < 2n$,
 coût amorti d'une opération de la suite en $O(1)$.