Algorithmique

Paul Gastin

Paul.Gastin@lsv.ens-cachan.fr
http://www.lsv.ens-cachan.fr/~gastin/Algo/

L3 Informatique Cachan
2014 – 2015

Plan

1 Introduction

Preuve et terminaison

Complexité

Paradigmes

Structures de données (types abstraits)

Dictionnaires et tableaux associatifs

Files de priorité

Gestion des partitions (Union-Find)

Motivations

1. Étudier des modèles de données : types abstraits.
2. Étudier des implémentations : structures de données concrètes.
3. Étudier des algorithmes efficaces.
4. Prouver qu'un algorithme est correct.
5. Étudier la complexité des algorithmes (pire, moyenne, amortie).

Bibliographie

[1] Alfred V. Aho et Jeffrey D. Ullman.
Concepts fondamentaux de l'informatique.
Dunod, 1993.

[2] Danièle Beauquier, Jean Berstel, Philippe Chrétienne.
Éléments d'algorithmique.
Masson, 1992. Épuisé. http://www-igm.univ-mlv.fr/~berstel/

[3] Gilles Brassard, Paul Bratley.
Fundamentals of Algorithmics.
Prentice Hall, 1996.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.
Introduction à l'algorithmique.
Dunod, 2002 (seconde édition).

[5] Philippe Flajolet et Robert Sedgewick.
Introduction à l'analyse d'algorithmes.
Addison-Wesley, 1996. Thomson Publishing, 1996.

Bibliographie

[7] Dexter C. Kozen.
The design and analysis of algorithms.
Springer-Verlag, 1992.

[6] Christine Froideveaux, Marie-Claude Gaudel et Michèle Soria.
Types de données et algorithmes.
McGraw-Hill, 1990.

[8] Jon Kleinberg et Éva Tardos.
Algorithm design.
Addison-Wesley, 2005.

[9] Robert Sedgewick.
Algorithms in C or C++ or Java.
Addison-Wesley.

[10] Glynn Winskel.
The Formal Semantics of Programming Languages.
MIT Press, 1993.

Plan

Introduction

2 Preuve et terminaison

Complexité

Paradigmes

Structures de données (types abstraits)

Dictionnaires et tableaux associatifs

Files de priorité

Gestion des partitions (Union-Find)

Invariants

Exemple : Urne

Une urne contient initialement n boules noires et b boules blanches.
Tant que l'urne contient au moins 2 boules, on tire 2 boules dans l'urne et

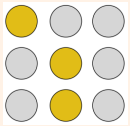
- on les jette si elles sont de la même couleur,
- on jette la noire et on remet la blanche dans l'urne sinon.

Si l'urne est vide, on remet une boule noire.

Quelle est, en fonction de n et b, la couleur de la boule dans l'urne à la fin ?

Exemples : Le revers de la médaille

Chaque médaille est dorée sur une face et argentée sur l'autre. On peut retourner simultanément les 3 médailles d'une ligne, d'une colonne ou d'une diagonale.



Peut-on faire en sorte que toutes les médailles présentent la face dorée ?
Si oui, en combien de coups au minimum ?

Invariants

Exemple : Mystère

Que fait l'algorithme suivant ?

```
Lire N
a <- 0; s <- 1; t <- 1
Tant que s ≤ N faire
   a <- a+1; s <- s+t+2; t <- t+2
Fin tant que
Afficher a
```

Preuves d'algorithmes

cf. [10, Chapters 6 and 7].

`http://www.lsv.ens-cachan.fr/~haddad/courstdagreg.pdf`

`http://www.brics.dk/~amoeller/talks/hoare.pdf`

Définition : Triplets de Hoare $\{\varphi\}\ P\ \{\psi\}$

- P est le programme.
- État s du programme P: valeur des variables, de la pile, du tas, ...
- φ est la pré-condition (hypothèse).
- ψ est la post-condition (conclusion).
- φ et ψ sont des assertions/formules faisant intervenir les variables du programme.
  Ex: Logique du 1er ordre et arithmétique sur les entiers.
- $s \models \varphi$: l'état s satisfait la formule φ.

Sémantique : $\models \{\varphi\}\ P\ \{\psi\}$
Pour tout état s tel que $s \models \varphi$, si le programme P termine en partant de s, alors l'état $P(s)$ obtenu vérifie ψ :

$$(s \models \varphi \wedge P \text{ termine sur } s) \Longrightarrow P(s) \models \psi$$

Preuves d'algorithmes

Exemples :

$\{t = 2a+1\}$ `a <- a+1; t <- t+2` $\{t = 2a+1\}$

$\{t=2a+1 \wedge s=(a+1)^2\}$ `a <- a+1; s <- s+t+2; t <- t+2` $\{t=2a+1 \wedge s=(a+1)^2\}$

$\{N \geq 0\}$
```
a <- 0; s <- 1; t <- 1
Tant que s ≤ N faire
   a <- a+1; s <- s+t+2; t <- t+2
Fin tant que
```
$\{a = \lfloor\sqrt{N}\rfloor\}$

Système de preuve de Hoare

Définition : Axiome et Règles de preuve

Arrêt
$$\frac{}{\{\psi\}\ SKIP\ \{\psi\}}$$

Affectation
$$\frac{}{\{\psi[e/x]\}\ x := e\ \{\psi\}}$$

$\psi[e/x]$: ψ dans laquelle on a substitué e aux occurrences de x.

Séquence
$$\frac{\{\psi_1\}\ P_1\ \{\psi_2\} \qquad \{\psi_2\}\ P_2\ \{\psi_3\}}{\{\psi_1\}\ P_1; P_2\ \{\psi_3\}}$$

Conditionnelle
$$\frac{\{c \wedge \varphi\}\ P_1\ \{\psi\} \qquad \{\neg c \wedge \varphi\}\ P_2\ \{\psi\}}{\{\varphi\}\ \texttt{si}\ c\ \texttt{alors}\ P_1\ \texttt{sinon}\ P_2\ \texttt{fsi}\ \{\psi\}}$$

Exemples :

$$\{s + t + 2 = (a+1)^2\}\ s := s + t + 2\ \{s = (a+1)^2\}$$

$$\{t = 2a+1 \wedge s = (a+1)^2\}\ a := a+1; s := s+t+2; t := t+2\ \{t = 2a+1 \wedge s = (a+1)^2\}$$

$$\{\texttt{True}\}\ \texttt{si}\ a < b\ \texttt{alors}\ x := a\ \texttt{sinon}\ x := b\ \texttt{fsi}\ \{x = \min(a, b)\}$$

Système de preuve de Hoare

Définition : Axiome et Règles de preuve

Itération
$$\frac{\{c \wedge I\}\ P\ \{I\}}{\{I\}\ \texttt{tant que}\ c\ \texttt{faire}\ P\ \texttt{ftq}\ \{\neg c \wedge I\}}$$

Pré-renforcement et Post-affaiblissement
$$\frac{\varphi \Rightarrow \varphi' \qquad \{\varphi'\}\ P\ \{\psi'\} \qquad \psi' \Rightarrow \psi}{\{\varphi\}\ P\ \{\psi\}}$$

Conjontion
$$\frac{\{\varphi_1\}\ P\ \{\psi_1\} \qquad \{\varphi_2\}\ P\ \{\psi_2\}}{\{\varphi_1 \wedge \varphi_2\}\ P\ \{\psi_1 \wedge \psi_2\}}$$

Exemple :

Prouver l'algorithme "Mystère"

Théorème : Le système de preuve est correct (sound)

$$\vdash \{\varphi\}\ P\ \{\psi\} \text{ implique } \models \{\varphi\}\ P\ \{\psi\}$$

Si on peut prouver $\{\varphi\}\ P\ \{\psi\}$ dans ce système de preuve ($\vdash \{\varphi\}\ P\ \{\psi\}$) alors pour tout état s vérifiant φ, si P termine sur s alors $P(s)$ vérifie ψ.

Preuves et itérations

Remarque :

- Une boucle possède de nombreux invariants.
- Certains sont indépendants de la condition : {t=2a+1}.
- D'autres nécessitent la condition : $\{a^2 \leq N\}$.

Boucle :

Pour prouver $\{\varphi\}$ `tant que` c `faire` P $\{\psi\}$
Deviner un invariant I et prouver

Initialisation : $\varphi \Rightarrow I$

Invariant : $\{I \wedge c\}\ P\ \{I\}$

Conclusion : $\neg c \wedge I \Rightarrow \psi$

Remarque : ceci ne prouve pas la terminaison.

Preuve d'algorithmes

Exemple : Correction de Partition pour le tri Rapide

```
Procédure partition (t:tableau[1..n] de réels; i,j,n:entiers; k:entier)
PRE : 1 ≤ i < j ≤ n
POST : 1 ≤ i ≤ k ≤ j ≤ n ∧ t[i..k-1] ≤ t[k] < t[k+1..j] ∧
       t[i..j] permutation de t[i..j] (où t est valeur initiale)
Début
   g <- i; d <- i



   tant que d < j faire
      si t[d] ≤ t[j] alors
         échanger(t[g],t[d])
         g <- g+1
      fsi
      d <- d+1
   ftq
   échanger(t[g],t[d])
   k <- g
Fin
```


Preuve d'algorithmes

Exemple : Recherche Dichotomique

```
fonction chercheDicho (t:tableau[1..n] de réels; n:entier; c:réel):entier
PRE : n ≥ 1 ∧ t[1..n] trié
POST : retourne 1 ≤ i ≤ n tq c = t[i] ∨ c ∉ t[1..n]
Début
   i <- 1; j <- n



   tq i < j faire
      m <- (i+j) div 2
      si c ≤ t[m] alors j <- m sinon i <- m+1 fsi
   ftq
   retourner i
Fin
```

Exercice : Exponentiation rapide

Écrire une fonction d'exponentiation rapide et prouver sa correction.

Preuve et récursivité

Exemple : Tri rapide

```
procédure triRapide (t:tableau[1..n]; i,j,n:entiers)
PRE : 1 ≤ i ∧ j ≤ n
POST : t[i..j] trié ∧ t[i..j] permutation de t[i..j]
Début
   si i < j alors

      Partition (t,i,j,n,k)


      triRapide (t,i,k-1,n)


      triRapide (t,k+1,j,n)


   fsi
Fin
```

On vérifie qu'avant chaque appel récursif l'hypothèse de triRapide est satisfaite.
On peut alors supposer la spécification vraie après chaque appel récursif.
La preuve doit être directe pour les exécutions sans appel récursif.

Plus faible pré-condition

Définition : Plus faible pré-condition (Dijkstra)

Soit P un programme et ψ une post-condition.
On note $wp(P,\psi)$ la plus faible pré-condition qui assure ψ après l'exécution de P :

$$\{wp(P,\psi)\}\ P\ \{\psi\}.$$

On note $[\![\psi]\!]$ l'ensemble des états du programme satisfaisant ψ.

$$[\![wp(P,\psi)]\!] = \{s \mid P \text{ ne s'arrête pas sur } s \text{ ou } P(s) \models \psi\}$$
$$[\![wp(P,\bot)]\!] = \{s \mid P \text{ ne s'arrête pas sur } s\ \}$$

Théorème : Complétude du système de preuve

Sous certaines hypothèses (variables entières, logique du premier ordre, ...)

- On peut calculer par induction sur le programme la plus faible pré-condition
- $\vdash \{wp(P,\psi)\}\ P\ \{\psi\}$
  le système de preuve permet de prouver la correction de la pfpc
- Le système de preuve est complet:

$$\models \{\varphi\}\ P\ \{\psi\} \text{ implique } \vdash \{\varphi\}\ P\ \{\psi\}$$

Plus faible pré-condition

Règles de calcul

Affectation	$wp(x := e, \psi) = \psi[e/x]$
Séquence	$wp(P_1; P_2, \psi) = wp(P_1, wp(P_2, \psi))$
Conditionnelle	$wp(\texttt{si } c \texttt{ alors } P_1 \texttt{ sinon } P_2 \texttt{ fsi}, \psi) = (c \wedge wp(P_1,\psi)) \vee (\neg c \wedge wp(P_2,\psi))$
Itération	possible dans certains cas mais difficile !

Exemples :

$$wp(s := s+t+2, s = (a+1)^2) \quad = \quad s+t+2 = (a+1)^2$$
$$wp(a{:=}a{+}1; s{:=}s{+}t{+}2; t{:=}t{+}2, t{=}2a{+}1 \wedge s{=}(a{+}1)^2) \quad = \quad t{=}2a{+}1 \wedge s{=}(a{+}1)^2$$
$$wp(\texttt{si } a < b \texttt{ alors } x := a \texttt{ sinon } x := b \texttt{ fsi}, x = \min(a,b)) \quad = \quad \texttt{True}$$

Relations bien fondées

Définition : Relation bien fondée

Une relation binaire $\prec$ sur un ensemble E est bien fondée s'il n'existe pas de suite infinie $(e_i)_{i\in\mathbb{N}}$ telle que $\forall i \in \mathbb{N}$, $e_{i+1} \prec e_i$ (décroissante).

Exemple : $(\mathbb{N}, <)$

Proposition : Propriétés

- Si $\prec$ est bien fondée alors $\prec$ est irréflexive.
- Si $\prec$ est bien fondée alors $\prec^+$ aussi.
- Si $\prec$ est bien fondée alors $\prec^*$ est une relation d'ordre partiel.

Proposition : Ordre lexicographique

Si $(E_1, \prec_1)$ et $(E_2, \prec_2)$ sont deux ensembles bien fondés, alors $(E_1 \times E_2, \prec)$ aussi, où

$$(e_1, e_2) \prec (f_1, f_2) \quad \text{si} \quad e_1 \prec_1 f_1 \quad \text{ou} \quad e_1 = f_1 \wedge e_2 \prec_2 f_2$$

Exemple : $(\mathbb{N}^k, <)$

Terminaison

Définition : Terminaison des boucles

Définir une fonction f à valeurs dans un ensemble muni d'une relation bien fondée, et qui dépend des variables du programme.
Montrer que la valeur de f décroît strictement à chaque exécution du corps de boucle

Exemples :

- Urne
- Mystère
- recherche dichotomique
- Partition

Terminaison

Définition : Terminaison et récursivité

Définir une fonction f à valeurs dans un ensemble muni d'une relation bien fondée, et qui dépend des paramètres de la fonction/procédure.
Montrer que la valeur de f décroît strictement à chaque appel récursif.

Exemples :

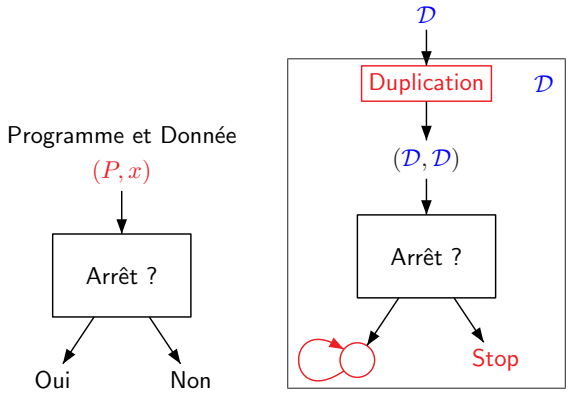
- Tri rapide
- Fonction d'Ackermann

```
fonction Ack (m,n:entiers):entier
Début
   si m = 0 alors retourner n+1
   si n = 0 alors retourner Ack(m-1,1)
   retourner Ack(m-1,Ack(m,n-1))
Fin
```


Terminaison

Théorème : Indécidabilité

Il n'existe pas de programme qui permet de tester si un programme s'arrête.



Il n'y a pas de programme pour tester l'arrêt !

Terminaison

Théorème : Indécidabilité

Il n'existe pas de programme qui permet de tester si un programme s'arrête.

- Un programme P est une suite de bits que l'on peut interpréter comme un entier noté $\overline{P}$ (en ajoutant 1 comme bit de poids fort).
- Remarque : On peut écrire une fonction `correct`(m) qui teste si un entier représente un programme syntaxiquement correct.
- On peut se restreindre aux programmes P ayant un paramètre entier.
- On cherche donc une fonction `arret`(m, n) qui retourne
  - vrai si $m = \overline{P}$ est le code d'un programme P ayant un paramètre entier et que $P(n)$ s'arrête;
  - faux sinon.

Plan

Introduction

Preuve et terminaison

3 Complexité

Paradigmes

Structures de données (types abstraits)

Dictionnaires et tableaux associatifs

Files de priorité

Gestion des partitions (Union-Find)

Quelques notations

Définition : $\mathcal{O}$, Θ et Ω

On s'intéresse au comportement asymptotique d'une fonction $g : \mathbb{R} \to \mathbb{R}$ au voisinage de $+\infty$.

$$\mathcal{O}(g) = \{f : \mathbb{R} \to \mathbb{R} \mid \exists \beta > 0, \exists x_0 > 0, \forall x \geq x_0, |f(x)| \leq \beta |g(x)|\}$$

$$\begin{aligned}\Omega(g) &= \{f : \mathbb{R} \to \mathbb{R} \mid g \in \mathcal{O}(f)\} \\ &= \{f : \mathbb{R} \to \mathbb{R} \mid \exists \alpha > 0, \exists x_0 > 0, \forall x \geq x_0, \alpha |g(x)| \leq |f(x)|\}\end{aligned}$$

$$\begin{aligned}\Theta(g) &= \mathcal{O}(g) \cap \Omega(g) \\ &= \{f : \mathbb{R} \to \mathbb{R} \mid \exists \alpha, \beta > 0, \exists x_0 > 0, \forall x \geq x_0, \alpha |g(x)| \leq |f(x)| \leq \beta |g(x)|\}\end{aligned}$$

Les mêmes définitions s'appliquent aux fonctions $g : \mathbb{N} \to \mathbb{N}$ ou $g : \mathbb{N} \to \mathbb{R}$.

Complexité au pire

Définition : Complexité au pire

Soit A un algorithme et x une donnée de A.
On note $c(x)$ le coût (en temps, en mémoire, ...) de l'exécution de A sur x.
Pour $n \in \mathbb{N}$, on note D_n l'ensemble des données de taille n.

La complexité au pire de A est définie pour $n \in \mathbb{N}$ par

$$C_{\text{pire}}(n) = \max_{x \in D_n} c(x).$$

Parfois on définit la complexité sur les données de taille au plus n:

$$C_{\text{pire}}(n) = \max_{m \leq n} \max_{x \in D_m} c(x).$$

Complexité au pire

Exemple : Fusion pour le tri fusion

```
procédure Fusion (t:tableau[1..m]; i,j,k:entiers)
PRE : 1 ≤ i ≤ k < j ≤ m et t[i..k], t[k+1..j] triés
POST : t[i..j] trié ∧ t[i..j] permutation de t[i..j]
Début
   s:tableau[i..j]
   a <- i; b <- k+1; c <- i
   tq a ≤ k et b ≤ j faire
      si t[a] ≤ t[b] alors s[c] <- t[a]; a++; c++
                     sinon s[c] <- t[b]; b++; c++
      finsi
   ftq
   tq a ≤ k faire s[c] <- t[a]; a++; c++ ftq
   tq b ≤ j faire s[c] <- t[b]; b++; c++ ftq
   t[i..j] <- s[i..j]
Fin
```

Complexité au pire

Exercice : Prouver la correction de la procédure Fusion

Complexité au pire

Exemple : Complexité au pire du tri fusion

```
procédure tri-fusion (t:tableau[1..m] de réels; i,j:entiers)
PRE : 1 ≤ i ≤ j ≤ m
POST : t[i..j] trié ∧ t[i..j] permutation de t[i..j]
Début
   si i < j alors
      k <- (i+j) div 2
      tri-fusion (t,i,k)
      tri-fusion (t,k+1,j)
      fusion (t,i,j,k)
   finsi
Fin
```

On suppose la complexité au pire de `fusion` en $\Theta(j - i + 1)$.
Complexité au pire du tri fusion satisfait la récurrence de partition ($n = j - i + 1$) :

$$\begin{aligned} c(1) &= 1 \\ c(n) &= n + c(\lceil n/2 \rceil) + c(\lfloor n/2 \rfloor) \qquad \text{si } n > 1 \end{aligned}$$

Récurrences de partitions

Théorème : Récurrences de partitions (cf. [2, chap. 2.2.3])

Soit $t : \mathbb{N} \to \mathbb{R}_+$ une fonction croissante à partir d'un certain rang et telle que $\exists n_0 > 0, \exists b > 1, \exists k \geq 0, \exists a, c, d > 0$ vérifiant

$$t(n_0) = d$$
$$t(n) = at(n/b) + cn^k \qquad \text{si } n > n_0 \text{ et } n/n_0 \text{ est une puissance de } b$$

alors

$$t(n) = \begin{cases} \Theta(n^k) & \text{si } a < b^k \\ \Theta(n^k \log_b(n)) & \text{si } a = b^k \\ \Theta(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

Exemple : Tri fusion

$n_0 = 1$, $c = d = k = 1$, $a = b = 2$.

La complexité au pire du tri fusion est en $\Theta(n \log n)$.

Récurrences de partitions

Exercice :

Soit $t : \mathbb{N} \to \mathbb{R}_+$ une fonction croissante à partir d'un certain rang et telle que

$$t(1) = d$$
$$t(n) = 2t(n/2) + \log_2(n) \qquad \text{si } n > 1 \text{ est une puissance de } 2$$

Montrer que $t(n) = \Theta(n)$.

Généraliser à $t(n) = at(n/b) + cn^k(\log_b n)^q$.

Exercice :

Montrer que la fonction t définie par

$$t(1) = 0$$
$$t(n) = n + t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) \qquad \text{si } n > 2$$

est

$$t(n) = n\lfloor \log n \rfloor + 2n - 2^{1+\lfloor \log n \rfloor}$$

Complexité au pire

Exercice : Récursivité en temps linéaire

Soit $t : \mathbb{N} \to \mathbb{R}_+$ une fonction.
Supposons qu'il existe un entier $k \geq 1$, des fonctions $f_1, \ldots, f_k : \mathbb{N} \to \mathbb{N}$, un entier n_0, et deux réels $0 < \alpha < 1$ et $\beta > 0$ tels que $\forall n > n_0$,

$$f_1(n) + \cdots + f_k(n) \leq \alpha n$$

et

$$t(n) \leq t(f_1(n)) + \cdots + t(f_k(n)) + \beta n$$

Montrer que $t(n) = \mathcal{O}(n)$.

Exercice : Sélection

Écrire une fonction de sélection dont la complexité au pire est linaire.

```
fonction Sélection (t:tableau[1..n] d'éléments; n,m:entiers):  élément
PRE : 1 ≤ m ≤ n
POST : retourne l'élément de rang m du tableau t
```


Coût amorti

Définition : Coût amorti

On considère une suite d'opérations $o_1, \ldots, o_m$ agissant sur des données

$$d_0 \xrightarrow{o_1} d_1 \xrightarrow{o_2} d_2 \ldots \xrightarrow{o_m} d_m$$

Le coût amorti de cette suite d'opérations est $\frac{1}{m}\sum_{i=1}^{m} c(o_i)$

Exemple : opérations de pile

dep-emp$_k(x)$: dépiler k éléments (si possible) puis empiler l'élément x.

Le coût de dep-emp$_k(x)$ est $1 + \min(k, \text{taille(pile)})$.

Le coût amorti d'une suite de telles opérations à partir d'une pile vide est 2.

Remarque : le potentiel d'une pile est sa hauteur.

Coût amorti

Définition : Méthode du potentiel

On définit une fonction potentiel sur l'ensemble des données : $h : D \to \mathbb{R}_+$.
Le coût amorti par potentiel de l'opération $d_{i-1} \xrightarrow{o_i} d_i$ est :

$$a(o_i) = c(o_i) + h(d_i) - h(d_{i-1})$$

Le coût amorti par potentiel d'une suite d'opérations est $\frac{1}{m}\sum_{i=1}^{m} a(o_i)$.

Lemme : Coût amorti et potentiel

Si $h(d_0) \leq h(d_m)$ alors $\frac{1}{m}\sum_{i=1}^{m} c(o_i) \leq \frac{1}{m}\sum_{i=1}^{m} a(o_i)$.

Exemple : Incrémentation

On considère l'opération d'incrémentation d'un entier écrit en binaire.
Le coût de $i{+}{+}$ est $1+k$ si $\overline{i}^2$ (l'écriture binaire de i) se termine par 01^k.
Le coût amorti d'une suite de n incrémentations à partir de 0 est 2.

Implémentation d'une pile

Exemple : Implémentation statique d'une pile

```
N = 100
pile :  tableau[1..N] de réels
n <- 0

empiler(x) :  si n < N alors n++; pile[n] <- x fsi
dépiler() :  si n > 0 alors n--; retourner pile[n+1] fsi
```

Exemple : Implémentation dynamique d'une pile

empiler(x) lorsque le tableau est plein :
on crée un nouveau tableau de taille double,
on copie la pile dans le nouveau tableau avant d'empiler le nouvel élément.
Initialement, on part d'un tableau de taille $N = 2$.

Proposition : Coût amorti

Le coût amorti des opérations avec l'implémentation dynamique est constant.

Complexité en moyenne

Définition : Complexité en moyenne

Soit A un algorithme.
Pour $n \in \mathbb{N}$, on note D_n l'ensemble des données de A de taille n.
On suppose D_n muni d'une distribution de probabilité $p : D_n \to [0,1]$:

$$1 = \sum_{x \in D_n} p(x)$$

Le coût $c_n : D_n \to \mathbb{R}_+$ de l'exécution de A sur une donnée est une variable aléatoire.

La complexité en moyenne de A est l'espérance de cette variable aléatoire :

$$C_{\text{moy}}(n) = \mathbf{E}(c_n) = \sum_{x \in D_n} p(x) \cdot c_n(x).$$

Dans le cas d'une distribution uniforme, on a :

$$C_{\text{moy}}(n) = \frac{1}{|D_n|} \sum_{x \in D_n} c_n(x).$$

Borne inférieure de complexité

Proposition : Profondeur moyenne des feuilles

La profondeur moyenne des feuilles d'un arbre binaire ayant n feuilles est au moins $\log_2 n$.

Proposition : Comparaisons

Le nombre moyen de comparaisons de clés pour les tris par comparaisons est en $\Omega(n \log n)$.

Corollaire : Borne inférieure pour les tris par comparaison

La complexité en moyenne pour les tris par comparaisons de clés est en $\Omega(n \log n)$.
La complexité au pire pour les tris par comparaisons de clés est en $\Omega(n \log n)$.

Complexité en moyenne du Tri rapide

On considère que le tableau contient les éléments $1, 2, \ldots, n$.

C'est donc une permutation $\sigma \in \mathfrak{S}_n$.

Univers : $\mathfrak{S}_n$ avec probabilité uniforme $p\colon \mathfrak{S}_n \to [0,1]$.

Théorème :

Soit $X_n : \mathfrak{S}_n \to \mathbb{N}$ la v.a. donnant le nombre de comparaisons du tri rapide.

$$E(X_n) \leq 2n\ln(n)$$

On suppose que la procédure `partition(t, l, r, k)` procède uniquement par comparaisons avec le pivot et échanges.

Pour $\sigma \in \mathfrak{S}_n$, on note σ^g la permutation "gauche" obtenue après partition.

On suppose que le pivot est $\sigma(n) = k$. On a alors $\sigma^g \in \mathfrak{S}_{k-1}$.

Lemme : L'uniformité est préservée par la partition

Soit $\rho \in \mathfrak{S}_{k-1}$, alors $\quad |\{\sigma \in \mathfrak{S}_n \mid \text{ pivot} = k \wedge \sigma^g = \rho\}| = \frac{(n-1)!}{(k-1)!}$

et donc $\quad p(\sigma^g = \rho \mid \text{pivot} = k) = \frac{1}{(k-1)!}$

Complexité en moyenne du Tri rapide

Pour $1 \leq i < j \leq n$, on définit la variable aléatoire $X_n^{i,j} : \mathfrak{S}_n \to \mathbb{N}$ par $X_n^{i,j}(\sigma) =$ nombre de fois que les valeurs i et j sont comparées durant le tri.

Lemme : Espérance des comparaisons

- Si $1 \leq i < j < k \leq n$ alors $E(X_n^{i,j} \mid \text{pivot} = k) = E(X_{k-1}^{i,j})$

La fonction partition compare chaque clé exactement une fois avec le pivot.

- $E(X_n^{i,j}) = \dfrac{2}{j-i+1}$

Soit $X_n : \mathfrak{S}_n \to \mathbb{N}$ la v.a. donnant le nombre de comparaisons du tri rapide.

Théorème : $E(X_n) \leq 2n\ln(n)$

Complexité en moyenne

Exercice : Tri rapide

On définit les variables aléatoires $X_n^p, X_n^g, X_n^d : \mathfrak{S}_n \to \mathbb{N}$ donnant les complexités respectives de la partition, de l'appel récursif "gauche" et de l'appel récursif "droit". Montrer que

- $X_n = X_n^p + X_n^g + X_n^d$
- $E(X_n^g) = \dfrac{1}{n}\displaystyle\sum_{k=1}^{n} E(X_{k-1})$

La fonction partition fait $n-1$ comparaisons sur un tableau de taille n.

- $E(X_n) = n - 1 + \dfrac{2}{n}\displaystyle\sum_{k=1}^{n} E(X_{k-1})$

En déduire que $E(X_n) \in \mathcal{O}(n\ln(n))$.

Plan

Introduction

Preuve et terminaison

Complexité

4 Paradigmes
- Algorithmes gloutons
- Programmation dynamique
- Diviser pour régner

Structures de données (types abstraits)

Dictionnaires et tableaux associatifs

Files de priorité

Gestion des partitions (Union-Find)

Bibliographie

[3] Gilles Brassard, Paul Bratley.
Fundamentals of Algorithmics.
Prentice Hall, 1996.

[8] Jon Kleinberg et Éva Tardos.
Algorithm design.
Addison-Wesley, 2005.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.
Introduction à l'algorithmique.
Dunod, 2002 (seconde édition).

Problèmes d'optimisation

Définition : Problème d'optimisation

- Un problème peut avoir plusieurs solutions
- Un critère d'évaluation permet de comparer les solutions
- On cherche une solution optimale.

Exemple : Problème du sac à dos

On dispose de n objets de poids et de valeurs fixés et d'un sac à dos avec une charge maximale. On veut maximiser la valeur des objets transportés.

- Données : les poids $w_1,\ldots,w_n$ et valeurs $v_1,\ldots,v_n$ des n objets et la charge maximale W du sac à dos.
- Solution : $x_1,\ldots,x_n$ avec $x_i \in \{0,1\}$ tels que $x_1w_1 + \cdots + x_nw_n \leq W$.
- Critère d'évaluation : maximiser $x_1v_1 + \cdots + x_nv_n$.

Variante : possibilité de fractionner les objets, i.e., $x_i \in [0,1]$.

Algorithmes gloutons

Définition : Algorithme glouton

- Cadre : Une solution est composée d'une suite de choix.
  Le premier choix fait, on est ramené à un sous-problème de même nature.
- Glouton : à chaque étape, on fait le choix qui semble optimal (optimum local) et on ne remet pas ce choix en question.
- Solution globale : premier choix + solution du sous-problème correspondant.

Problème du sac à dos : Choix glouton

$W = 100$

- objet le plus léger
- objet de plus grande valeur
- objet de rapport $\frac{\text{valeur}}{\text{poids}}$ maximal

Objets	1	2	3	4	5	6
Poids	100	50	45	20	10	5
valeur	40	35	18	4	10	2
$\frac{\text{valeur}}{\text{poids}}$	0,4	0,7	0,4	0,2	1	0,4

Proposition : Sac à dos

Si les objets sont fractionnables, le choix glouton basé sur le rapport $\frac{\text{valeur}}{\text{poids}}$ conduit à une solution optimale.

Algorithmes gloutons

Quand est-ce que ça marche ?

On suppose que l'on sait déterminer efficacement un choix localement optimal (glouton) sans regarder les sous-problèmes générés.

- Propriété du choix glouton :
  Il existe une solution globalement optimale dont le premier choix est localement optimal.
- Propriété des sous-structures optimales :
  Si un problème A contient un sous-problème B, toute solution optimale au problème A contient une solution optimale au sous-problème B.
  Contre-exemple : chemin simple de longueur maximale dans un graphe.
- Combinaison : Solution optimale au problème =
  choix glouton + solution optimale au sous-problème associé

Difficultés des algorithmes glouton

- Trouver un bon choix glouton.
- Prouver que la solution obtenue est optimale.

Sélection de tâches

Exemple : Sélection de tâches

On dispose de n tâches à éxecuter. Chaque tâche doit s'éxecuter durant un intervalle de temps donné. On dispose d'une seule ressource. On veut choisir un sous-ensemble maximal de tâches à exécuter.

- Données : les intervalles $I_1 = [a_1, b_1], \ldots, I_n = [a_n, b_n]$ associés aux tâches.
- Solution : $i_1, \ldots, i_k$ tels que les intervalles $I_{i_1}, \ldots, I_{i_k}$ sont 2 à 2 disjoints.
- Critère d'évaluation : maximiser k.

Choix gloutons

- Choisir la tâche qui commence le plus tôt (utiliser la ressource au plus tôt).
- Choisir la tâche dont la durée d'exécution est la plus courte (ne pas monopoliser longtemps la ressource).
- Choisir la tâche ayant le moins de conflits.
- Choisir la tâche qui termine le plus tôt (libérer la ressource au plus tôt).

Sélection de tâches

Algorithme glouton

Trier les tâches selon les dates de fin: on suppose donc $b_1 \leq \cdots \leq b_n$.

$X \leftarrow \emptyset$; $t \leftarrow -1$
`Pour i de 1 à n faire`
`  si` $t < a_i$ `alors` $X \leftarrow X \cup \{i\}$; $t \leftarrow b_i$ `fsi`
`fpour`

Problèmes d'optimisation

Exercice : Sélection de tâches

On dispose de n tâches à éxecuter. Chaque tâche doit s'éxecuter durant un intervalle de temps donné. On dispose de plusieurs ressources. On veut affecter les tâches aux différentes ressources.

- Données : les intervalles $I_1 = [a_1, b_1], \ldots, I_n = [a_n, b_n]$ associés aux tâches.
- Solution : une affectation $\sigma : \{1, \ldots, n\} \to \{1, \ldots, k\}$ telle que pour toute ressource $r \in \{1, \ldots, k\}$, les intervalles des tâches de $\sigma^{-1}(r)$ sont 2 à 2 disjoints.
- Critère d'évaluation : minimiser k.

Matroïdes et algorithmes gloutons

Définition : Matroïde (Whithey, 1935. Indépendance linéaire)

Un matroïde est un couple (S, I) où S est un ensemble fini non vide et I est une famille non vide de parties de S vérifiant :

- hérédité : si $X \in I$ et $Y \subseteq X$ alors $Y \in I$,
- échange : si $X, Y \in I$ et $|X| < |Y|$ alors $\exists y \in Y \setminus X$ tel que $X \cup \{y\} \in I$.

Une partie $X \in I$ est dite indépendante.
Remarque : toutes les parties indépendantes maximales ont la même taille.

Exemple : Colonnes indépendantes d'une matrice de $\mathbb{R}^{n \times m}$

Exemple : Matroïde de graphe

Soit $G = (V, E)$ un graphe non dirigé.

Soit $M_G = (E, I)$ avec $I = \{A \subseteq E \mid (V, A) \text{ est acyclique}\}$.

M_G est un matroïde.

Matroïdes et algorithmes gloutons

Définition : Matroïde valué

Un matroïde valué est un triplet $M = (S, I, w)$ où (S, I) est un matroïde muni d'une fonction de poids $w : S \to]0, +\infty[$.

La fonction poids est étendue aux parties de S additivement: pour $X \subseteq S$,

$$w(X) = \sum_{x \in X} w(x)$$

Problème

Trouver une partie indépendante de poids maximal.
Remarque : une partie indépendante de poids maximal est une partie indépendante maximale au sens de l'inclusion.

Matroïdes et algorithmes gloutons

Définition : Choix glouton et algorithme

Ajouter l'élément restant de poids maximal qui ne viole pas l'indépendance.
On suppose $S = \{x_1, \ldots, x_n\}$ avec $w(x_1) \geq w(x_2) \geq \cdots \geq w(x_n)$.

```
X <- ∅; i <- 1
Tant que i ≤ n faire
   Invariant : ∃Y ∈ I de poids maximal telle que X ⊆ Y ⊆ X ∪ {x_i, ..., x_n}
   si X ∪ {x_i} ∈ I alors X <- X ∪ {x_i} finsi
   i <- i + 1
ftq
Retourner X
```

Théorème : Optimalité

L'algorithme glouton est correct, i.e., à la fin, X est une partie indépendante de poids maximal.

Exercice : Problème 3 du partiel 2011

Algorithmes gloutons

Caractéristiques

- Approche descendante (récursive ou itérative)
- Un choix n'est jamais remis en question (pas de backtracking)
- Un choix engendre un seul sous-problème.
- L'optimalité globale n'est pas garantie par les choix gloutons.
- Il faut prouver l'optimalité de la solution obtenue.

Exemples d'applications

- Ordonnancement de tâches.
- Codage de Huffman pour la compression
- Arbres couvrants minimaux : Kruskal ou Prim
- Plus courts chemins : Dijkstra
- ...

Problème du sac à dos

Exemple : Sac à dos

- On suppose que les objets ne sont pas fractionnables et les poids sont entiers.
  - Données : les poids $w_1, \ldots, w_n$ et valeurs $v_1, \ldots, v_n$ des n objets et la charge maximale W du sac à dos.
  - Solution : $x_1, \ldots, x_n$ avec $x_i \in \{0, 1\}$ tels que $x_1 w_1 + \cdots + x_n w_n \leq W$.
  - Critère d'évaluation : maximiser $x_1 v_1 + \cdots + x_n v_n$.
- Sous-problème : $V(i, j)$ valeur maximale si on n'utilise que les objets 1 à i et que la capacité du sac à dos est j.
- Solution récursive (sous-structure optimale) : $V(0, j) = V(i, 0) = 0$ et

$$V(i, j) = \begin{cases} V(i-1, j) & \text{si } j < w_i \\ \max\{V(i-1, j),\ v_i + V(i-1, j-w_i)\} & \text{sinon.} \end{cases}$$

- On en déduit facilement une calcul itératif de V ou une solution récursive avec mémoïsation. $\mathcal{O}(nW)$
- On construit à partir de V une solution optimale. $\mathcal{O}(n)$

Problème du sac à dos

Exemple : Sac à dos

- Solution itérative ascendante :

```
Variables globales :
v[1..n] :  valeurs des objets
w[1..n] :  poids des objets
V[0..n,0..W] : valeurs optimales

pour j <- 0 à W faire V[0,j] <- 0 fpour
pour i <- 1 à n faire
   pour j <- 0 à W faire
      si w[i] ≤ j alors
           V[i,j] <- max(V[i-1,j],v[i]+V[i-1,j-w[i]])
      sinon
           V[i,j] <- V[i-1,j]
      fsi
   fpour
fpour
```

- Complexité $\mathcal{O}(nW)$
- On peut retrouver la solution optimale à partir du tableau `V`. $\mathcal{O}(n)$

Mémoïsation

Définition : Mémoïsation

- Mémorisation des résultats antérieurs.
- Nécessaire à l'efficacité d'un algorithme récursif (top-down).

Exemple : Sac à dos

- Solution récursive descendante :

```
Variables globales :
v[1..n] :  valeurs des objets
w[1..n] :  poids des objets
V[0..n,0..W] : résultats déjà calculés, initialisés à ∞

fonction g(i,j) :  entier
   si V[i,j] < ∞ alors retourner V[i,j] fsi
   si i = 0 ou j = 0 alors V[i,j] <- 0; retourner V[i,j] fsi
   V[i,j] <- g[i-1,j]
   si j ≥ W[i] alors V[i,j] <- max(V[i,j], v[i] + g[i-1,j-w[i]]) fsi
   retourner V[i,j]
```

- Complexité $\mathcal{O}(nW)$
- On peut retrouver la solution optimale à partir du tableau `V`. $\mathcal{O}(n)$

Programmation dynamique

Principes

- Chaque choix engendre un sous-problème.
  Propriété de sous-structure optimale : Une solution optimale qui utilise ce choix contient une solution optimale au sous-problème associé.
  Il faut résoudre tous les sous-problèmes pour déterminer la solution optimale.
  Les sous-problèmes ne sont pas indépendants.
- Définir récursivement les valeurs optimales.
  Calculer les valeurs optimales (itératif ou récursif avec Mémoïsation).
  Il faut éviter de faire certains calculs plusieurs fois.
  Construire la solution optimale en utilisant les valeurs calculées.

Exemple : Sac à dos

- Problème : $V(i,j)$ valeur maximale si on n'utilise que les objets 1 à i et que la capacité du sac à dos est j.
  Choix : prendre ou non l'objet i.
  Sous-problèmes associés : $V(i-1, j-w_i)$ ou $V(i-1,j)$.
- Solution récursive (sous-structure optimale) : $V(0,j) = V(i,0) = 0$ et

$$V(i,j) = \begin{cases} V(i-1,j) & \text{si } j < w_i \\ \max\{V(i-1,j),\ v_i + V(i-1, j-w_i)\} & \text{sinon.} \end{cases}$$

Programmation dynamique

Exemple : Sélection de tâches pondérées

On dispose de n tâches à éxecuter. Chaque tâche doit s'éxecuter durant un intervalle de temps donné et possède une valeur. On dispose d'une seule ressource. On veut maximiser la somme des valeurs des tâches exécutées.

- Données : les intervalles de temps $I_1 = [a_1, b_1], \ldots, I_n = [a_n, b_n]$ et les valeurs $v_1, \ldots, v_n$ associés aux tâches.
- Solution : $i_1, \ldots, i_k$ tels que les intervalles $I_{i_1}, \ldots, I_{i_k}$ sont 2 à 2 disjoints.
- Critère d'évaluation : maximiser $v_{i_1} + \cdots + v_{i_k}$.

Remarques

Remarque : Dynamique versus glouton

- Dynamique :
  - $+$: La solution obtenue est optimale.
  - $-$: Le choix dépend des solutions aux sous-problèmes.
  - $-$: Il faut résoudre tous les sous-problèmes.
  - $-$: La complexité dépend du nombre de sous-problèmes.
- Glouton :
  - $+$: Le choix ne dépend pas des solutions aux sous-problèmes.
  - $+$: Chaque choix n'engendre qu'un sous-problème à résoudre.
  - $+$: La complexité est bonne en général.
  - $-$: La solution obtenue n'est pas forcément optimale.

La propriété de sous-structure optimale est nécessaire aux deux techniques.

Suite de multiplications de matrices

Définition : Suite de multiplications de matrices

- On considère une suite $M_1, M_2, \ldots, M_n$ de matrices rectangulaires de dimensions $d_0 \times d_1, d_1 \times d_2, \ldots, d_{n-1} \times d_n$.
- On veut calculer $M_1 \cdot M_2 \cdots M_n$ en minimisant le nombre de multiplications.
- On suppose que le produit d'une matrice $p \times q$ par une matrice $q \times r$ se fait avec pqr multiplications scalaires.

Exemple :

Avec la suite de dimensions $13 \times 5, 5 \times 89, 89 \times 3, 3 \times 34$:

Parenthésage	Multiplications
$((M_1 \cdot M_2) \cdot M_3) \cdot M_4$	10582
$(M_1 \cdot M_2) \cdot (M_3 \cdot M_4)$	54201
$(M_1 \cdot (M_2 \cdot M_3)) \cdot M_4$	2856
$M_1 \cdot ((M_2 \cdot M_3) \cdot M_4)$	4055
$M_1 \cdot (M_2 \cdot (M_3 \cdot M_4))$	26418

Exercice :

Proposer un choix glouton. Conduit-il à une solution optimale ?

Suite de multiplications de matrices

Méthode naïve

- On calcule le nombre de multiplications pour chaque parenthésage.
- Nombre de parenthésages P_n pour un produit de n matrices : $P_1 = 1$ et si $n > 1$ alors

$$P_n = \sum_{i=1}^{n-1} P_i \cdot P_{n-i}$$

Exercice : Nombres de Catalan

Montrer que P_{n+1} est le n-ième nombre de Catalan :

$$C_n = \frac{1}{n+1}\binom{2n}{n} \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$$

Complexité de la méthode naïve

Exponentielle : $n \cdot P_n$

Programmation dynamique

Exemple : Suite de multiplications de matrices

- Sous-problème : $m(i,j)$ nombre optimal de multiplications pour calculer le produit $M_i \cdot M_{i+1} \cdots M_j$.
- Définition récursive : $m(i,i) = 0$ et si $i < j$ alors

$$m(i,j) = \min_{i \le k < j} \Big(m(i,k) + m(k+1,j) + d_{i-1} d_k d_j \Big)$$

- Solution itérative ascendante : Calculs dans l'ordre $s = j - i$ croissant.
- Complexité :

$$\sum_{s=1}^{n-1} (n-s) \cdot s = \frac{n^3 - n}{6}$$

- Reconstruire la solution : pour accélérer, on peut mémoriser dans un tableau l'indice k qui réalise le minimum ci-dessus.

Programmation dynamique

Exemple : Suite de multiplications de matrices

- Solution récursive déscendante avec mémoïsation :

```
Variables globales :
d[0..n] :  suite des dimensions
m[1..n,1..n] :  résultats déjà calculés, initialisés à ∞
                sauf diagonale initialisée à 0

fonction calc(i,j) :  entier
   si m[i,j] < ∞ alors retourner m[i,j] fsi
   pour k <- i à j - 1 faire
      x <- d[i-1]·d[k]·d[j]
      si x < m[i,j] alors
         m[i,j] <- min(m[i,j], x + calc(i,k) + calc(k+1,j)
      fsi
   fpour
   retourner m[i,j]
```

- Complexité : au pire comme la version itérative, souvent mieux.
- Initialisation : $\frac{n(n+1)}{2}$
- On peut utiliser la technique d'initialisation virtuelle.

Initialisation virtuelle

Initialisation virtuelle

- $t[1..n]$ le tableau à initialiser virtuellement.
- On utilise 2 tableaux auxiliaires a et b de même dimension, et un compteur c.
- Initialement, $c = 0$: c est le nombre d'éléments initialisés dans t.
- Pour $1 \leq i \leq c$, $a[i] = k$ et $b[k] = i$ si l'élément k de t a été initialisé le i-ème.
- $t[k]$ initialisé $\iff$ $(1 \leq b[k] \leq c$ et $a[b[k]] = k)$.

Diviser pour régner

Définition : Algorithme générique

```
fonction DR(x)
   si x est petit alors retourner adhoc(x) fsi
   Décomposer x en instances plus petites x_1,...,x_k
   pour i <- i à k faire y_i <- DR(x_i)
   Combiner y_1,...,y_k pour obtenir la solution y au problème x
   retourner y
```

Exemple :

- Recherche dichotomique
- Tri fusion
- Tri rapide

Diviser pour régner

Exemple : Multiplication rapide de matrices (Strassen 1969)

Données : 2 matrices carrées A et B de dimention n.
Problème : calculer $C = A \times B$.

Algorithme naïf : $\mathcal{O}(n^3)$ multiplications de scalaires.

Diviser pour régner :
On peut calculer AB avec 7 multiplications de matrices de dimension moitié.

$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) \cdot \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right) = \left(\begin{array}{c|c} D_2 + D_3 & D_1 + D_2 + D_5 + D_6 \\ \hline D_1 + D_2 + D_4 - D_7 & D_1 + D_2 + D_4 + D_5 \end{array}\right)$$

$$D_1 = (A_{21} + A_{22} - A_{11})(B_{22} - B_{12} + B_{11}) \qquad D_4 = (A_{11} - A_{21})(B_{22} - B_{12})$$
$$D_2 = A_{11}B_{11} \qquad D_5 = (A_{21} + A_{22})(B_{12} - B_{11})$$
$$D_3 = A_{12}B_{21} \qquad D_6 = (A_{12} - A_{21} + A_{11} - A_{22})B_{22}$$
$$D_7 = A_{22}(B_{11} + B_{22} - B_{12} - B_{21})$$

Récurrence de partition : $t(n) = 7t(n/2) + \Theta(n^2)$.

On en déduit un algorithme en $\mathcal{O}(n^{\log_2 7})$. $\log_2 7 \approx 2,807$.

Coppersmith and Winograd 1986 : $\mathcal{O}(n^{2,376})$.

Diviser pour régner

Exercice : Médian d'un tableau

Soit T un tableau contenant n éléments.
L'élément de rang k de T est celui qui se trouverait en position k si on triait le tableau.
Montrer que l'on peut calculer l'élément de rang k en temps $\mathcal{O}(n)$.

Indication : utiliser la fonction partition du tri rapide.

Convolution

Exemple : Convolution

- Données : 2 suites de réels $a = (a_0, a_1, \ldots, a_{n-1})$ et $b = (b_0, b_1, \ldots, b_{p-1})$
- Problème : calculer la suite $c = (c_0, c_1, \ldots, c_{n+p-2})$ où

$$c_k = \sum_{i+j=k} a_i b_j$$

Méthode naïve $\mathcal{O}(np)$ opérations (sommes et produits).

Application : lissage en traitement du signal.
La suite $a = (a_0, a_1, \ldots, a_{n-1})$ est une suite de mesures (un échantillonnage).
Le lissage Gaussien remplace chaque mesure a_i par

$$a'_i = \sum_{-k \leq s \leq k} w_s a_{i+s} = \frac{1}{Z} \sum_{-k \leq s \leq k} e^{-s^2} a_{i+s}$$

Il s'agit de convolutions avec la suite de poids (vecteur unitaire)

$$w = \frac{1}{Z}(e^{-k^2}, e^{-(k-1)^2}, \ldots, e^{-1}, 1, e^{-1}, \ldots, e^{-(k-1)^2}, e^{-k^2})$$

Produit de polynômes

Exemple : Produit de polynômes

- Données : 2 polynômes $A = a_0 + a_1 X + \cdots + a_{n-1} X^{n-1}$ et $B = b_0 + b_1 X + \cdots + b_{p-1} X^{p-1}$
- Problème : calculer le polynôme $C = AB$.

Il s'agit d'une convolution : $C = c_0 + c_1 X + \cdots + c_{n+p-2} X^{n+p-2}$ avec

$$c_k = \textstyle\sum_{i+j=k} a_i b_j$$

Méthode naïve $\mathcal{O}(np)$ opérations (sommes et produits).

Diviser pour régner : Anatolii Alexeevitch Karatsuba 1960

On suppose $n = p = 2k$. On écrit $A = A_0 + X^k A_1$ et $B = B_0 + X^k B_1$.
On calcule AB avec 3 produits de polynômes de degré $k-1$ (et $\Theta(n)$ additions).
On obtient un algorithme en $\Theta(n^{\log_2 3})$ opérations $\log_2 3 \approx 1,585$.

`http://www.cs.pitt.edu/~kirk/cs1501/animations/Karatsuba.html`

Produit de polynômes et FFT

Produit de polynômes et interpolation $\mathcal{O}(n \log n)$

On suppose encore $n = p$. On choisit $2n$ valeurs $x_0, \ldots, x_{2n-1}$.

1. Calculer $A(x_i)$ et $B(x_i)$ pour $0 \leq i \leq 2n-1$.
2. Calculer $C(x_i) = A(x_i)B(x_i)$ pour $0 \leq i \leq 2n-1$.
3. Retrouver les coefficients de C à partir des valeurs calculées (interpolation).

Pb : la méthode naïve utilise $\mathcal{O}(n^2)$ opérations pour les évaluations.
Pour les étapes 1 et 3 on utilise la transformée de Fourier rapide.

Transformée de fourier discrète (DFT)

- Données : $A = a_0 + a_1 X + \cdots + a_{n-1} X^{n-1}$ polynôme complexe ($a_i \in \mathbb{C}$)
- Problème : calculer $A(\omega_n^s)$ pour $0 \leq s \leq n-1$ où $\omega_n = e^{i\frac{2\pi}{n}}$ (une racine primitive n-ième de l'unité).

Théorème : Transformée de Fourier rapide (FFT)

On peut calculer la DFT en temps $\mathcal{O}(n \log n)$.

DFT inverse et FFT

DFT inverse

Soit $A = a_0 + a_1 X + \cdots + a_{n-1} X^{n-1}$ un polynôme de degré $n-1$.

- Données : $d_k = A(\omega_n^k)$ pour $0 \leq k \leq n-1$ où $\omega_n = e^{i\frac{2\pi}{n}}$
- Problème : calculer $a_0, a_1, \ldots, a_{n-1}$.

Théorème : DFT inverse et FFT

On peut calculer la DFT inverse par une FFT et donc en temps $\mathcal{O}(n \log n)$.

Diviser pour régner

Exemple : Multiplication rapide de grands nombres (Karatsuba)

Problème : calculer $x \cdot y$ de façon efficace.
Taille de z : $|z|$ est le nombre de chiffres dans l'écriture de z en base b.

Algorithme naïf : $\mathcal{O}(|x| \cdot |y|)$ (on sait multiplier 2 chiffres en temps constant).

Diviser pour régner :
On peut calculer xy avec 3 multiplications d'entiers de taille $\sim \frac{\max(|x|,|y|)}{2}$.

On en déduit un algorithme en $\mathcal{O}(\max(|x|,|y|)^{\log_2 3})$. $\log_2 3 \approx 1,585$.

Si $|x| \leq |y|$ on peut améliorer en $\mathcal{O}\left(\left\lceil \frac{|y|}{|x|} \right\rceil |x|^{\log_2 3}\right)$.

Exercice :

1. Montrer que la récurrence de partition exacte pour la multiplication rapide est
$$t(n) = t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + t(1 + \lceil n/2 \rceil) + \Theta(n)$$
et montrer que $t(n) = \mathcal{O}(n^{\log_2 3})$.
2. Montrer qu'on peut calculer xy avec 5 multiplications d'entiers de taille $1/3$. En déduire un algorithme de multiplication et sa complexité. Généraliser.

Diviser pour régner

Exercice : Exponentiation rapide

Montrer que le nombre de multiplications utilisées pour caculer a^n par exponentiation rapide est $\Theta(\log n)$.

Exercice : Exponentiation de grands nombres

Montrer que si on combine l'exponentiation rapide et la multiplication rapide, on obtient un algorithme pour calculer a^n en $\Theta\left(|a|^{\log_2 3} \cdot n^{\log_2 3}\right)$.

Plan

Introduction

Preuve et terminaison

Complexité

Paradigmes

5 Structures de données (types abstraits)

Dictionnaires et tableaux associatifs

Files de priorité

Gestion des partitions (Union-Find)

Types abstraits

Définition : Types abstraits

Un type abstrait est défini par

- Un ensemble de données,
- La liste des opérations qui permettent de manipuler ces données.

Exemple : Pile

Soit T un type. On définit pile(T) par

- Données : listes d'éléments de type T
- Opérations :

estVide :	pile(T) $\rightarrow$ booléen	
nonVide :	pile(T) $\rightarrow$ booléen	
empiler :	pile(T) $\times$ T $\rightarrow$ pile(T)	
sommet :	pile(T) $\rightarrow$ T	pré-condition : pile non vide
dépiler :	pile(T) $\rightarrow$ pile(T)	pré-condition : pile non vide
créer :	$\rightarrow$ pile(T)	constructeur
détruire :	pile(T) $\rightarrow$	

Types abstraits

Exemple : Sémantique pour les opérations de Pile

La sémantique peut être définie par des équations :

- nonVide(p) $=\neg$ estVide(p)
- estVide(créer())
- nonVide(empiler(p,x))
- sommet(empiler(p,x)) $=$ x
- dépiler(empiler(p,x)) $=$ p

Le plus souvent, la sémantique est donnée par des spécifications moins formelles, sous la forme de pré- et post-conditions pour chaque opération.

Types abstraits

On peut utiliser un type abstrait sans connaître son implémentation.

Exemple : Évaluation d'une expression postfixe

8 3 4 $\times$ + 5 3 $-$ / #

```
pile(réel) p          Appel au constructeur
s <- nextSymbol()
tq s ≠ # faire
   cas s parmi
      réel : p.empiler(s)
      +    : y <- p.sommet(); p.dépiler(); x <- p.sommet(); p.dépiler();
             p.empiler(x+y)
      log  : x <- p.sommet(); p.dépiler(); p.empiler(log(x))
      ⋮
   fincas
   s <- nextSymbol()
ftq
afficher p.sommet()
```


Implémentations d'une pile

1. Tableau
   + : toutes les opérations en $\mathcal{O}(1)$ (très efficaces).
   − : l'implémentation n'est pas vraiment dynamique.
   : Doubler la taille du tableau lorsqu'il est plein (coût amorti constant).
2. Liste chaînée
   + : toutes les opérations en $\mathcal{O}(1)$ (un peu moins efficaces).
   + : implémentation dynamique.

Types abstraits

Exercice : Enlever la récursivité

1. Écrire une fonction récursive pour évaluer une expression préfixe.
   $/ + 8 \times 3\ 4 - 5\ 3$
2. Écrire une version itérative de cette fonction.
3. Écrire une version non récursive du tri rapide.

Exercice : File

1. Définir le type abstrait File.
2. Décrire des implémentations du type File.
3. Implémenter une file à l'aide de 2 piles.
   Calculer le coût amorti d'une opération.
4. Implémenter une pile à l'aide de 2 files.
   Calculer le coût amorti d'une opération.

Types abstraits récursifs

Exemple : Arbre binaire

Soit T un type. On définit arbre(T) par

- Données : définition récursive
  Base : arbre vide, noté () ou null ou nil ...
  Induction : arbre composé d'un élément (racine) de type T et de deux sous-arbres (gauche et droit).
- Opérations :

estVide :	arbre(T) $\rightarrow$ booléen	
nonVide :	arbre(T) $\rightarrow$ booléen	
faireArbre :	T $\times$ arbre(T)^2 $\rightarrow$ arbre(T)	
racine :	arbre(T) $\rightarrow$ T	pré-condition : arbre non vide
fg :	arbre(T) $\rightarrow$ arbre(T)	pré-condition : arbre non vide
fd :	arbre(T) $\rightarrow$ arbre(T)	pré-condition : arbre non vide
créer :	$\rightarrow$ arbre(T)	constructeur
détruire :	arbre(T) $\rightarrow$	

Types abstraits récursifs

Exemple : Expression infixes parenthésées

Construction de l'arbre représentant une expression infixe.
Syntaxe : exp ::= réel | (exp op-bin exp) | op-un exp
Exemple : $(((8+3)\times 4)/(5-3))$

```
fonction expInfixe () :  arbre
Début
  s <- nextSymbol()
  cas s parmi
    réel   : retourner faireArbre(s,null,null)
    op-un  : x <- expInfixe()
             retourner faireArbre(s,x,null)
    (      : x <- expInfixe()
             s <- nextSymbol(); si s ∉ op-bin alors ERREUR fsi
             y <- expInfixe()
             t <- nextSymbol(); si t ≠ ) alors ERREUR fsi
             retourner faireArbre(s,x,y)
  fin cas
Fin
```


Types abstraits récursifs

Exercice : Évaluation

Écrire une fonction pour évaluer une expression représentée par un arbre.

Exercice : Expressions avec priorités

Écrire une fonction pour construire l'arbre représentant une expression infixe en respectant les règles de priorité et d'associativité.
Syntaxe : exp ::= réel | exp op-bin exp | op-un(exp) | (exp)
Exemple : $(8+3\times 4)/(5-3)$

Exercice : Axiomatisation

Donner l'axiomatisation du type abstrait arbre.

Exercice : Implémentation

Proposez des implémentations à l'aide de tableaux ou de pointeurs/références pour le type arbre.
Comparez leurs avantages et inconvénients.

Plan

Introduction

Preuve et terminaison

Complexité

Paradigmes

Structures de données (types abstraits)

6 Dictionnaires et tableaux associatifs
- Type abstrait
- Implémentation par tableaux et arbres de recherche
- Tables de hachage
- Arbres équilibrés a–b

Files de priorité

Dictionnaire / Tableau associatif

Définition : Dictionnaire / tableau associatif

Soit Tclé un type de clés et Tval un type de valeurs.
On définit le type dico(Tclé, Tval) par

- Données : Ensemble de couples (clé,val).
- Opérations :

$$\begin{aligned}
\text{estVide} &: \text{dico(Tclé,Tval)} \to \text{booléen}\\
\text{chercher} &: \text{dico(Tclé,Tval)} \times \text{Tclé} \to \text{Tval}\\
\text{insérer} &: \text{dico(Tclé,Tval)} \times \text{Tclé} \times \text{Tval} \to \text{dico(Tclé,Tval)}\\
\text{supprimer} &: \text{dico(Tclé,Tval)} \times \text{Tclé} \to \text{dico(Tclé,Tval)}\\
\text{créer} &: \to \text{dico(Tclé,Tval)}\\
\text{détruire} &: \text{dico(Tclé,Tval)} \to
\end{aligned}$$

On pourra retourner une valeur spéciale UNSET lors d'une recherche infructueuse.

Donner plusieurs spécifications "naturelles" pour insérer.

Dictionnaire / Tableau associatif

Exemple : Serveur pédagogique (PHP)

Tableau / dictionnaire des mots de passe : `$password`

- clé : string (email)
- valeur : string (mot de passe)

```
if ($password["paul.gastin@lsv.ens-cachan.fr"] == "TropFacile") ...
```

Tableau des notes : `$notes`

- clé : entier (identifiant étudiant, e.g., 2012043 → Marie)
- valeur : Tableau des notes : `$notes[2012043]`
  - clé : entier (identifiant du cours, e.g., 2012050 → Algorithmique)
  - valeur : réel (note)

```
$notes[$id_etud][$id_cours] = 15
```


Dictionnaire / Tableau associatif

Autres opérations

$$\begin{aligned}
\text{isset} &: \text{dico(Tclé,Tval)} \times \text{Tclé} \to \text{booléen}\\
\text{unset} &: \text{dico(Tclé,Tval)} \times \text{Tclé} \to \text{dico(Tclé,Tval)}\\
\text{taille} &: \text{dico(Tclé,Tval)} \to \text{entier}\\
\text{clé_de_val} &: \text{dico(Tclé,Tval)} \times \text{Tval} \to \text{Tclé}\\
\text{trierClé} &: \text{dico(Tclé,Tval)} \to \text{dico(Tclé,Tval)}\\
\text{trierVal} &: \text{dico(Tclé,Tval)} \to \text{dico(Tclé,Tval)}\\
\text{fusionner} &: \text{dico(Tclé,Tval)}^2 \to \text{dico(Tclé,Tval)}\\
\text{scinder} &: \text{dico(Tclé,Tval)} \times \text{Tclé} \to \text{dico(Tclé,Tval)}^2
\end{aligned}$$

Itérateur

```
trierValDecroissant(notes[idEtudiant])
ectsValid[idEtudiant] <- 0
Pour Chaque (idCours,note) dans notes[idEtudiant] faire
   si note >= 10 alors
      ectsValid[idEtudiant] <- ectsValid[idEtudiant] + ects[idCours]
   finsi
FinPour
```

Tableau non trié

Tableau classique de couples (clé,valeur)

- Doubler la taille du tableau lorsqu'il est plein.
- Complexité :

	meilleur	pire	moyen
recherche	1	n	$n/2$
insertion	1	n	$n/2$
suppression	1	n	$n/2$

Tableau trié

Tableau classique de couples (clé,valeur) trié sur la clé

- Doubler la taille du tableau lorsqu'il est plein.
- Complexité :

	meilleur	pire	moyen
recherche	1	$\log n$	$\log n$
insertion	1	n	$n/2$
suppression	1	n	$n/2$

- Si le type Tclé est numérique, on peut améliorer la recherche en faisant une interpolation :

$$g + (d - g) \times \frac{\text{clé} - t[g].\text{clé}}{t[d].\text{clé} - t[g].\text{clé}}$$

Si les clés du dictionnaire sont uniformément réparties, la complexité moyenne de la recherche est en $\mathcal{O}(\log(\log n))$.

Arbre binaire de recherche

Arbre binaire de recherche (ABR)

- Binaire : chaque nœud a 0, 1 ou 2 fils.
- de recherche : pour chaque nœud x :

$$\{\text{clés de } x.\text{fg}\} < x.\text{clé} < \{\text{clés de } x.\text{fd}\}$$

- Implémentation dynamique
- Complexité :

	meilleur	pire	moyen
recherche	1	n	$\log n$
insertion	1	n	$\log n$
suppression	1	n	$\log n$

- On peut aussi utiliser une sentinelle.

Exercice : ABR

Implémenter les primitives d'un dictionnaire avec un ABR.

Arbres binaires de recherche

Exercice : Complexité en moyenne

1. Calculer la hauteur moyenne d'un ABR construit aléatoirement.
   Univers : $\mathfrak{S}_n$ avec distribution uniforme.
   Hauteur : $H_n : \mathfrak{S}_n \to \mathbb{N}$ où $H_n(\sigma)$ (pour $\sigma \in \mathfrak{S}_n$) est la hauteur de l'ABR obtenu par insertions successives de $\sigma(1)$, $\sigma(2)$, …, $\sigma(n)$.
   Attention : ne pas confondre avec une distribution uniforme sur les ABR contenant les clés $\{1, \ldots, n\}$.
2. Calculer le coût moyen d'une recherche fructueuse/infructueuse dans un ABR construit aléatoirement.

Arbres binaires de recherche

Exercice : ABR balisés

Dans un ABR balisé, les couples (clé,val) ne sont qu'aux feuilles et les nœuds internes ont tous exactement 2 fils et contiennent des balises qui permettent d'orienter la recherche.
Si x est un nœud interne, alors

$$\{\text{clés de } x.\text{fg}\} < x.\text{balise} \leq \{\text{clés de } x.\text{fd}\}$$

1. Montrer que $n(a) = 2f(a) - 1$ si a est un ABR balisé.
2. Écrire des procédures d'insertion et de suppression dans un ABR balisé.
3. Écrire une procédure pour transformer en temps linéaire un ABR balisé en un ABR classique ayant la même structure.
4. Écrire la procédure inverse.

Arbres de recherche équilibrés

Exemples : Arbres équilibrés

- Arbres a-b
- Arbres rouge et noir
- Arbres AVL (Adelson-Velsky et Landis)

Arbres de recherche équilibrés

- La hauteur d'un arbre contenant n nœuds est au pire $\mathcal{O}(\log n)$.
- Implémentation dynamique
- Complexité :

	meilleur	pire	moyen
recherche	1	$\log n$	$\log n$
insertion	$\log n$	$\log n$	$\log n$
suppression	$\log n$	$\log n$	$\log n$

Tables de hachage

Objectif : implémenter les primitives principales du type abstrait dictionnaire en temps constant.

- Primitives: recherche, insertion, modification, suppression.
- Complexité : en moyenne et même si possible au pire.

Définition : Table de hachage

Soit `d:dico(Tclé,Tval)` un dictionnaire.
On fixe un entier $m > 0$ et on choisit une fonction de hachage

$$h : \text{Tclé} \to [m] = \{0, \ldots, m-1\}.$$

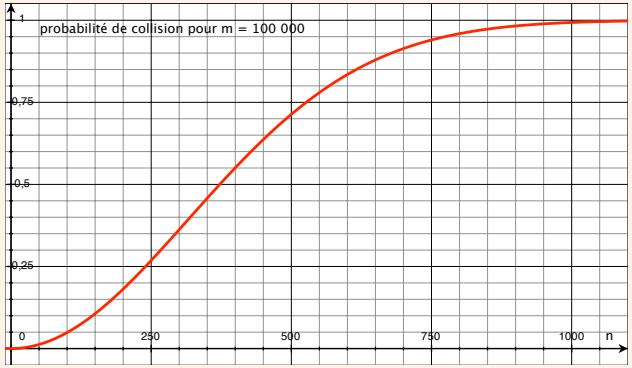
On considère un tableau t de taille m.
On vise une implémentation de `d[clé]` par `t[h(clé)]`.
Il faut résoudre les collisions : $h(c_1) = h(c_2)$ avec $c_1 \neq c_2$.

Tables de hachage

Exercice : Paradoxe des anniversaires (Richard von Mises)

Sous l'hypothèse de hachage uniforme (voir plus loin) montrer que si on tire n clés, la probabilité qu'il y ait au moins une collision est

$$1 - \frac{m!}{(m-n)!} \cdot \frac{1}{m^n} \approx 1 - e^{-\frac{n^2}{2m}}.$$



Chaînage séparé

`t[i]` pointe sur une liste chaînée de couples `(clé,val)` tels que `h(clé)=i`.

Recherche

```
fonction chercher (t:Table, clé:Tclé) :Tval
Spec :  retourne la valeur associée à la clé dans la table t
Début
   p <- t[h(clé)]
   tant que p ≠ NULL faire
      si p->clé = clé alors retourner p->val finsi
      p <- p->suiv
   ftq
   retourner UNSET
Fin
```

Exercice :

Écrire les fonctions d'insertion et de suppression.

Complexité du chaînage séparé

Définition : Hachage uniforme

Soit $\mathcal{U}$ (=`Tclé`) l'univers des clés et $p : \mathcal{U} \to [0,1]$ une distribution de probabilité.
Note : en général, p n'est pas uniforme (ex: table des symboles d'un compilateur)
Le hachage $h : \mathcal{U} \to [m]$ est uniforme si $\forall i \in [m]$, $p(h=i) = \sum_{k \in h^{-1}(i)} p(k) = \frac{1}{m}$
(et cela doit rester vrai sur $\mathcal{U} \setminus F$ pour toute "petite" partie $F \subseteq \mathcal{U}$.)

Définition : Facteur de remplissage

$\alpha = \frac{n}{m}$ où n est le nombre de clés dans la table.

Proposition : Recherche, recherche infructueuse (avec table arbitraire)

Si le hachage est uniforme, le nombre moyen de comparaisons de clés lors d'une recherche est $\leq \alpha$.
Idem pour une recherche infructueuse.
Hypothèses : la table est arbitraire (mais fixée),
la clé cherchée est tirée aléatoirement dans $\mathcal{U}$.

Complexité du chaînage séparé

Proposition : Recherche fructueuse (table construite aléatoirement)

Si le hachage est uniforme, le nombre moyen de comparaisons de clés lors d'une recherche fructueuse est $\leq 1 + \alpha/2$.
Hypothèses : La table est construite par insertion de n clés tirées aléatoirement (et supposées 2 à 2 distinctes).
La clé cherchée est tirée uniformément parmi les n clés de la table.

Corollaire : Temps moyen constant

Si le hachage est uniforme et que $\alpha = \mathcal{O}(1)$ alors les opérations sont en moyenne en $\mathcal{O}(1)$.

Exercice :

La table étant arbitraire, montrer que le nombre moyen de comparaisons de clés lors d'une recherche fructueuse est entre $(1+\alpha)/2$ et $(n+1)/2$.
Hypothèse : La clé cherchée est tirée uniformément parmi les n clés de la table.

Problèmes du hachage uniforme

- On ne maîtrise pas l'utilisation d'un dictionnaire et donc pas l'ensemble des clés utilisées dans une application.
- On ne peut donc pas faire d'hypothèses sur la distribution de probabilité de l'univers de clés.
- On ne peut pas construire une fonction de hachage uniforme indépendante des utilisations d'un dictionnaire.
- $|\texttt{Tclé}| >> m$: pour toute fonction de hachage h: `Tclé` $\to [m]$ et pour tout n (réaliste) il existe une suite de n clés qui ont toutes la même image par h.
- Si la fonction de hachage est connue, elle peut facilement être attaquée.

Solution : choisir aléatoirement la fonction de hachage.

Hachage universel

Définition : Hachage universel

On fixe l'ensemble TClé des clés, un entier $m > 0$ et une constante $c \geq 1$.

Une famille H_m de fonctions de hachage de TClé dans $[m]$ est c-universelle pour une distribution de probabilité p sur H_m si

$\forall x_1, x_2 \in$ TClé avec $x_1 \neq x_2$ on a $\quad p(h(x_1) = h(x_2)) \leq \frac{c}{m}$

Si la distribution de probabilité sur H_m est uniforme, la condition s'écrit

$\forall x_1, x_2 \in$ TClé avec $x_1 \neq x_2$ on a $\quad |\{h \in H_m \mid h(x_1) = h(x_2)\}| \leq c\frac{|H_m|}{m}$

Théorème : Chaînage séparé et hachage universel

Soit H_m une famille c-universelle de fonctions de hachage de TClé dans $[m]$.

Soit $S \subseteq$ TClé un ensemble fixé de $n = |S|$ clés. Soit $x \in$ TClé.

Si la fonction de hachage est choisie aléatoirement dans H_m alors
la complexité moyenne d'une opération sur x dans une table contenant les clés S est $\mathcal{O}(1 + c\alpha)$ où $\alpha = \frac{n}{m}$.

Opération sur x : recherche, insertion, suppression, modification.

Famille 1-universelle

On suppose qu'une clé peut être codée par une suite de q bits : $|\text{TClé}| \leq 2^q$.

Définition : Hachage universel

Soit m un nombre premier et $p = \lfloor \log_2 m \rfloor$.

On tronçonne une clé $x \in$ TClé en k blocs de p bits ($kp \geq q$):
$\mathbf{x} = (x_1, \ldots, x_k)$ et on interprète chaque x_i comme un entier $< 2^p \leq m$.

On choisit $\mathbf{a} = (a_1, \ldots, a_k) \in [m]^k$.

On définit la fonction $h_{\mathbf{a}}$ par

$$\begin{aligned} h_{\mathbf{a}}(x) &= \mathbf{a} \cdot \mathbf{x} \mod m \\ &= \sum_{1 \leq i \leq k} a_i x_i \mod m \end{aligned}$$

Théorème : Hachage universel

Avec une distribution uniforme, la famille $H_{m,k} = \{h_{\mathbf{a}} \mid \mathbf{a} \in [m]^k\}$ est 1-universelle.

Hachage universel en pratique

Lors de la création d'un dictionnaire:

- Choisir un nombre premier m (e.g., $m = 257$).
- Choisir aléatoirement une fonction de hachage $h \in H_{m,k}$, i.e., un tuple $\mathbf{a} \in [m]^k$.

Lorsque le facteur de remplissage augmente (e.g., $\alpha \geq 1$)
ou que certaines listes sont trop longues:

- Choisir un nouveau nombre premier $m' \geq 2m$.
  Choisir m' dans une table ou avec un algo en temps $\mathcal{O}(m)$.
- Choisir aléatoirement une nouvelle fonction de hachage dans $H_{m',k}$, i.e., un tuple $\mathbf{a}' \in [m']^k$.
- Redistribuer les couples (clé,valeur) dans la nouvelle table.

Exercice : rehashing

Montrer que le coût amorti du rehashing est constant.

Familles universelles

Exercice : Produit de matrices

Soit $M \in \{0,1\}^{p \times q}$ une matrice booléenne.

On interprète une clé de q bits comme un vecteur $x \in \{0,1\}^q$.

On définit la fonction de hachage $h_M(x) = Mx \mod 2$.

On interprète ce vecteur de p bits comme un entier.

Montrer que la famille $H_{2^p} = \{h_M \mid M \in \{0,1\}^{p \times q}\}$ est 1-universelle.

Montrer comment calculer $h_M(x)$ efficacement.

Exercice : 2-universelle

Montrer que pour tout entier $m > 0$, il existe une famille 2-universelle de fonctions de hachage dans $[m]$.

Hachage parfait

Hypothèse : L'ensemble $S = \{x_1, \ldots, x_n\}$ des clés du dictionnaire est fixé.
Objectif : complexité au pire en temps constant (recherche, modification).

Exemples :

- Les mots clés d'un langage de programmation.
- Les étudiants inscrits sur le serveur pédagogique durant une année universitaire.

Problème.

On cherche un algorithme qui, étant donné un ensemble S de n clés, construit une fonction de hachage injective $h : S \to [m]$ telle que

- le taux de remplissage est bon, e.g., $\frac{1}{3} \leq \alpha$ et donc $m \leq 3n$,
- La construction de la fonction h est efficace : $\mathcal{O}(n)$ en moyenne.
- Pour chaque clé x, le calcul de $h(x)$ est efficace : $\mathcal{O}(1)$ (indépendant de n).

Hachage parfait

Soit H_m une famille c-universelle de fonctions de hachage dans $[m]$.
Soit $X(h) = |\{(x, y) \in S^2 \mid x \neq y \text{ et } h(x) = h(y)\}$.

Lemme :

- $E(X) \leq \frac{cn(n-1)}{m}$
- $\mathbb{P}(\text{non injective}) \leq \frac{1}{2}\frac{cn(n-1)}{m}$

Première solution

S'il n'y avait pas la contrainte $m = \mathcal{O}(n)$ on aurait l'algorithme

$m \leftarrow \lceil \delta n(n-1) \rceil$
`Répéter choisir` h `aléatoirement dans` H_m
`Jusqu'à trouver une fonction` h `injective`

Chaque tirage est indépendant (Bernoulli).
Si $c = 2$ et $\delta = \frac{3}{2}$ alors la probabilité de succès est $b \geq \frac{1}{3}$.
Le nombre moyen de tirages pour un succès est $\frac{1}{b} \leq 3$.

Tester si h est injective: temps $\mathcal{O}(n + m)$ en supposant le calcul $h(x)$ en $\mathcal{O}(1)$.

Hachage parfait

Deuxième solution

$n' \leftarrow \lceil \gamma n \rceil$
`Répéter`
  `choisir` h `aléatoirement dans` $H_{n'}$
  `Pour chaque` $i \in [n']$ `faire` $n_i \leftarrow 0$; $S_i \leftarrow \emptyset$ `fpour`
  `Pour chaque` $x \in S$ `faire` $i \leftarrow h(x)$; n_i`++`; $S_i \leftarrow S_i \cup \{x\}$ `fpour`
  $s_0 \leftarrow 0$
  `Pour chaque` $i \in [n']$ `faire` $m_i \leftarrow \lceil \delta n_i(n_i - 1) \rceil$; $s_{i+1} \leftarrow s_i + m_i$ `fpour`
  $m \leftarrow s_{n'}$
`Jusqu'à avoir` $m \leq \beta n$

`Pour chaque` $i \in [n']$ `faire`
  `Répéter choisir` h_i `aléatoirement dans` H_{m_i}
  `Jusqu'à avoir` h_i `injective sur` S_i
`fpour`

`Fonction g(TClé):`$[m]$
  $i \leftarrow h(x)$; `Retourner` $s_i + h_i(x)$

Théorème : Hachage parfait $\beta = 3$, $\gamma = \delta = \frac{3}{2}$

L'algorithme ci-dessus termine en moyenne en temps $\mathcal{O}(n)$.
La fonction g est injective et la taille de la table est $m \leq 3n$.

Autres fonctions de hachage

Définition : Hachage par division

$h : \mathbb{N} \to [m]$ définie par $h(k) = k \mod m$.

- Éviter $m = 2^p$ ou $m = 10^p$ car $h(k)$ ne dépendrait que des bits de poids faible
- Éviter $m = |\Sigma| - 1$ car $h(u_\ell \cdots u_0)$ serait invariant par permutation des lettres
- Choisir un nombre premier éloigné d'une puissance de 2.

Autres fonctions de hachage

Définition : Hachage par multiplication

Choisir $\theta \in]0,1[$ et définir $h(k) = \lfloor m \cdot \text{frac}(k \cdot \theta) \rfloor \in [m]$
où $\text{frac}(x) = x - \lfloor x \rfloor$ est la partie fractionnaire d'un réel $x \geq 0$.

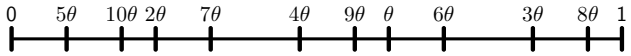
Théorème : Vera Turan Sos (1957)

Soit $\theta > 0$ un nombre irrationnel. Soit $0 < a_1 < \cdots < a_n < 1$ la suite ordonnée des valeurs $\text{frac}(\theta)$, $\text{frac}(2\theta)$, ..., $\text{frac}(n\theta)$.

Les segments $[0, a_1]$, $[a_1, a_2]$, ..., $[a_n, 1]$ ont au plus 3 longueurs différentes.
Le prochain point $\text{frac}((n+1)\theta)$ tombe dans l'un des plus grands segments.

Conséquence : le hachage par multiplication répartit bien les clés $\{1, \ldots, n\}$.

Avec $\theta = \frac{\sqrt{5}-1}{2}$:



- On choisit pour m une puissance de 2 : $m = 2^p$
- On utilise q bits significatifs pour θ : $\lfloor \theta \cdot 2^q \rfloor$
  q : taille d'un mot mémoire.
- Il suffit d'une multiplication entière $k \cdot \lfloor \theta \cdot 2^q \rfloor$ et de décalages.

Adressage ouvert

Définition : Fonction de hachage

$$h : \mathcal{U} \to \mathfrak{S}_m \qquad \text{ou} \qquad h : \mathcal{U} \times [m] \to [m]$$

Pour une clé $k \in \mathcal{U}$ on utilise la séquence de sondage $h(k)$, i.e.,

$$h(k,0), h(k,1), \ldots, h(k, m-1)$$

Recherche

```
fonction chercher (t:Table, clé:Tclé) :Tval
Spec :  retourne la valeur associée à la clé dans la table t
Début
   pour i <- 0 à m-1 faire
      j <- h(clé,i)
      si t[j].clé = clé alors retourner t[j].val finsi
      si t[j].clé = cléVide alors retourner UNSET finsi
   fpour
   retourner UNSET
Fin
```


Adressage ouvert

Exercice :

1. Écrire une fonction pour l'insertion.
2. Écrire une fonction pour la suppression.
   Utiliser une constante `cléEnlevée`.

Adressage ouvert versus chaînage séparé

+ : Pas d'espace perdu pour les pointeurs.
− : La table ne peut pas contenir plus de m couples `(clé,val)` : $\alpha = \frac{n}{m} \leq 1$.
− : (rehashing) Lorsque le taux de remplissage s'approche de 1,
il faut augmenter la taille de la table et re-distribuer les clés.

Sondage linéaire

Définition : Sondage linéaire

Soit $h_1 : \mathcal{U} \to [m]$ une fonction de hachage ordinaire.
On définit $h : \mathcal{U} \times [m] \to [m]$ par

$$h(k,i) = (h_1(k) + i) \mod m$$

+ : Simplicité.
− : Problème de la grappe forte.
− : Seulement m séquences de sondage.

Sondage quadratique

Définition : Sondage quadratique

Soit $h_1 : \mathcal{U} \to [m]$ une fonction de hachage ordinaire.
On définit $h : \mathcal{U} \times [m] \to [m]$ par

$$h(k,i) = (h_1(k) + c_1 \cdot i + c_2 \cdot i^2) \mod m$$

En choisisssant c_1, c_2 et m pour que la séquence de sondage soit une permutation.

+ : Plus efficace que le sondage linéaire.
– : Seulement m séquences de sondage.
– : Problème de la grappe faible.

Exercice :

Montrer qu'avec $c_1 = c_2 = 1/2$ et $m = 2^p$ la séquence de sondage est une permutation.

Double hachage

Définition : Double hachage

Soit $h_1 : \mathcal{U} \to [m]$ une fonction de hachage ordinaire.
Soit $h_2 : \mathcal{U} \to [m]$ une fonction de hachage ordinaire.
On définit $h : \mathcal{U} \times [m] \to [m]$ par

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m$$

+ : Une des meilleures méthodes.
+ : $\Theta(m^2)$ séquences de sondage.

Exercice :

Montrer que si h_2 génère des valeurs premières avec m alors les séquences de sondage sont des permutations.

Exemple : Double hachage

- $h_1(k) = k \mod m$ avec m premier et $h_2(k) = 1 + (k \mod m')$ avec $1 < m' < m$.
- $h_1(k) = k \mod m$ avec $m = 2^p$ et $h_2(k)$ impair.

Complexité de l'adressage ouvert

Définition : Hachage uniforme

Soit $\mathcal{U}$ l'univers des clés et $p : \mathcal{U} \to [0,1]$ une distribution de probabilité.
Note : en général, p n'est pas uniforme (ex: table des symboles d'un compilateur)
Le hachage $h : \mathcal{U} \to \mathfrak{S}_m$ est uniforme si $\forall \sigma \in \mathfrak{S}_m$,

$$p(h = \sigma) = \sum_{k \in h^{-1}(\sigma)} p(k) = \frac{1}{m!}$$

Définition : Facteur de remplissage

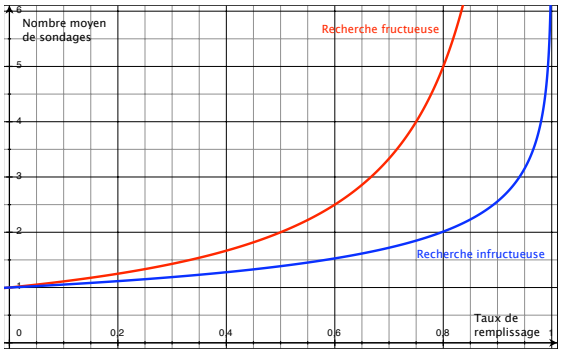
$\alpha = \frac{n}{m}$ où n est le nombre de clés dans la table.

Complexité de l'adressage ouvert

Proposition : Recherche

Si le hachage est uniforme, le nombre moyen de sondages

- lors d'une recherche infructueuse est $\frac{m+1}{m+1-n} \sim \frac{1}{1-\alpha}$.
- lors d'une recherche fructueuse est $\frac{m+1}{n}(H_{m+1} - H_{m-n+1}) \leq \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$ où $H_j = 1 + 1/2 + \cdots + 1/j$ est le $j$ème nombre harmonique.



Arbres équilibrés a–b

Définition : Arbres a–b avec $a \geq 2$ et $b \geq 2a-1$ ou $b \geq 2a$

- La racine, possède entre 1 et $b-1$ clés.
- Chaque nœud autre que la racine possède entre $a-1$ et $b-1$ clés.
- Les nœuds internes ont un fils de plus que de clés.
- Toutes les feuilles sont à la même profondeur.
- C'est un arbre de recherche.

Notations : Soit x un nœud de l'arbre

- d(x) : degré du nœud x, i.e., nombre de fils
- x.c[1],. . . ,x.c[d(x)-1] : clés du nœud x
- x.f[1],. . . ,x.f[d(x)] : fils du nœud x

Arbre de recherche

{clés de x.f[1]} < x.c[1] < $\cdots$ < x.c[d(x)-1] < {clés de x.f[d(x)]}

Convention : x.c[0] $= -\infty$ et x.c[d(x)] $= +\infty$

Recherche dans un arbre a–b

Algorithme de recherche

```
fonction chercherAB (x:arbre, clé:Tclé) :Tval
Spec :  retourne la valeur associée à la clé dans l'arbre
Début
   si x.estVide() alors retourner UNSET finsi
   Trouver 1 ≤ i ≤ d(x) tel que x.c[i-1] < clé ≤ x.c[i]
   si clé = x.c[i] alors retourner x.val[i] finsi
   retourner chercherAB(x.f[i], clé)
Fin
```

Lemme : Hauteur et complexité

Soit x un arbre de hauteur h comportant n clés. on a

$$\log_b(n+1) \leq h \leq \log_a(n+1) + 1 - \log_a 2$$

La complexité au pire de la recherche est en $\log_a b \times \log_2 n$.

$\log_2 3 \approx 1,58$ $\log_2 4 = 2$ $\log_3 5 \approx 1,46$ $\log_3 6 \approx 1,63$ $\log_4 7 \approx 1,4$

Insertion dans un arbre a–b

- Exemple d'insertion dans un arbre 2-4.
- Éclatement d'un nœud (trop) plein.
- Méthode préventive à la descente.
  + : Implémentation simple en récursif ou itératif.
  – : Des éclatements inutiles qui augmentent les problèmes à la suppression.
  – : Nécessite $b \geq 2a$.
- Méthode curative à la remontée.
  + : Il suffit de $b \geq 2a-1$.
  + : Pas d'éclatement inutile.
  – : Implémentation plus compliquée.
- Complexité $\mathcal{O}(\log n)$.

Suppression dans un arbre a–b

- Exemple de suppression dans un arbre 2-4.
- Fusion d'un nœud avec son frère.
- Partage entre un nœud et son frère.
  Le partage peut aussi être utilisé à l'insertion car il arrête le ré-équilibrage.
  Si on a le choix, on fait plutôt un partage qu'une fusion ou un éclagement.
- Méthode préventive à la descente.
  + : Implémentation simple en récursif ou itératif.
  – : Des fusions inutiles qui augmentent les problèmes à l'insertion.
  – : Nécessite $b \geq 2a$.
- Méthode curative à la remontée.
  + : Il suffit de $b \geq 2a-1$.
  + : Pas de fusion inutile.
  – : Implémentation plus compliquée.
- Complexité $\mathcal{O}(\log n)$.

Coût amorti en ré-équilibrages

Théorème : Ré-équilibrages pour la méthode curative avec $b \geq 2a \geq 4$.

On considère une suite de I insertions et S suppressions à partir d'un arbre vide. On note P le nombre de partages, E le nombre d'éclatements et F le nombre de fusions. On a

$$E + F + P \leq \frac{2a-1}{a}(I+S)$$

Le coût amorti en ré-équilibrages est donc $\leq \frac{2a-1}{a} \leq 2$.

Exercice :

Calculer le coût amorti en ré-équilibrages pour les arbres 2-3.

Arbres a–b

Implémentation d'un nœud

- Tableau trié.
  - $+$: Implémentation simple.
  - $+$: Recherche en $\log_2 b$ dans le nœud.
  - $-$: Gaspillage de place.
  - $-$: décalages et recopies pour le partage, la fusion et l'éclatement: $\mathcal{O}(b)$
- Liste chaînée avec sentinelle $+\infty$.
  - $+$: Espace optimisé.
  - $-$: Recherche en $\mathcal{O}(b)$.
  - $-$: Il faut calculer le degré ou le mémoriser dans tous les nœuds.
- Arbre binaire équilibré
  - $+$: Espace optimisé.
  - $+$: Recherche en $\mathcal{O}(\log_2 b)$.
  - $-$: Il faut parfois rééquilibrer l'arbre avec des rotations.

Exemple :

Implémentation en Java du type abstrait dictionnaire avec des arbres a–b:
`http://www.lsv.ens-cachan.fr/~gastin/Algo/monDico.java`

Arbres a–b

Exercice :

Comparer les arbres 2-4 et les arbres bicolores.

Exercice :

Écrire les algorithmes d'union et de scission pour les arbres a–b.

Plan

Introduction

Preuve et terminaison

Complexité

Paradigmes

Structures de données (types abstraits)

Dictionnaires et tableaux associatifs

7 Files de priorité
- Tas binaires
- Tas binomiaux
- Tas de Fibonacci

Gestion des partitions (Union-Find)

Type abstrait File de priorité

Définition : File de priorité

Soit Tprio un type ordonné et Tinfo un type arbitraire.
On définit le type FPrio(Tprio, Tinfo) par

- Données : Ensemble de couples (prio,info).
- Opérations :

$$
\begin{aligned}
\text{créer} &: \to \text{FPrio(Tprio,Tinfo)} \\
\text{détruire} &: \text{FPrio(Tprio,Tinfo)} \to \\
\text{estVide} &: \text{FPrio(Tprio,Tinfo)} \to \text{booléen} \\
\text{insérer} &: \text{FPrio(Tprio,Tinfo)} \times \text{Tprio} \times \text{Tinfo} \to \text{FPrio(Tprio,Tinfo)} \\
\text{minimum} &: \text{FPrio(Tprio,Tinfo)} \to \text{Tprio} \times \text{Tinfo} \\
\text{supprimerMin} &: \text{FPrio(Tprio,Tinfo)} \to \text{FPrio(Tprio,Tinfo)} \\
\text{fusionner} &: \text{FPrio(Tprio,Tinfo)}^2 \to \text{FPrio(Tprio,Tinfo)}
\end{aligned}
$$

Dans certains cas, on peut vouloir modifier la priorité d'un couple pointé par p.

$$
\begin{aligned}
\text{diminuerPrio} &: \text{FPrio(Tprio,Tinfo)} \times \text{Ptr} \times \text{Tprio} \to \text{FPrio(Tprio,Tinfo)} \\
\text{augmenterPrio} &: \text{FPrio(Tprio,Tinfo)} \times \text{Ptr} \times \text{Tprio} \to \text{FPrio(Tprio,Tinfo)} \\
\text{supprimer} &: \text{FPrio(Tprio,Tinfo)} \times \text{Ptr} \to \text{FPrio(Tprio,Tinfo)}
\end{aligned}
$$

Implémentation d'une file de priorité

Définition : Tournoi

Arbre dans lequel la priorité d'un noeud est inférieure aux priorités contenues dans ses sous-arbres.

Définition : Arbre parfait

Arbre dans lequel tous les niveaux sont pleins sauf éventuellement le dernier dont les feuilles sont à gauche.

Un arbre parfait s'implémente efficacement avec un tableau.

Définition : Tas binaire = tournoi binaire parfait

Implémentation d'une file de priorité avec un tas binaire.

F.n : nombre d'éléments dans la file de priorité
F.prio : tableau [1..Max] des priorités
F.info : tableau [1..Max] des informations associées

Pour simplifier on oublie l'information associée à l'élément.

File de priorité et tas binaire

Implémentation

```
fonction estVide () :  booléen
   retourner (F.n = 0)

fonction minimum () :  Tprio
   retourner (si F.n > 0 alors F.prio[1] sinon prioVide)

Procédure insérer (c:Tprio)
   F.n++; F.prio[F.n] <- c; F.remonter(F.n)

Procédure remonter (p:Int)
   tant que p > 1 et F.prio[p div 2] > F.prio[p] faire
      échanger(F.prio[p div 2], F.prio[p]); p <- p div 2
   fintq

Procédure diminuerPrio (p:Int, c:Tprio)
   si 1 ≤ p ≤ F.n et c < F.prio[p] alors
      F.prio[p] <- c; F.remonter(p)
   fsi
```


File de priorité et tas binaire

Implémentation

```
Procédure supprimerMin ()
   si F.n > 0 alors
      F.prio[1] <- F.prio[F.n]; F.n--; F.descendre(1)
   fsi

Procédure descendre (p:Int)
   tant que 2p ≤ F.n faire
      p <- 2p
      si p+1 ≤ F.n et F.prio[p+1] < F.prio[p] alors p++ fsi
      si F.prio[p div 2] ≤ F.prio[p] alors exit_tq fsi
      échanger(F.prio[p div 2], F.prio[p])
   fintq

Procédure augmenterPrio (p:Int, c:Tprio)
   si 1 ≤ p ≤ F.n et c > F.prio[p] alors
      F.prio[p] <- c; F.descendre(p)
   fsi
```

File de priorité et tas binaire

Implémentation

```
Procédure supprimer (p:Int)
   si 1 ≤ p ≤ F.n alors
      F.prio[p] <- F.prio[F.n]; F.n--
      F.remonter(p); F.descendre(p)
   fsi
```

Pas de méthode efficace (i.e., en $\mathcal{O}(\log n)$) pour la fusion.

Complexité au pire : récapitulatif

$\Theta(1)$: estVide, minimum
$\Theta(\log n)$: insérer, supprimerMin
$\Theta(\log n)$: diminuerPrio, augmenterPrio, supprimer
$\Theta(n)$: fusion

Tas binaire

Exercice : Tri par tas (Williams, 1964)

Les opérations doivent se faire en place dans le tableau.

1. Transformer un tableau en tas binaire.
2. Transformer un tas binaire en tableau trié.
3. Étudier la complexité au pire de ces deux opérations.
4. Montrer que la construction du tas peut se faire en $\mathcal{O}(n)$.

Arbres binomiaux

Définition : Arbre binomial

Un arbre binomial est un arbre ordonné défini récursivement :

- B_0 est constitué d'un unique noeud.
- B_k est constitué de deux arbres B_{k-1}, l'un étant ajouté comme fils le plus à gauche au second.

Dessiner les arbres B_0 à B_4.

Lemme : Propriétés combinatoires

1. L'arbre B_k comporte 2^k noeuds.
2. L'arbre B_k est de hauteur k.
3. L'arbre B_k comporte $\binom{k}{i}$ noeuds à la profondeur i
4. La racine de B_k est de degré k et ses fils de gauche à droite sont $B_{k-1}, B_{k-2}, \ldots, B_1, B_0$.

Arbres binomiaux

Implémentation d'un arbre binomial

Problème : le degré des noeuds n'est pas constant.
Solution : pour les fils d'un noeud, on utilise une liste chaînée.
Chaque noeud est une structure comportant

- la priorité,
- l'information associée,
- le degré du noeud,
- un pointeur vers le premier fils du noeud,
- un pointeur vers le frère (droit) du noeud,
- un pointeur vers le père du noeud.

Tas binomiaux

Définition : Tas binomial

Un tas binomial est un ensemble d'arbres binomiaux vérifiant :

- chaque arbre binomial est un tournoi.
- $\forall k \geq 0$, il existe au plus un arbre binomial de degré k dans le tas binomial

Remarque :

Soit t un tas binomial contenant n éléments.
Soit $n = \overline{c_p \cdots c_1 c_0}^2$ l'écriture binaire de n.
Alors, t contient un arbre binomial de degré k ssi $c_k = 1$.

Implémentation d'un tas binomial

On utilise une liste chaînée d'arbres binomiaux classés par degrés croissants.

Pour le chaînage, on utilise le pointeur "frère" des racines des arbres binomiaux.

F.tête est le pointeur vers le premier arbre binomial de la liste.

On utilise une sentinelle de degré ∞ à la fin de la liste des arbres binomiaux.

File de priorité et tas binomiaux

Implémentation

```
fonction estVide () :  booléen
   retourner (F.tête.degré = ∞)

fonction minimum () :  Tprio
   min <- ∞; x <- F.tête
   tant que x.degré ≠ ∞ faire
      si x.prio < min alors min <- x.prio fsi
      x <- x.frère
   ftq
   retourner min
```

Rem : on peut aussi mémoriser et retourner un pointeur vers le noeud contenant la priorité minimale.

Complexité au pire : $\mathcal{O}(\log n)$.

File de priorité et tas binomiaux

Implémentation : c'est une addition en binaire

```
procédure FusionTas(x,y,z)
Hyp : x et y tas binomiaux se terminant par un maillon de degré ∞
Spec : retourne un tas binomial z contenant l'union des éléments de x et y
z <- NULL; t <- NULL; r <- NULL     t dernier arbre de z et r la retenue
TQ x.degré < ∞ ou y.degré < ∞ faire
 si r = NULL alors cas parmi
  x.degré < y.degré : a <- x; x <- x.frère; ajouter(z,t,a)
  x.degré > y.degré : a <- y; y <- y.frère; ajouter(z,t,a)
  x.degré = y.degré : a <- x; x <- x.frère;
                      b <- y; y <- y.frère; r <- FusionBin(a,b)
 sinon cas parmi
  r.degré < x.degré, y.degré : ajouter(z,t,r); r <- NULL
  r.degré = x.degré < y.degré : a <- x; x <- x.frère; r <- FusionBin(a,r)
  r.degré = y.degré < x.degré : a <- y; y <- y.frère; r <- FusionBin(a,r)
  r.degré = y.degré = x.degré : ajouter(z,t,r); a <- x; x <- x.frère;
                                b <- y; y <- y.frère; r <- FusionBin(a,b)
 finsi
FTQ
si r ≠ NULL alors ajouter(z,t,r) fsi
ajouter(z,t,x)     pour avoir le maillon de degré ∞ à la fin de z
Libérer(y)
```

File de priorité et tas binomiaux

Implémentation

```
fonction FusionBin(a,b) : arbre                                    O(1)
Hyp : a et b sont des arbres binomiaux non NULL de même degré
Spec : retourne l'arbre r fusion de a et b
   a.frère <- NULL; b.frère <- NULL
   si b.prio < a.prio alors échanger(a,b) fsi
   b.père <- a; b.frère <- a.fils; a.fils <- b; a.degré++
   retourner a

Procédure Ajouter(z,t,x)                                           O(1)
Hyp : z est un tas binomial et t un pointeur sur le dernier arbre de z
      x est un arbre binomial avec t.degré < x.degré (si non NULL)
Spec : ajoute l'arbre x à la fin du tas binomial z
   x.frère <- NULL
   si t = NULL alors z <- x sinon t.frere <- x fsi
   t <- x
```

File de priorité et tas binomiaux

Complexité

- FusionBin est efficace car les fils sont rangés par degrés décroissants.
- FusionTas est efficace car les arbres sont rangés par degrés croissants.
- Complexité au pire de FusionTas : $\mathcal{O}(\log n)$ où n est le nombre d'éléments du tas résultant.

File de priorité et tas binomiaux

Implémentation

```
Procédure insérer(c:Tprio)
   y <- nouveauNoeud();
   y.degré <- 0; y.fils <- NULL; y.père <- NULL; y.prio <- c
   y.frère <- nouveauNoeud(); y.frère.degré <- ∞
   x <- F.tête
   FusionTas(x,y,z)
   F.tête <- z

Procédure supprimerMin()
   Trouver la racine z de priorité minimale dans la liste F.tête
   et supprimer simultanément l'arbre z de la liste F.tête
   Créer la liste y des fils de z dans l'ordre inverse
   et terminer la liste y par un maillon de degré ∞
   x <- F.tête
   FusionTas(x,y,z)
   F.tête <- z
```

Complexité au pire : $\mathcal{O}(\log n)$.

File de priorité et tas binomiaux

Implémentation

```
Procédure diminuerPrio(x, c:Tprio)
C'est ici que le pointeur vers le père est utile.
Spec : diminuer la priorité du noeud pointé par x
   si x.prio ≤ c alors retourner finsi
   x.prio <- c
   tant que x.père ≠ NULL et x.prio < x.père.prio faire
      échanger(x.prio, x.père.prio)
      x <- x.père
   ftq

Procédure supprimer(x)
Spec : supprime l'élément pointé par x
   F.diminuerPrio(x, −∞)
   F.supprimerMin()
```

Complexité au pire : $\mathcal{O}(\log n)$.

File de priorité et complexité

Complexité au pire : tas binaires

$\Theta(1)$: estVide, minimum
$\Theta(\log n)$: diminuerPrio, insérer
$\Theta(\log n)$: augmenterPrio, supprimerMin, supprimer
$\Theta(n)$: fusion

Complexité au pire : tas binomiaux

$\Theta(1)$: estVide
$\Theta(\log n)$: minimum, fusion
$\Theta(\log n)$: insérer, supprimerMin
$\Theta(\log n)$: diminuerPrio, supprimer

Coût amorti : tas de Fibonacci

$\Theta(1)$: estVide
$\Theta(1)$: insérer, minimum
$\Theta(1)$: fusion, diminuerPrio
$\Theta(\log n)$: supprimerMin, supprimer

Plan

Introduction

Preuve et terminaison

Complexité

Paradigmes

Structures de données (types abstraits)

Dictionnaires et tableaux associatifs

Files de priorité

8 Gestion des partitions (Union-Find)

Gestion de partitions

Définition : Type abstrait Partition

- Donnée : une partition P d'un ensemble $\{1, \ldots, N\}$ et un représentant pour chaque classe de P.
- Opérations :

$$\begin{aligned} \text{créer} &: \text{Int} \to \text{Partition} \quad \text{constructeur} \\ \text{find} &: \text{Partition} \times \text{Int} \to \text{Int} \\ \text{union} &: \text{Partition} \times \text{Int} \times \text{Int} \to \text{Partition} \\ \text{détruire} &: \text{Partition} \to \end{aligned}$$

- `Partition(N) P` crée une partition P formée de N singletons.
- `P.find(x)` retourne le représentant de la classe de x.
  Les représentants des classes ne sont pas modifiés.
- `P.union(x,y)` fusionne les classes de x et y.
  Les représentants des autres classes ne sont pas modifiés.

Gestion de partitions

Exemple : Composantes connexes d'un graphe $G = (S, A)$

On suppose $S = \{1, \ldots, N\}$.

```
Partition(N) P
Pour chaque (x,y) ∈ A faire
   x' <- P.find(x); y' <- P.find(y)
   si x' ≠ y' alors P.union(x',y') fsi
Fin pour
```

Exemple : Algorithme de Kruskal

Arbre couvrant minimum dans un graphe pondéré $G = (S, A, w)$.
On suppose $S = \{1, \ldots, N\}$.

```
Partition(N) P
B ← ∅
Pour chaque (x,y) ∈ A (par poids croissants) faire
   x' <- P.find(x); y' <- P.find(y)
   si x' ≠ y' alors B ← B ∪ {(x,y)}; P.union(x',y') fsi
Fin pour
Retourner B
```


Gestion de partitions

Implémentation : forêt

Chaque arbre de la forêt contient les éléments d'une classe.
La forêt est représentée par un tableau `père[1..N]` avec `père[x] = x` si `x` racine.

```
procédure créer(N)
   Pour x <- 1 à N faire père[x] <- x fpour
Si N est grand, utiliser l'initialisation virtuelle

fonction find(x)
   Tant que père[x] ≠ x faire x <- père[x] ftq
   retourner x

procédure union(x,y)
Hyp :  x = find(x) et y = find(y)
   si x ≠ y alors père[x] <- y fsi
```

Remarque : une union arbitraire se réalise par `P.union(P.find(x),P.find(y))`

Proposition : Complexité

Une suite d'opérations comportant $n-1$ unions et m find (dans un ordre arbitraire) se réalise en temps $\mathcal{O}(n + mn)$.

Gestion de partitions

Union pondérée par taille [2]

`taille[1..N]` mémorise (un majorant de) la taille des arbres de racines `1,...,N`.

```
procédure créér(N)
   Pour x <- 1 à N faire taille[x] <- 1; père[x] <- x fpour
procédure union(x,y)
Hyp :  x = find(x) et y = find(y)
   si x ≠ y alors
      si taille[y] < taille[x] alors échanger(x,y) fsi
      père[x] <- y
      taille[y] <- taille[x] + taille[y]
   fsi
```

Proposition : Union pondérée par taille

Une suite d'opérations comportant $n-1$ unions par taille et m find (dans un ordre arbitraire) se réalise en temps $\mathcal{O}(n + m(1 + \log_2 n))$.

Gestion de partitions

Union pondérée par hauteur [4]

`haut[1..N]` mémorise (un majorant de) la hauteur des arbres de racines `1..N`.

```
procédure créér(N)
   Pour x <- 1 à N faire haut[x] <- 0; père[x] <- x fpour
procédure union(x,y)
Hyp :  x = find(x) et y = find(y)
   si x ≠ y alors
      si haut[y] < haut[x] alors échanger(x,y) fsi
      père[x] <- y
      haut[y] <- max(haut[y], 1+haut[x])
   fsi
```

Exercice : Union pondérée par hauteur

Montrer qu'une suite d'opérations comportant $n-1$ unions par hauteur et m find (dans un ordre arbitraire) se réalise en temps $\mathcal{O}(n + m(1 + \log_2 n))$.

Gestion de partitions

Compression de chemins

```
fonction find(x)
   z <- x
   Tant que père[x] ≠ x faire x <- père[x] ftq
   Tant que père[z] ≠ x faire
      y <- père[z]; père[z] <- x; z <- y
   ftq
   retourner x
```

Proposition : Compression des chemins

Une suite d'opérations comportant $n-1$ unions simples et m find avec compression (dans un ordre arbitraire) se réalise en temps $\Theta(n + m(1 + \log_{2+m/n} n))$.

Gestion de partitions

Définition : Fonction d'Ackermann (variante) et inverse

$$A_0 : j \mapsto j+1 \qquad A_{k+1} : j \mapsto A_k^{(j)}(j) = \underbrace{A_k \circ \cdots \circ A_k}_{j \text{ fois}}(j)$$

En particulier,

$$A_1(j) = 2j \qquad A_2(j) = j2^j \geq 2^j \qquad A_3(j) \geq 2^{\left.\cdot^{\cdot^{\cdot^{2}}}\right\} j \text{ fois}}$$

De plus, on a $A_k(0) = 0$ pour $k \geq 1$, $A_k(1) = 2$ pour $k \geq 0$, et

$$A_0(2) = 3 \quad A_1(2) = 4 \quad A_2(2) = 8 \quad A_3(2) = 2048 \quad A_4(2) \geq 2^{\left.\cdot^{\cdot^{\cdot^{2}}}\right\} 2048 \text{ fois}}$$

On définit aussi l'inverse $\alpha : \mathbb{N} \to \mathbb{N}$ par

$$\alpha(n) = \min\{k \geq 0 \mid A_k(2) \geq n\}\,.$$

Remarque : $\alpha(n) \leq 4$ en pratique.

Gestion de partitions

Théorème : Union pondérée + compression des chemins — Tarjan

Une suite d'opérations comportant $n-1$ unions pondérées (par taille ou par hauteur) et m find avec compression (dans un ordre arbitraire) se réalise en temps $\mathcal{O}((n+m)(1+\alpha(n)))$.

Corollaire : Le coût amorti est constant en pratique

Preuve : Voir [7, 2, 4]

Le coût d'une opération d'union est constant.

Si $n \leq 4$ alors le coût total des m opérations find est $\mathcal{O}(m)$.

On suppose dans la suite $n \geq 5$.

On veut calculer le coût total des opérations find.