Algorithmique

Paul Gastin

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

L3 Informatique Cachan
2010-2011

Plan

1 Introduction

Preuve et terminaison

Complexité

Structures de données (types abstraits)

Structures de recherche (Dictionnaires)

Files de priorité

Paradigmes

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).

Bibliographie

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

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

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

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

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

Plan

Introduction

2 Preuve et terminaison

Complexité

Structures de données (types abstraits)

Structures de recherche (Dictionnaires)

Files de priorité

Paradigmes

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 à la fin la couleur de la boule dans l'urne ?

Invariants

Exemple : Mystère

Que fait l'algorithme suivant ?

```
Lire N
a <- 0; s <- 1; t <- 1
Tant que s ≤ N faire Inv :  s = (a+1)^2
   a <- a+1; s <- s+t+2; t <- t+2
Fin tant que
Afficher a
```

Preuve d'algorithmes

cf. [9, 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 $\{\varphi\}\ S\ \{\psi\}$

- φ est la pré-condition (hypothèse).
  Intuitivement, c'est une assertion/formule qui fait intervenir les variables du programme.
- ψ est la post-condition (conclusion).
- S est le programme.

Sémantique : ψ doit être vraie après chaque exécution de S qui termine en partant d'une situation où φ est vraie.

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)`$^2\}$

Logique de Hoare

Définition : Axiome et Règles de preuve

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

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

Séquence
$$\frac{\{\psi_1\}\ S_1\ \{\psi_2\} \qquad \{\psi_2\}\ S_2\ \{\psi_3\}}{\{\psi_1\}\ S_1; S_2\ \{\psi_3\}}$$

Conditionnelle
$$\frac{\{c \wedge \varphi\}\ S_1\ \{\psi\} \qquad \{\neg c \wedge \varphi\}\ S_2\ \{\psi\}}{\{\varphi\}\ \texttt{si}\ c\ \texttt{alors}\ S_1\ \texttt{sinon}\ S_2\ \texttt{fsi}\ \{\psi\}}$$

Exemples :

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

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

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

Logique de Hoare

Définition : Axiome et Règles de preuve

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

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

Exemple :

Prouver l'algorithme "Mystère"

Théorème : Logique de Hoare

- Le système de preuve est correct (sound) :
  Si on peut prouver $\{\varphi\}\ S\ \{\psi\}$ dans ce système de preuve alors toute exécution de S partant d'un état vérifiant φ et qui termine, arrive dans un état vérifiant ψ (correction partielle).
- Sous certaines hypothèses, il est aussi complet.

Preuve 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` S $\{\psi\}$
Deviner un invariant I et prouver

Initialisation : $\varphi \Rightarrow I$

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

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

Remarque : ceci ne prouve pas la terminaison.

Plus faible pré-condition

Définition : Plus faible pré-condition (Dijkstra)

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

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

Règles de calcul

Affectation	$wp(x := e, \psi) = \psi[e/x]$
Séquence	$wp(S_1; S_2, \psi) = wp(S_1, wp(S_2, \psi))$
Conditionnelle	$wp(\texttt{si } c \texttt{ alors } S_1 \texttt{ sinon } S_2 \texttt{ fsi}, \psi) = (c \wedge wp(S_1, \psi)) \vee (\neg c \wedge wp(S_2, \psi))$

Exemples :

$$wp(s := s+t+2, s = (a+1)^2) = s+t+2 = (a+1)^2$$

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

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

Preuve d'algorithmes

Exemple : Recherche Dichotomique

```
fonction chercheDico (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 Inv1 ∧ Inv2 ∧ Inv3
      m <- (i+j) div 2
      si c ≤ t[m] alors j <- m sinon i <- m+1 fsi
   ftq
   retourner i
Fin

Inv1 : t[1..n] trié
Inv2 : 1 ≤ i ≤ j ≤ n
Inv3 : c ∈ t[i..j] ∨ c ∉ t[1..n]
```

Exercice : Exponentiation rapide

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

Preuve et récursivité

Exemple : Tri fusion

```
procédure triFusion (t:tableau[1..n]; i,j:entiers)
PRE : 1 ≤ i ≤ j ≤ n
POST : permute t[i..j] en un tableau trié
Début
   si i < j alors
      1 ≤ i < j ≤ n
      k <- (i+j) div 2
      1 ≤ i ≤ k < j ≤ n
      triFusion (t,i,k)
      1 ≤ i ≤ k < j ≤ n ∧ t[i..k] permutation triée de t'[i..k]
      triFusion (t,k+1,j)
      1 ≤ i ≤ k < j ≤ n ∧ t[i..k] permutation triée de t'[i..k]
                        ∧ t[k+1..j] permutation triée de t'[k+1..j]
      Fusion (t,i,j,k)
      t[i..j] permutation triée de t'[i..j]
   fsi
Fin
```

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

Preuve et récursivité

Exemple : Correction de Fusion

```
procédure Fusion (t:tableau[1..n]; i,j,k:entiers)
PRE : 1 ≤ i ≤ k < j ≤ n et t[i..k], t[k+1..j] triés
POST : permute t[i..j] en un tableau trié
Début
   s:tableau[i..j]
   a <- i; b <- k+1; c <- i
   Inv1:  s[i..c-1] permutation triée de t[i..a-1] ∪ t[k+1..b-1]
        ∧ s[i..c-1] ≤ t[a..k] ∪ t[b..j]
        ∧ i ≤ a ≤ k+1 ≤ b ≤ j+1 ∧ c = a + b-(k+1)
   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
   Inv2:  Inv1 ∧ (a > k ∨ b > j)
   tq a ≤ k faire s[c] <- t[a]; a++; c++ ftq
   Inv3:  Inv1 ∧ a > k
   tq b ≤ j faire s[c] <- t[b]; b++; c++ ftq
   Inv1 ∧ a > k ∧ b > j
   t[i..j] <- s[i..j]
Fin
```

Terminaison

Définition : Terminaison des boucles

Définir une fonction f à valeurs dans un ensemble muni d'une relation bien fondée, e.g. $(\mathbb{N}, <)$, et qui dépend des variables du programme.
Montrer que la valeur de f décroît strictement à chaque éxécution du corps de boucle

Exemples :

- Urne : nombre de boules dans l'urne
- Mystère : $\max(0, N - s + 1)$
- recherche dichotomique : $j - i$
- Fusion : $j + 1 - c$

Définition : Terminaison et récursivité

Définir une fonction f à valeurs dans un ensemble muni d'une relation bien fondée, e.g. $(\mathbb{N}, <)$, et qui dépend des paramètres de la fonction/procédure.
Montrer que la valeur de f décroît strictement à chaque appel récursif.

Exemple :

Tri fusion : $j - i$

Plan

Introduction

Preuve et terminaison

3 Complexité

Structures de données (types abstraits)

Structures de recherche (Dictionnaires)

Files de priorité

Paradigmes

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

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

Complexité au pire

Exemple : Partition dans le tri rapide

```
Procédure partition (t:tableau[1..n] de réels; i,j:entiers; k:entier)
PRE : 1 ≤ i < j ≤ n
POST : permute t[i..j] et retourne i≤k≤j tq t[i..k-1]≤t[k]<t[k+1..j]
Début
   échanger(t[i], t[(i+j) div 2]) // mieux si le tableau est déjà trié
   pivot <- t[i]; g <- i+1; d <- j
   Inv :  t[i..g-1] ≤ pivot < t[d+1..j] et pivot = t[i]
           et i < g ≤ j+1 et i ≤ d ≤ j
   tq g < d faire
      tq g ≤ j et t[g] ≤ pivot faire g++ ftq
      tq t[d] > pivot faire d-- ftq
      si g < d alors échanger(t[g],t[d]); g++; d-- finsi
   ftq
   si t[d] > pivot alors d-- fsi
   échanger(t[i],t[d])
   k <- d
Fin
```

Complexité au pire en nombre de comparaisons de clés : $\Theta(j-i)$.

Complexité au pire

Exemple : Complexité au pire du tri fusion

```
procédure tri-fusion (t:tableau[1..n] de réels; i,j:entiers)
PRE : 1 ≤ i ≤ j ≤ n
POST : permute t[i..j] en un tableau trié
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 $\Theta(j-i)$.
Complexité au pire du tri fusion satisfait la récurrence de partition :

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

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

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

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

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

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

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

est

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

Complexité au pire

Proposition : Borne inférieure

La complexité au pire en nombre de comparaisons de clés pour les tris par comparaisons est en $\Omega(n \log n)$.
Remarque : on verra un résultat plus fort pour la complexité en moyenne.

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 $\bar{i}^2$ (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.

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 :

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

Complexité en moyenne

Proposition : Borne inférieure

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

Exemple : Complexité en moyenne du tri rapide

```
procédure tri-rapide (t:tableau[1..n] de réels; i,j:entiers)
PRE : 1 ≤ i et j ≤ n
POST : permute t[i..j] en un tableau trié
Début
   si i < j alors
      partition (t,i,j,k)
      tri-rapide (t,i,k-1)
      tri-rapide (t,k+1,j)
   finsi
Fin
```

Complexité en moyenne : $\Theta(n \log n)$.

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.

On suppose que la fonction de partition 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 $k = \sigma(1)$. 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

$$|\{\sigma \in \mathfrak{S}_n \mid \sigma(1) = k \wedge \sigma^g = \rho\}| = \frac{(n-1)!}{(k-1)!}$$

et donc

$$p(\sigma^g = \rho \mid \sigma(1) = k) = \frac{1}{(k-1)!}$$

L'uniformité est préservée par la partition

Preuve : $|\{\sigma \in \mathfrak{S}_n \mid \sigma(1) = k \wedge \sigma^g = \rho\}| = \frac{(n-1)!}{(k-1)!}$

Fixons $S \subseteq \{2, \ldots, n\}$ avec $|S| = k-1$ et $T = \{2, \ldots, n\} \setminus S$.

On note $\mathfrak{S}_n^{k,S} = \{\sigma \in \mathfrak{S}_n \mid \sigma(1) = k \wedge S = \sigma^{-1}(\{1, \ldots, k-1\})\}$.

Si $\sigma \in \mathfrak{S}_n^{k,S}$, le pivot est k, $\sigma(S) = \{1, \ldots, k-1\}$ et $\sigma(T) = \{k+1, \ldots, n\}$.

Donc les résultats des comparaisons de clés durant la partition sont les mêmes pour tous $\sigma \in \mathfrak{S}_n^{k,S}$. Ils conduisent donc aux mêmes échanges. On en déduit qu'il existe une bijection $\tau : \{1, \ldots, k-1\} \to S$ donnant la provenance des éléments de σ^g:

$$\forall \sigma \in \mathfrak{S}_n^{k,S}, \quad \sigma^g = \sigma \circ \tau$$

On en déduit

$$\{\sigma \in \mathfrak{S}_n^{k,S} \mid \sigma^g = \rho\}| = (n-k)!$$

Puisque

$$\{\sigma \in \mathfrak{S}_n \mid \sigma(1) = k \wedge \sigma^g = \rho\} = \bigcup_{S \subseteq \{2,\ldots,n\} \mid |S|=k} \{\sigma \in \mathfrak{S}_n^{k,S} \mid \sigma^g = \rho\}$$

on obtient

$$|\{\sigma \in \mathfrak{S}_n \mid \sigma(1) = k \wedge \sigma^g = \rho\}| = \binom{n-1}{k-1} \times (n-k)! = \frac{(n-1)!}{(k-1)!}$$

$$p(\sigma^g = \rho \mid \sigma(1) = k) = \frac{p(\sigma^g = \rho \wedge \sigma(1) = k)}{p(\sigma(1) = k)} = \frac{\frac{1}{n!}\frac{(n-1)!}{(k-1)!}}{\frac{1}{n}} = \frac{1}{(n-1)!}$$

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

On suppose que la fonction partition compare chaque clé exactement une fois avec le pivot.

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

Variable aléatoire : $X_n : \mathfrak{S}_n \to \mathbb{N}$: nombre de comparaisons du tri rapide.

Décomposition : $X_n = \sum_{1 \leq i < j \leq n} X_n^{i,j}$

Linéarité : $E(X_n) = \sum_{1 \leq i < j \leq n} E(X_n^{i,j})$

Calcul : $E(X_n) \leq 2n \ln(n)$.

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

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

On suppose que la fonction partition fait au plus $n+1$ comparaisons sur un tableau de taille n.

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

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

Plan

Introduction

Preuve et terminaison

Complexité

4 Structures de données (types abstraits)

Structures de recherche (Dictionnaires)

Files de priorité

Paradigmes

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

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.
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) → booléen	
nonVide :	arbre(T) → booléen	
faireArbre :	T × arbre(T)^2 → arbre(T)	
racine :	arbre(T) → T	pré-condition : arbre non vide
fg :	arbre(T) → arbre(T)	pré-condition : arbre non vide
fd :	arbre(T) → arbre(T)	pré-condition : arbre non vide
créer :	→ arbre(T)	constructeur
détruire :	arbre(T) →	

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,nil,nil)
    op-un  : x <- expInfixe()
             retourner faireArbre(s,x,nil)
    (      : 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

Structures de données (types abstraits)

5 Structures de recherche (Dictionnaires)
- Type abstrait et implémentations naïves
- Arbres binaires de recherche
- Arbres équilibrés a–b
- Tables de hachage

Files de priorité

Paradigmes

Type abstrait dictionnaire

Définition : Dictionnaire

Soit Tclé un type ordonné et Tinfo un type arbitraire.
On définit le type dico(Tclé, Tinfo) par

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

estVide :	dico(Tclé,Tinfo) → booléen
nonVide :	dico(Tclé,Tinfo) → booléen
chercher :	dico(Tclé,Tinfo) × Tclé → Tinfo
insérer :	dico(Tclé,Tinfo) × Tclé × Tinfo → dico(Tclé,Tinfo)
supprimer :	dico(Tclé,Tinfo) × Tclé → dico(Tclé,Tinfo)
fusionner :	dico(Tclé,Tinfo)^2 → dico(Tclé,Tinfo)
scinder :	dico(Tclé,Tinfo) × Tclé → dico(Tclé,Tinfo)^2
créer :	→ dico(Tclé,Tinfo)
détruire :	dico(Tclé,Tinfo) →

On pourra retourner une information spéciale InfoVide lors d'une recherche infructueuse.

Donner plusieurs spécifications "naturelles" pour insérer ou fusionner.

Implémentation naïve 1

Tableau non trié

- Implémentation statique :(
- Complexité :

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

Implémentation naïve 2

Tableau trié

- Implémentation statique :(
- 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 $\mathcal{O}(\log(\log n))$.

Implémentation naïve 3

Liste chaînée triée

- Implémentation dynamique :)
- Complexité :

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

- Liste circulaire avec sentinelle.

Implémentation moins naïve

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.

Exemple : ABR

Insertions et suppressions dans un ABR.

Insertion dans un ABR

Exemple : Implémentation de l'insertion

```
Procédure insérerABR (a:arbre, c:Tclé, i:Tinfo)
Spec :  insertion du couple (c,i) dans l'ABR
Spec :  modifie l'information si la clé est déjà présente
Début
   si a.estVide alors
      a <- faireFeuille(c,i)
   sinon si c < a.clé alors
      insérerABR(a.fg,c,i)
   sinon si c > a.clé alors
      insérerABR(a.fd,c,i)
   sinon
      a.info <- i
   finsi
Fin
```

Exercice :

Implémenter les autres primitives, en particulier fusionner et scinder.

Arbres binaires de recherche

Exercice : ABR balisés

Dans un ABR balisé, les couples (clé,info) 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 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 équilibrés a–b

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

- Chaque nœud interne autre que la racine possède entre a et b fils.
- La racine, si ce n'est pas une feuille, possède entre 2 et b fils.
- Les nœuds internes ont une clé de moins que de fils.
- 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] < ··· < 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) :Tinfo
Spec :  retourne l'information associée à la clé dans l'arbre
Début
   si x.estVide() alors retourner infoVide finsi
   Trouver 1 ≤ i ≤ d(x) tel que x.c[i-1] < clé ≤ x.c[i]
   si clé = x.c[i] alors retourner x.info[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 $\log_a b \times \log_2 n$.

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

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 $\leq \dfrac{2a-1}{a} \leq 2$.

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.

Tables de hachage

Objectif : implémentation dynamique du type abstrait dictionnaire avec des opérations en $\mathcal{O}(1)$ en moyenne.

Définition : Table de hachage

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 de taille m et on vise une implémentation du type abtrait dictionnaire qui se rapproche de

```
d.chercher(clé)       return t[h(clé)]
d.insérer(clé,info)   t[h(clé)] <- info
```

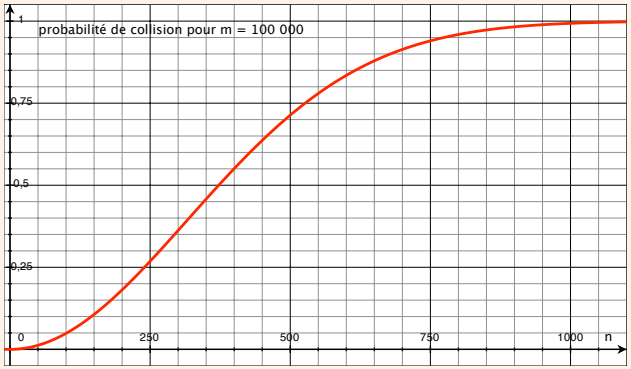
Il faut résoudre les collisions : $h(c_1) = h(c_2)$ avec $c_1 \neq c_2$.

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,info) tels que h(clé)=i.

Recherche

```
fonction chercher (t:Table, clé:Tclé) :Tinfo
Spec :  retourne l'information 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->info finsi
      p <- p->suiv
   ftq
   retourner infoVide
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 fixée)

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 fixée (mais arbitraire),
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 $n = \mathcal{O}(m)$ alors les opérations sont en moyenne en $\mathcal{O}(1)$.

Exercice :

La table étant fixée, 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.

Fonctions de hachage

On interprète les clés comme des entiers

$f : \Sigma^* \to \mathbb{N}$ où $f(u)$ est le nombre qui s'écrit u en base $|\Sigma|$

Exemple : $\Sigma = \text{ASCII} = \{0, \ldots, 127\}$ et pour $u \in \Sigma^*$

$$f(u_\ell \cdots u_0) = \sum_{j=0}^{\ell} u_j |\Sigma|^j.$$

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.

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 $x \geq 0$.

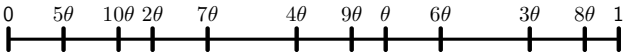
Théorème : Vera Turan Sos (1957)

Soit $\theta > 0$ un nombre irrationel. 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.

Fonctions de hachage

Définition : Hachage universel

On tronçonne la clé $k \in \mathbb{N}$ en r blocks de p bits avec $2^p \leq m$.
On choisit aléatoirement a_0, a_1, ..., a_{r-1} dans $[0, 2^p - 1]$.

$$h\left(\sum_{i=0}^{r-1} k_i \cdot (2^p)^i\right) = \left(\sum_{i=0}^{r-1} k_i \cdot a_i\right) \mod m$$

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) :Tinfo
Spec :  retourne l'information associée à la clé dans la table t
Début
   pour i <- 0 à m-1 faire
      j <- h(clé,i)
      si table[j].clé = clé alors retourner table[j].info finsi
      si table[j].clé = cléVide alors retourner infoVide finsi
   fpour
   retourner infoVide
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,info)` : $\alpha = \frac{n}{m} \leq 1$.
– : (rehashing) Lorsque le taux de remplissage s'approche de 1,
il faut augmenter la taille de la table et re-distribuer les clés.

Exercice : rehashing

Montrer que le coût amorti du rehashing est constant.

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

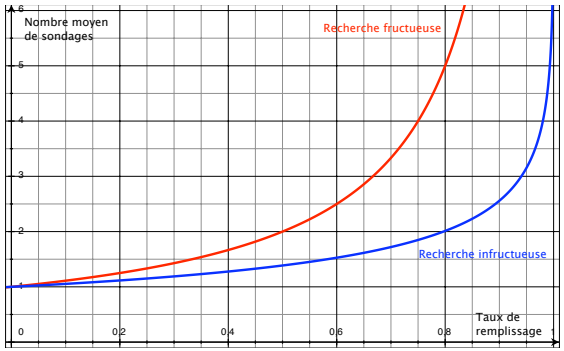
$\alpha = \dfrac{n}{m}$ où n est le nombre de clés dans la table.

Complexité de l'adressage ouvert

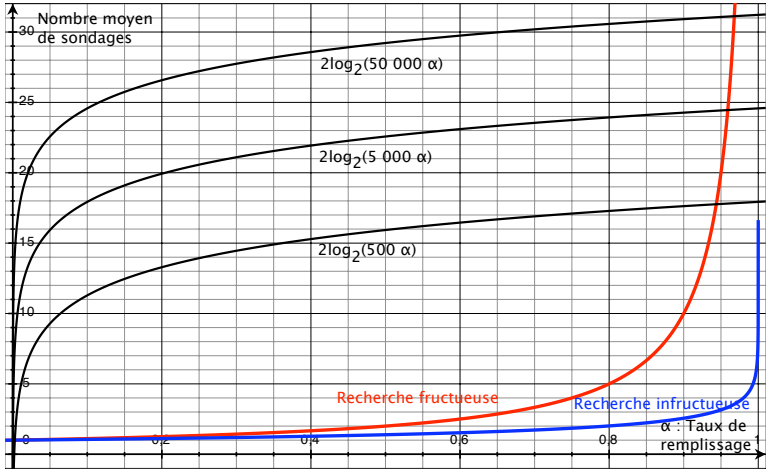
Proposition : Recherche

Si le hachage est uniforme, le nombre moyen de sondages

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



Hachage versus Arbres 2–4



Plan

Introduction

Preuve et terminaison

Complexité

Structures de données (types abstraits)

Structures de recherche (Dictionnaires)

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

Paradigmes

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 :

créer :		$\rightarrow$ FPrio(Tprio,Tinfo)
détruire :	FPrio(Tprio,Tinfo)	$\rightarrow$
estVide :	FPrio(Tprio,Tinfo)	$\rightarrow$ booléen
insérer :	FPrio(Tprio,Tinfo) $\times$ Tprio $\times$ Tinfo	$\rightarrow$ FPrio(Tprio,Tinfo)
minimum :	FPrio(Tprio,Tinfo)	$\rightarrow$ Tprio $\times$ Tinfo
supprimerMin :	FPrio(Tprio,Tinfo)	$\rightarrow$ FPrio(Tprio,Tinfo)
fusionner :	FPrio(Tprio,Tinfo)^2	$\rightarrow$ FPrio(Tprio,Tinfo)

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

diminuerPrio :	FPrio(Tprio,Tinfo) $\times$ Ptr $\times$ Tprio	$\rightarrow$ FPrio(Tprio,Tinfo)
augmenterPrio :	FPrio(Tprio,Tinfo) $\times$ Ptr $\times$ Tprio	$\rightarrow$ FPrio(Tprio,Tinfo)
supprimer :	FPrio(Tprio,Tinfo) $\times$ Ptr	$\rightarrow$ FPrio(Tprio,Tinfo)

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.t : tableau des priorités
F.n : nombre d'éléments dans la file de priorité

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.t[1] sinon prioVide)

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

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

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


File de priorité et tas binaire

Implémentation

```
Procédure supprimerMin ()
   si F.n > 0 alors
      F.t[1] <- F.t[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.t[p+1] < F.t[p] alors p++ fsi
      si F.t[p div 2] ≤ F.t[p] alors retourner fsi
      échanger(F.t[p div 2], F.t[p])
   fintq

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


File de priorité et tas binaire

Implémentation

```
Procédure supprimer (p:Int)
   si 1 ≤ p ≤ F.n alors
      F.t[p] <- F.t[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
$\Theta(\log n)$: diminuerPrio, insérer
$\Theta(\log n)$: augmenterPrio, supprimerMin
$\Theta(\log n)$: 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,
- 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

```
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; FusionArbres(a,b,r)
 sinon cas parmi
  r.degré < x.degré, y.degré : ajouter(z,t,r)
  r.degré = x.degré < y.degré : a <- x; x <- x.frère; FusionArbres(a,r,r)
  r.degré = y.degré < x.degré : a <- y; y <- y.frère; FusionArbres(a,r,r)
  r.degré = y.degré = x.degré : ajouter(z,t,r); a <- x; x <- x.frère;
                                b <- y; y <- y.frère; FusionArbres(a,b,r)
 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

```
Procédure FusionArbres(a,b,r)
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++
   r <- a; a <- NULL; b <- NULL

Procédure Ajouter(z,t,x)
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; x <- NULL
```


File de priorité et tas binomiaux

Complexité

- FusionArbres 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

Structures de données (types abstraits)

Structures de recherche (Dictionnaires)

Files de priorité

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

Gestion des partitions (Union-Find)

Bibliographie

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

[7] 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 : Faire l'appoint (ou rendre la monnaie)

Le problème est de faire l'appoint (rendre la monnaie) pour une somme donnée dans un système monétaire donné en minimisant le nombre de pièces.

- Données : les valeurs des pièces $1 = v_1 < v_2 < \cdots < v_k$ et la somme n.
- Solution : $n_1,\ldots,n_k$ tels que $n = n_1 v_1 + \cdots + n_k v_k$.
- Critère d'évaluation : minimiser $n_1 + \cdots + n_k$.

Variante : certaines pièces en nombre limité.

Problèmes d'optimisation

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 $x_1 w_1 + \cdots + x_n w_n \leq W$.
- Critère d'évaluation : maximiser $x_1 v_1 + \cdots + x_n v_n$.

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.

Exemple : Faire l'appoint

- Choix : une pièce à donner.
- Glouton : la pièce de plus grande valeur qui ne dépasse pas n :
  On cherche i maximal tel que $v_i \leq n$.
  Ce choix semble optimal.
- Sous-problème : $n - v_i$ avec le même système monétaire.

L'algorithme glouton ne donne pas forcément une solution globalement optimale :
Exemple : suite de valeurs 1, 4, 6

Algorithmes gloutons

Proposition : Faire l'appoint

L'algorithme glouton est optimal (i.e., donne toujours une solution optimale) si la suite des valeurs est : $1, p, p^2, \ldots, p^k$ avec $p \geq 2$ entier.

Exercice : Faire l'appoint

L'algorithme glouton est-il optimal dans les systèmes monétaires suivants ?

- $v_1 = 1$ et $v_i \geq 2v_{i-1}$ pour $1 < i \leq k$.
- 1, 2, 5, 10, 20, 50, 100. (Euros)
- 1, 5, 10, 25, 50, 100. (US)
- 1, $2\frac{1}{2}$, 5, 10, 20, 25, 50. (Portugais).
  Attention : n est toujours entier.
- 1, 3, 6, 12, 24, 30. (Anglais avant le système décimal)

Algorithmes gloutons

Quand est-ce que ça marche ?

On suppose que l'on sait déterminer 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.
- sous-structures optimales :
  Une solution optimale à un problème contient une solution optimale à un sous-problème.
  Contre-exemple : chemin simple de longueur maximale dans un graphe.
- Combinaison :
  Solution optimale au problème =
  choix glouton + solution optimale au sous-problème associé

Problème du sac à dos fractionnaire

Choix glouton

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

$W = 100$ et

Objets	1	2	3	4	5
Poids	10	20	30	40	50
valeur	20	30	66	40	60
$\frac{\text{valeur}}{\text{poids}}$	2	1,5	2,2	1	1,2

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.

Exercice :

L'algorithme glouton est-il optimal pour le problème du sac à dos si les objets ne sont pas fractionnables ?

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 $w : S \to]0, +\infty[$.

Problème

Trouver une partie indépendante de poids maximal.
Remarque : une partie indépendante de poids maximal est une partie indépendante maximale.

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.

Matroïdes et algorithmes gloutons

On peut aussi prouver l'optimalité de la façon suivante.

Théorème : Optimalité

L'algorithme glouton appliqué à un matroïde valué donne un ensemble indépendant de poids maximal.

1. Propriété du choix glouton : si x est l'élément de poids maximal dans S tel que $\{x\} \in I$, alors il existe $Y \in I$ de poids maximal tel que $x \in Y$.
2. Sous-structure optimale : soit x l'élément de poids maximal tel que $\{x\} \in I$. Trouver un ensemble indépendant de poids maximal contenant x pour M équivaut à trouver un ensemble indépendant de poids maximal pour le contracté $M_{|x} = (S', I')$ avec $S' = S \setminus \{x\}$ et $I' = \{Z \subseteq S' \mid Z \cup \{x\} \in I\}$.

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
- ...

Programmation dynamique

Définition : Programmation dynamique : cadre

- Une solution optimale s'obtient à partir des solutions optimales de plusieurs sous-problèmes de même nature.
- Les sous-problèmes ne sont pas indépendants.
- Il faut éviter de faire certains calculs plusieurs fois.

Exemple : Coefficients binomiaux

Si $0 < k < n$ alors $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$.

Solution : triangle de Pascal.

Programmation dynamique

Exemple : Faire l'appoint

- Le système monétaire $1 = v_1 < v_2 < \cdots < v_k$ est fixé.
- On veut calculer $f(n)$ le nombre minimal de pièces pour la somme n.
- Définition récursive (sous-structure optimale) :

$$f(n) = 1 + \min\{f(n - v_1), \ldots, f(n - v_k)\}$$

- Les sous-problèmes $n - v_1, \ldots, n - v_k$ ne sont pas indépendants.
- Solution itérative ascendante (bottom-up) :

```
f[0] <- 0
Pour j de 1 à n faire
   f[j] <- ∞
   pour i de 1 à k faire
      si j ≥ v[i] alors f[j] <- min(f[j], 1+f[j-v[i]]) fsi
   fpour
fpour
```

Complexité $\mathcal{O}(nk)$

- On peut retrouver la solution optimale à partir du tableau `f`.

Programmation dynamique

Principes

- Définir des sous-problèmes et montrer la propriété de sous-structure optimale : Une solution optimale contient des solutions optimales à ces sous-problèmes.
- Définir récursivement la solution.
- Calculer les valeurs optimales.
- Construire la solution optimale en utilisant les valeurs calculées.

Exemple : Faire l'appoint

- Sous-problème $g(i, j)$: nombre minimal de pièces pour la somme j en utilisant uniquement les pièces $v_1, \ldots, v_i$.
- Définition récursive (sous-structure optimale) :

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

Mémoïsation

Définition : Mémoïsation

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

Exemple : Faire l'appoint

- Solution récursive descendante :

```
Variables globales :
v[1..k] :  valeurs des pièces 1 = v[1] < ··· < v[k]
c[1..k,1..n] :  résultats déjà calculés, initialisés à ∞

fonction g(i,j) :  entier
   si c[i,j] < ∞ alors retourner c[i,j] fsi
   si i = 1 alors c[i,j] <- j; retourner c[i,j] fsi
   si j = v[i] alors c[i,j] <- 1; retourner c[i,j] fsi
   c[i,j] <- g[i-1,j]
   si j > v[i] alors c[i,j] <- min(c[i,j], 1 + g[i,j-v[i]]) fsi
   retourner c[i,j]
```

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

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.

Propriété de sous-structure optimale

- Propriété nécessaire aux deux techniques.
  Exemple : calcul d'un chemin de longueur minimale.
- Elle n'est pas toujours satisfaite.
  Exemple : calcul d'un chemin simple de longueur maximale.

Problème du sac à dos

Exemple : Sac à dos

- On suppose que les objets ne sont pas fractionnables.
- 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 :

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

- On en déduit facilement une solution itérative ou une solution récursive avec mémoïsation.

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 est le n-ième nombre de Catalan :

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

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 $1 \le i \le c$, $a[i] = k$ et $b[k] = i$ si l'élément k de t a été initialisé le i-ème.
- $t[k]$ initialisé $\iff$ $(1 \le b[k] \le c$ 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 x1,...,xk
   pour i <- i à k faire yi <- DR(xi)
   Combiner y1,...,yk pour obtenir la solution y au problème x
   retourner y
```

Exemple :

- Recherche dichotomique
- Tri fusion
- Tri rapide

Diviser pour régner

Exemple : Multiplication rapide de grands nombres

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 $\Theta\left(|a|^{\log_2 3} \cdot n^{\log_2 3}\right)$.

Diviser pour régner

Exemple : Multiplication rapide de matrices (Strassen 1969)

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

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

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

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

avec

$$\begin{array}{ll} D_1 = (A_{21} + A_{22} - A_{11})(B_{22} - B_{12} + B_{11}) & D_4 = (A_{11} - A_{21})(B_{22} - B_{12}) \\ D_2 = A_{11}B_{11} & D_5 = (A_{21} + A_{22})(B_{12} - B_{11}) \\ D_3 = A_{12}B_{21} & D_6 = (A_{12} - A_{21} + A_{11} - A_{22})B_{22} \\ & D_7 = A_{22}(B_{11} + B_{22} - B_{12} - B_{21}) \end{array}$$

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

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

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.

Plan

Introduction

Preuve et terminaison

Complexité

Structures de données (types abstraits)

Structures de recherche (Dictionnaires)

Files de priorité

Paradigmes

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 :

créer :	Int $\rightarrow$ Partition	constructeur
find :	Partition $\times$ Int $\rightarrow$ Int	
union :	Partition $\times$ Int $\times$ Int $\rightarrow$ Partition	
détruire :	Partition $\rightarrow$	

- `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
```

Autre application

Algorithme de Kruskal pour la recherche d'un arbre couvrant minimum dans un graphe.

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 créér(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 $\mathcal{O}(n + m\log_2 n)$.

Gestion de partitions

Union pondérée par rang [4]

`rang[1..N]` mémorise un majorant de la hauteur des arbres de racines `1..N`.

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

Proposition : Union pondérée par rang

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

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

$$A_0 : j \mapsto 2j$$
$$A_{i+1} : j \mapsto A_i^{(j)}(1) = \underbrace{A_i \circ \cdots \circ A_i}_{j \text{ fois}}(1)$$

En particulier, on a $A_1(j) = 2^j$ et $A_2(j) = \left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}j \text{ fois}$.

De plus, pour $i \geq 1$, on a $A_i(0) = 1$, $A_i(1) = 2$ et $A_i(2) = 4$.

On définit aussi l'inverse $\alpha : \mathbb{N}^2 \to \mathbb{N}$ par

$$\alpha(m,n) = \min\{k \geq 1 \mid A_k(4\lceil m/n\rceil) > \log_2 n\}.$$

Remarque : $A_2(4) = 65536$, donc si $m > 0$ et $0 < n < 2^{65536}$ alors $\alpha(m,n) \leq 2$.

Gestion de partitions

Théorème : Union par rang + compression des chemins — Tarjan

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

Théorème : Union par taille + compression des chemins — Tarjan

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

Corollaire : Le coût amorti est constant en pratique — $(n < 2^{65536})$