Algorithmique

Partiel du 17 novembre 2014

durée 2 heures 30

Le polycopié du cours est le seul document autorisé.

Les exercices sont indépendants.

Les procédures et fonctions devront être écrites en pseudo-code (pas en Java, Caml, ...)

Toutes les réponses devront être correctement justifiées. La rigueur des raisonnements, la clarté des explications, et la qualité de la présentation influeront sensiblement sur la note.

Le chiffre en regard d'une question est une indication sur sa difficulté ou sa longueur.

1 Preuve de Hoare

On considère le programme suivant ($t[1 \cdot\cdot n]$ est un tableau d'entiers) :

```
j ← n
tant que j > 0 faire
   i ← 1
   tant que i < j faire
      si t[i] > t[i+1] alors Echanger(t[i], t[i+1]) fsi
      i ← i + 1
   ftq
   j ← j − 1
ftq
```

[4] **a)** Montrer que la boucle interne (sur i) satisfait l'invariant : $t[1 \cdot\cdot i-1] \leq t[i]$.
Prouver avec la méthode de Hoare que ce programme réalise un tri du tableau t.
Donner bien tous les invariants utilisés pour votre preuve.

2 Complexité

On considère le programme suivant ($t[1 \cdot\cdot n]$ est un tableau d'entiers) :

```
pour i ← 1 à n − 1 faire
   si t[i] > t[i+1] alors Echanger(t[i], t[i+1]) fsi
fpour
```

Soit $X : \mathfrak{S}_n \to \mathbb{N}$ la variable aléatoire telle que $X(\sigma)$ est le nombre de fois où l'instruction `Echanger`$(-,-)$ est exécutée par le programme ci-dessus lorsque le tableau t est initialisé avec σ. L'objectif est de calculer $E(X)$.

[1] **a)** Soit $1 \leq k \leq n$. Montrer que $|A_k| = \frac{n!}{k}$ où $A_k = \{\sigma \in \mathfrak{S}_n \mid \sigma(k) = \max(\sigma(1), \ldots, \sigma(k))\}$.
La preuve est simple, ne vous lancez pas dans des calculs compliqués.

[3] **b)** Montrer que $E(X) = n - \sum_{k=1}^{n} \frac{1}{k}$.
Indication : pour tout $k \in \{1, \ldots, n-1\}$, calculer l'espérance de la variable aléatoire $X_k : \mathfrak{S}_n \to \{0, 1\}$ définie par $X_k(\sigma) = 1$ si et seulement si l'instruction `Echanger`$(t[k], t[k+1])$ est exécutée par le programme ci-dessus lorsque le tableau t est initialisé avec σ.

3 Relais pour téléphones mobiles

On considère une route *droite* sur laquelle se trouvent n maisons. On cherche à placer des relais pour téléphones mobiles le long de la route de telle sorte que chaque maison se trouve à une distance au plus K d'un relais. On veut bien sûr minimiser le nombre de relais utilisés.

Formellement, une donnée du problème est une suite de réels $x_1 < x_2 < \cdots < x_n$. Une solution est une suite $y_1, y_2, \ldots, y_m$ de réels telle que

$$\forall\ 1 \leq i \leq n,\ \exists\ 1 \leq j \leq m, \qquad |x_i - y_j| \leq K. \tag{1}$$

Une solution est optimale si elle utilise un nombre minimal de relais (m minimal).

[4] **a)** Proposer un algorithme pour trouver une solution optimale en temps $\mathcal{O}(n)$.
Montrer que la solution fournie par votre algorithme est correcte, i.e., vérifie (3).
Montrer que la solution fournie par votre algorithme est optimale.

4 Multiplication de matrices booléennes

Dans ce problème, la numérotation des lignes et colonnes d'une matrice commence à 0. Si B est une matrice $k \times n$, on note $B[i, j]$ le coefficient i, j de la matrice B (avec $0 \leq i < k$ et $0 \leq\ j < n$). Une matrice est booléenne si ses coefficients sont 0 ou 1.

Soit $k > 0$ un entier positif. On note $S^{(k)}$ la matrice booléenne $2^k \times k$ telle que pour tout $a \in \{0, \ldots, 2^k - 1\}$, la ligne a de $S^{(k)}$ est l'écriture binaire de a en commençant par le bit de poids faible : si $S^{(k)}[a, j] = a_j$ pour $0 \leq j < k$ alors $a = \overline{a_{k-1} \cdots a_1 a_0}^2$. Par exemple,

$$A^{(2)} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}$$

[4] **a)** Soit B une matrice $k \times n$ arbitraire.
Pour $k > 1$, donner une décomposition en blocs de $S^{(k)}$ qui fasse intervenir $S^{(k-1)}$.
Exprimer le produit $C = S^{(k)} \times B$ par blocs.
Montrer que l'on peut calculer le produit $C = S^{(k)} \times B$ en temps $\mathcal{O}(2^k n)$.
Écrire l'algorithme *itératif* qui réalise ce calcul.

[2] **b)** Soit A une matrice $m \times k$ *booléenne* et soit B une matrice $k \times n$ arbitraire. Montrer que l'on peut calculer la matrice $A \times B$ en temps $\mathcal{O}(mk + mn + 2^k n)$.

[1] **c)** Soit A une matrice $m \times (\ell k)$ *booléenne* et soit B une matrice $(\ell k) \times n$ arbitraire. Montrer que l'on peut calculer la matrice $A \times B$ en temps $\mathcal{O}(\ell(mk + mn + 2^k n))$.

[1] **d)** Soit A une matrice $n \times n$ *booléenne* et soit B une matrice $n \times n$ arbitraire. Montrer que l'on peut calculer la matrice $A \times B$ en temps $\mathcal{O}(\frac{n^3}{\log_2 n})$.

Algorithmique

Partiel du 13 novembre 2013
durée 2 heures

Le polycopié du cours est le seul document autorisé.
Les exercices sont indépendants.
Les procédures et fonctions devront être écrites en pseudo-code (pas en Java, Caml, ...)
Toutes les réponses devront être correctement justifiées.
Le chiffre en regard d'une question est une indication sur sa difficulté ou sa longueur.

1 Partitions

On définit une variante *nommée* du type abstrait `partition` comme suit :

- Donnée : une partition P de $\{1, \ldots, N\}$, chaque classe $X \in P$ étant désignée par un entier de $\{1, \ldots, M\}$, *son nom*.
  Attention : le nom d'une classe n'est pas nécessairement un élément de la classe et deux classes distinctes ne doivent pas avoir le même nom.
- Opérations :

$$\begin{aligned} \texttt{créer} &: \text{Int} \times \text{Int} \to \texttt{Partition} \\ \texttt{find} &: \texttt{Partition} \times \text{Int} \to \text{Int} \\ \texttt{union} &: \texttt{Partition} \times \text{Int} \times \text{Int} \times \text{Int} \to \texttt{Partition} \\ \texttt{renommer} &: \texttt{Partition} \times \text{Int} \times \text{Int} \to \texttt{Partition} \end{aligned}$$

- `Partition(N,M) P` crée une partition P composée des singletons de $\{1, \ldots, N\}$ et dont l'ensemble de noms est $\{1, \ldots, M\}$.
  Il faut $N \leq M$ et initialement, le nom du singleton $\{i\}$ est i.
- `P.find(i)` retourne le nom de la classe de P qui contient i.
- `P.union(x,y,z)` fusionne les classes de noms x et y en une classe de nom z.
  Ne fait rien s'il n'y a pas de classe de nom x ou pas de classe de nom y ou s'il y a déjà une classe de nom z et que $z \notin \{x, y\}$.
- `P.renommer(x,y)` change le nom de la classe x en y.
  Ne fait rien s'il n'y a pas de classe de nom x ou s'il y a déjà une classe de nom y.

[5] **a)** Décrire une structure de données concrète pour implémenter très efficacement le type abstrait `partition` et donner les algorithmes pour les 4 opérations sur ce type. Les opérations `union` et `renommer` devront s'exécuter en temps constant. L'opération `créer` devra être en temps $\mathcal{O}(N+M)$. Le coût d'une suite d'opérations comportant $n-1$ `unions` et m `find` sera en $\mathcal{O}((n+m)(1+\alpha(n)))$ où $\alpha(n)$ est l'inverse de la fonction d'Ackerman vue en cours.
Remarque : on peut bien sûr utiliser les théorèmes du cours.

On souhaite maintenant résoudre le problème des minima hors-ligne. La donnée est une suite σ d'entiers 2 à 2 distincts et d'instructions `extraire-min`, notées E dans la suite. Chaque préfixe de σ doit contenir au moins autant d'entiers que d'instructions E.
Par exemple : $\sigma = 5, 4, 8, 2, E, 7, E, 1, 6, E, E, 3$.
Chaque instruction E doit afficher le plus petit entier qui précède et qui n'a pas déjà été affiché. Pour l'exemple ci-dessus on affichera 2, 4, 1, 5.
On cherche à résoudre ce problème avec un algorithme *hors-ligne*, i.e., on peut lire *toute* la donnée σ *avant* de commencer à afficher le résultat.

[5] **b)** Donner un algorithme pour résoudre le problème des minima hors-ligne lorsque les entiers de la suite σ forment une permutation de $\{1, \ldots, n\}$. L'algorithme devra s'exécuter en temps $\mathcal{O}(n\alpha(n))$.
Indication : on pourra commencer par créer une partition dont la partie de nom $i \geq 1$ contient les entiers qui se trouvent entre le $(i-1)$-ème E et le i-ème E. Avec l'exemple ci-dessus, nous aurons les parties $X_1 = \{5, 4, 8, 2\}$, $X_2 = \{7\}$, $X_3 = \{1, 6\}$, $X_4 = \emptyset$, et $X_5 = \{3\}$.

2 Recherche de motifs

Dans ce problème, un texte t de n caractères est représenté par un tableau $t[1 \cdot\cdot n]$ de caractères. Pour $i, j \geq 0$, on note $t[i \cdot\cdot j]$ le facteur de t allant de la position i à la position j. Par convention, si $j = 0$ ou $i > n$ ou $j < i$ alors $t[i \cdot\cdot j] = \varepsilon$ est le mot vide.

Un *bord* de t est un préfixe *strict* de t qui est aussi suffixe de t : un bord est donc déterminé par un entier $0 \leq i < n$ tel que $t[1 \cdot\cdot i] = t[n - i + 1 \cdot\cdot n]$.

Le texte $t[1 \cdot\cdot n]$ étant fixé, on définit la fonction $\mathsf{lbm}\colon [1 \cdot\cdot n] \to [0 \cdot\cdot n-1]$ par $\mathsf{lbm}(k)$ est la longueur du bord maximal de $t[1 \cdot\cdot k]$. On a donc $\mathsf{lbm}(k) < k$.

On note $\mathsf{lbm}^+(k) = \{\mathsf{lbm}(k), \mathsf{lbm}^{(2)}(k), \ldots, \mathsf{lbm}^{(i)}(k) = 0\}$ l'ensemble des entiers obtenus en itérant l'application lbm jusqu'à obtenir 0.

[1] **a)** Donner les valeurs de la fonction lbm pour le texte *abaababbababaaba*. Donner l'ensemble $\mathsf{lbm}^+(16)$.

On admet (provisoirement) que la fonction lbm satisfait

$$\forall\, 1 \leq k < n,\ \mathsf{lbm}(k+1) = \begin{cases} 0 & \text{si } t[k+1] \neq t[j+1],\ \text{pour tout } j \in \mathsf{lbm}^+(k) \\ j+1 & \text{si } j = \max\{\ell \in \mathsf{lbm}^+(k) \mid t[k+1] = t[\ell+1]\} \end{cases} \tag{2}$$

On considère le programme suivant :

```
PRE   : t tableau [1 .. n] de caractères
POST  : B[i] = lbm[i] pour tout 1 ≤ i ≤ n
B[1] ← 0; k ← 1
tant que k < n faire
   Inv1: ∀ 1 ≤ i ≤ k, B[i] = lbm(i)
   j ← B[k];
   tant que j > 0 et t[k+1] ≠ t[j+1] faire
      Inv2: j ∈ lbm⁺(k) et ∀ ℓ ∈ lbm⁺(k), j < ℓ ⟹ t[k+1] ≠ t[ℓ+1]
      j ← B[j]
   ftq
   si t[k+1] = t[j+1] alors j ← j+1 fsi
   B[k+1] ← j
   k ← k+1
ftq
```

[2] **b)** Montrer que le programme s'exécute en temps $\mathcal{O}(n)$.

[5] **c)** Prouver que `Inv1` et `Inv2` sont bien des invariants des boucles `tant que`. En déduire que le programme satisfait sa spécification.

[2] **d)** Donner un algorithme qui affiche toutes les occurrences d'un motif $x[1 \cdot\cdot p]$ contenues dans un texte $y[1 \cdot\cdot q]$. L'algorithme devra s'exécuter en temps $\mathcal{O}(p+q)$. Une occurrence de x dans y est déterminée par un entier i tel que $x[1 \cdot\cdot p] = y[i \cdot\cdot i+p-1]$.
Indication : considérer une lettre # qui n'apparaît ni dans le motif x ni dans le texte y, et exécuter le programme ci-dessus sur le texte $t = x \cdot \# \cdot y$ obtenu par concaténation du motif x, de la lettre # et du texte y.

[3] **e)** Prouver la propriété (2) de la fonction lbm (propriété admise ci-dessus).

Algorithmique

Partiel du 15 novembre 2012

durée 3 heures

Le polycopié du cours est le seul document autorisé.

Les exercices sont indépendants.

Les procédures et fonctions devront être écrites en pseudo-code (pas en Java, Caml, ...)

Toutes les réponses devront être correctement justifiées. La rigueur des raisonnements, la clarté des explications, et la qualité de la présentation influeront sensiblement sur la note.

Le chiffre en regard d'une question est une indication sur sa difficulté ou sa longueur.

1 Preuve et terminaison

On considère le programme suivant (toutes les variables sont entières) :

```
Lire a et b
p ← 3; d ← 1; m ← 1
tant que a > 1 ou b > 1 faire
   cas
      p divise a et p divise b: a := a/p; b ← b/p; d ← d · p; m ← m · p
      p divise a et p ne divise pas b: a ← a/p; m ← m · p
      p ne divise pas a et p divise b: b ← b/p; m ← m · p
      p ne divise pas a et p ne divise pas b: p ← p + 2
   fincas
ftq
Afficher d et m
```

On suppose que les données a et b sont deux entiers **impairs positifs**.

[5] **a)** Qu'affiche ce programme sur les données $a = 45$ et $b = 75$?
Montrer que ce programme termine.
Que calcule ce programme ?
Prouver avec la méthode de Hoare que ce programme calcule bien cela.

2 Complexité

On considère le programme suivant ($t[1..n]$ est un tableau d'entiers strictement positifs) :

```
m ← 0; j ← 0;
pour i ← 1 à n faire
   si t[i] > m alors m ← t[i]; j ← i fsi
fpour
```

Pour un entier $k \in \{1, \ldots, n\}$, on note $X_k \colon \mathfrak{S}_n \to \{0, 1\}$ la variable aléatoire définie par $X_k(\sigma) = 1$ si et seulement si l'instruction $m \leftarrow t[k]$ est exécutée par le programme ci-dessus lorsque le tableau t est initialisé avec σ.

[2] **a)** Montrer que $E(X_k) = \frac{1}{k}$.

Soit $X \colon \mathfrak{S}_n \to \{1, \ldots, n\}$ la variable aléatoire telle que $X(\sigma)$ est le nombre de fois que l'instruction $m \leftarrow t[i]$ est exécutée par le programme ci-dessus lorsque le tableau t est initialisé avec σ.

[2] **b)** Montrer que $E(X) \leq 1 + \ln(n)$.

3 Alignement de séquences

Il s'agit d'aligner *au mieux* deux mots. L'application principale est l'alignement de séquences du génome, représentées par des mots sur l'alphabet $\{A, T, C, G\}$ pour l'ADN ou $\{A, U, C, G\}$ pour l'ARN. Les mots à aligner sont très longs : entre 10^3 et 10^6 lettres pour un gène.

Un autre exemple d'application est la correction automatique d'orthographe. Si on tape "ocurrance", un système intelligent (smartphone/éditeur [1]) proposera "occurrence".

Considérons deux mots $x = x_1 \cdots x_m$ et $y = y_1 \cdots y_n$ sur un alphabet Σ fixé. Un alignement est une suite de couples $(i_1, j_1)(i_2, j_2) \cdots (i_k, j_k)$ telle que $1 \leq i_1 < i_2 < \cdots < i_k \leq m$ et $1 \leq j_1 < j_2 < \cdots < j_k \leq n$. Par exemple, $A = (2,1)(3,2)(4,3)(5,4)(6,5)$ et $B = (2,1)(3,2)(5,4)(6,5)$ sont deux alignements des mots $x =$ `global` et $y =$ `local` que l'on représente par

$$A = \begin{pmatrix} g & l & o & b & a & l \\ - & l & o & c & a & l \end{pmatrix} \quad \text{et} \quad B = \begin{pmatrix} g & l & o & b & - & a & l \\ - & l & o & - & c & a & l \end{pmatrix}.$$

La qualité d'un alignement est mesurée par un *coût* utilisant un paramètre $\delta > 0$ pour les "trous" (lettres non alignées) et des paramètres $\alpha(a, b) \geq 0$ pour les lettres alignées ($a, b \in \Sigma$). Le coût d'un alignement $C = (i_1, j_1)(i_2, j_2) \cdots (i_k, j_k)$ de deux mots $x = x_1 \cdots x_m$ et $y = y_1 \cdots y_n$ est

$$\texttt{coût}(C) = (m + n - 2k)\delta + \sum_{1 \leq \ell \leq k} \alpha(x_{i_\ell}, y_{j_\ell}).$$

En général, on suppose que $\alpha(a, a) = 0$ pour $a \in \Sigma$ (mais ce n'est pas nécessaire). Avec cette hypothèse, on obtient $\texttt{coût}(A) = \delta + \alpha(b, c)$ et $\texttt{coût}(B) = 3\delta$ pour l'exemple ci-dessus. L'alignement A est meilleur que B si $\alpha(b, c) \leq 2\delta$.

Dans la suite, les deux mots $x = x_1 \cdots x_m$ et $y = y_1 \cdots y_n$ sont fixés. On cherche un alignement optimal (i.e., ayant un coût minimal) de x et y.

1. dommage que ce ne soit pas encore disponible sur les stylos :)

Pour $0 \leq i \leq m$ et $0 \leq j \leq n$, on note $\texttt{opt-g}(i,j)$ le coût d'un alignement optimal de $x_1 \cdots x_i$ et de $y_1 \cdots y_j$.

[3] **a)** Combien vaut $\texttt{opt-g}(0,j)$ pour $0 \leq\ j \leq n$.
Donner un algorithme récursif pour calculer en temps $\mathcal{O}(mn)$ et en espace $\mathcal{O}(mn)$ toutes les valeurs $\texttt{opt-g}(i,j)$ pour $0 \leq i \leq m$ et $0 \leq j \leq n$.
Donner un algorithme qui, en supposant déjà calculées les valeurs ci-dessus, affiche un alignement optimal de x et y en temps $\mathcal{O}(m+n)$.

[2] **b)** Le problème majeur de l'algorithme ci-dessus est sa complexité en espace. Pour aligner deux séquences du génome ayant chacune environ 10^5 lettres, il faudrait 10 Go de mémoire. On veut donc un algorithme qui fonctionne en espace $\mathcal{O}(m+n)$.
Donner un algorithme itératif pour calculer le coût d'un alignement optimal pour x et y en temps $\mathcal{O}(mn)$ et en espace $\mathcal{O}(m+n)$.
Expliquer pourquoi cet algorithme ne permet plus de reconstruire et d'afficher un alignement optimal.

Pour $1 \leq i \leq m+1$ et $1 \leq j \leq n+1$, on note $\texttt{opt-d}(i,j)$ le coût d'un alignement optimal de $x_i \cdots x_m$ et de $y_j \cdots y_n$.

[2] **c)** Pour un entier $1 \leq \ell < m$ fixé, montrer que

$$\texttt{opt-g}(m,n) = \min\{\texttt{opt-g}(\ell,j) + \texttt{opt-d}(\ell+1,j+1) \mid 0 \leq j \leq n\}\,.$$

[2] **d)** Pour un entier $1 \leq \ell < m$ fixé, donner un algorithme pour calculer toutes les valeurs $(\texttt{opt-g}(\ell,j))_{0\leq j\leq n}$ et $(\texttt{opt-d}(\ell+1,j+1))_{0\leq j\leq n}$ en temps $\mathcal{O}(mn)$ et en espace $\mathcal{O}(m+n)$.

[4] **e)** Donner un algorithme récursif qui affiche un alignement optimal pour x et y en temps $\mathcal{O}(mn)$ et en espace $\mathcal{O}(m+n)$. Justifier la complexité en temps et la complexité en espace de votre algorithme.

On considère maintenant un mot $x = x_1 \cdots x_m$ et un ensemble $V = \{v_1, \ldots, v_k\}$ où chaque v_i est un mot de longueur au plus n (en général m est beaucoup plus grand que n). Il s'agit d'aligner x de façon optimale avec une concaténation (à déterminer) de mots de V. Le coût optimal est maintenant

$$\texttt{opt-star}(x,V) = \min\{\texttt{opt-g}(x,y) \mid y \in V^*\}$$

où V^* est l'ensemble des concaténations de mots de V, par exemple $v_2v_4v_1v_2v_7 \in V^*$ (il n'est pas nécessaire d'utiliser tous les mots de V et chaque mot de V peut être utilisé plusieurs fois).

[4] **f)** Donner un algorithme qui calcule $\texttt{opt-star}(x,V)$ et affiche un alignement optimal correspondant en temps polynomial. Préciser la complexité de votre algorithme.

Algorithmique

Partiel du 10 novembre 2011

durée 3 heures

Le polycopié du cours est le seul document autorisé.

Les exercices sont indépendants.

Les procédures et fonctions devront être écrites en pseudo-code (pas en Java, Caml, ...)

Toutes les réponses devront être correctement justifiées. La rigueur des raisonnements, la clarté des explications, et la qualité de la présentation influeront sensiblement sur la note.

Le chiffre en regard d'une question est une indication sur sa difficulté ou sa longueur.

1 Belle impression

Le problème consiste à effectuer une belle impression d'un paragraphe par une imprimante. L'entrée est un texte de n mots de longueur $\ell_1, \ell_2, ..., \ell_n$. Le but est d'imprimer ce paragraphe de manière équilibrée sur un nombre de lignes à calculer qui contiendront au plus m caractères chacune. On suppose $0 < \ell_i \leq m$ pour tout $1 \leq i \leq n$.

Si une ligne contient les mots i à j et qu'on laisse exactement un espace entre deux mots, le nombre d'espaces nécessaires à la fin de la ligne est

$$m - \ell_i - \sum_{k=i+1}^{j} 1 + \ell_k$$

qui doit être positif ou nul pour que les mots tiennent sur la ligne. Une impression est valide si chaque ligne vérifie cette condition.

Le coût (ou équilibre) d'une impression valide est la somme des cubes des nombres d'espaces nécessaires à la fin des lignes, hormis la dernière.

L'objectif est de fournir une impression valide de coût minimal.

[1] **a)** Est-ce que l'algorithme glouton consistant à remplir les lignes une à une en mettant à chaque fois le maximum de mots possible sur la ligne en cours, fournit l'optimum ?

[4] **b)** L'objectif est de donner un algorithme de programmation dynamique pour résoudre ce problème en temps $\mathcal{O}(mn)$.

1. Définir les sous-problèmes et montrer la propriété de sous-structure optimale.
2. Écrire une fonction qui calcule le coût minimal du problème en temps $\mathcal{O}(mn)$.
   On mémorisera dans un tableau `C` les coûts optimaux des sous-problèmes.
3. En utilisant le tableau `C` ci-dessus, écrire une procédure qui imprime le texte de façon optimale en temps $\mathcal{O}(n)$.
   On considère que l'impression d'un mot se fait en temps $\mathcal{O}(1)$.

On suppose maintenant que la fonction de coût à minimiser est la somme des nombres d'espaces nécessaires à la fin des lignes, hormis la dernière.

[3] **c)** Écrire une procédure qui imprime le texte de façon optimale en temps $\mathcal{O}(n)$.

2 Arbres d'intervalles

Un *arbre d'intervalles* est un arbre binaire de recherche (ABR) tel que chaque nœud a contient 3 réels : $a.x \leq a.y \leq a.z$. On note $a.g$ et $a.d$ les sous-arbres gauche et droit du nœud a. L'arbre vide est noté `Null`.

L'intervalle (fermé) représenté par le nœud a est $[a.x, a.y]$ et la clé servant à ordonner l'ABR est $a.x$. Donc $a.g \neq$ `Null` implique $a.g.x \leq a.x$ et $a.d \neq$ `Null` implique $a.x \leq a.d.x$ (noter qu'il peut y avoir égalité et qu'il est possible qu'un même intervalle $[x', y']$ soit représenté par plusieurs nœuds d'un arbre d'intervalles).

Le troisième réel $a.z$ est le maximum des bornes supérieures des intervalles contenus dans le sous-arbre de racine a. En particulier, si a est une feuille alors $a.z = a.y$ et si les sous-arbres $a.g$ et $a.d$ sont non vides alors $a.z = \max(a.y, a.g.z, a.d.z)$.

On s'intéresse à la fonction de recherche suivante :

```
Procédure recherche (a:arbre; x', y':réels; b:arbre)
PRE : a est un arbre d'intervalles et x' ≤ y'
POST : (b ≠ Null et [x', y'] ∩ [b.x, b.y] ≠ ∅) ou
       (b = Null et [x', y'] ∩ [c.x, c.y] = ∅ pour tout nœud c de a)
Début
   b := a
   tant que b ≠ Null et [x', y'] ∩ [b.x, b.y] = ∅ faire
      si b.g ≠ Null et x' ≤ b.g.z alors b := b.g sinon b := b.d fsi
   ftq
Fin
```

Soit b un nœud d'un arbre d'intervalles et soient $x' \leq y'$ deux réels. On définit la formule

$$\varphi(b, x', y') ::= \exists c \text{ nœud du sous-arbre } b \text{ tel que } [x', y'] \cap [c.x, c.y] \neq \emptyset$$

[3] **a)** Montrer que le triplet de Hoare suivant est valide :

$$\{\varphi(b, x', y') \wedge [x', y'] \cap [b.x, b.y] = \emptyset\}$$
$$\texttt{si } b.g \neq \texttt{Null et } x' \leq b.g.z \texttt{ alors } b := b.g \texttt{ sinon } b := b.d \texttt{ fsi}$$
$$\{\varphi(b, x', y'\}$$

[2] **b)** Prouver à l'aide d'un invariant que la procédure `recherche` ci-dessus est correcte.

[1] **c)** Écrire une procédure pour insérer un intervalle $[x', y']$ dans un arbre d'intervalles a avec une complexité au pire $\mathcal{O}(h(a))$ où $h(a)$ est la hauteur de l'arbre a. Il faudra bien sûr maintenir les propriétés d'un arbre d'intervalles, mais il n'est pas nécessaire d'éviter que l'intervalle $[x', y']$ ne figure plusieurs fois dans l'arbre.

3 Ordonnancement de tâches

On considère un ensemble $S = \{1, \ldots, n\}$ de tâches, chacune nécessitant une unité de temps pour son exécution. Pour chaque tâche $i \in S$, on donne une date limite $d[i] \in \mathbb{N}\setminus\{0\}$ de fin d'exécution et une pénalité $p[i] > 0$ en cas de non respect de cette date limite. On suppose que les tâches sont exécutées à partir de l'instant 0.

Un ordonnancement est une permutation $\sigma \in \mathfrak{S}_n$ qui définit l'ordre $\sigma(1), \sigma(2), \ldots, \sigma(n)$ d'exécution des tâches. La date de fin d'exécution de la tâche i est donc $\sigma(i)$ et sa date limite est respectée si $\sigma(i) \leq d[i]$.

Le coût de l'ordonnancement $\sigma \in \mathfrak{S}_n$ est la somme des pénalités associées aux tâches dont la date limite n'est pas respectée. On cherche bien sûr à trouver un ordonnancement de coût minimal.

Un sous-ensemble $A \subseteq S$ de tâches est *réalisable* s'il existe un ordonnancement de A sans pénalité, i.e., si on peut ordonner les éléments de A en une suite $a_1, \ldots, a_k$ telle que $i \leq d[a_i]$ pour tout $1 \leq i \leq k$. On note I l'ensemble des parties réalisables de S.

[2] **a)** Soit $A \subseteq S$ et $a_1, \ldots, a_k$ un ordonnancement de A tel que $d[a_1] \leq d[a_2] \leq \cdots \leq d[a_k]$. Montrer que $A \in I$ si et seulement si $i \leq d[a_i]$ pour tout $1 \leq i \leq k$.

On décide de représenter une partie $A \in I$ par sa taille $k = |A|$ et un tableau a (de dimension n) qui contient les éléments de A rangés par dates limites croissantes : $A = \{a[1], \ldots, a[k]\}$ et $d[a[1]] \leq \cdots \leq d[a[k]]$.

[3] **b)** Écrire une fonction qui réalise la spécification ci-dessous en temps $\mathcal{O}(k)$.
Justifier la correction et la complexité de votre fonction.

```
Fonction testeAjoute (a:tableau; k,i:entiers) : booléen
PRE : k ≥ 0, A = {a[1],...,a[k]} ∈ I, d[a[1]] ≤ ··· ≤ d[a[k]], i ∈ S \ A
POST : retourne FAUX si A ∪ {i} ∉ I et dans ce cas a et k sont inchangés
       retourne VRAI si A ∪ {i} ∈ I et dans ce cas a et k sont modifiés
       pour qu'ils codent A ∪ {i}
```

[5] **c)** Montrer que (S, I) est un matroïde.

[3] **d)** En déduire un algorithme efficace pour trouver un ordonnancement de coût minimal. Donner la complexité (au pire) de votre algorithme.

Algorithmique

Partiel du 10 novembre 2010

durée 2 heures 30

Le polycopié du cours est le seul document autorisé.

Les exercices sont indépendants.

Toutes les réponses devront être correctement justifiées. La rigueur des raisonnements, la clarté des explications, et la qualité de la présentation influeront sensiblement sur la note.

Le chiffre en regard d'une question est une indication sur sa difficulté ou sa longueur.

1 Partition du 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
   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[j])
   k <- g
Fin
```

[4] **a)** Prouver à l'aide d'un invariant que la procédure `partition` ci-dessus est correcte.

Attention : On demande une preuve de Hoare. Vous devez donc définir un invariant pour la boucle `tant que`, insérer des assertions entre les lignes du programme, montrer que chaque triplet de Hoare $\{\varphi\}\ S\ \{\psi\}$ est correct (où φ et ψ sont les assertions avant et après l'instruction S), ...

2 Multiplication de matrices booléennes

Dans ce problème, la numérotation des lignes et colonnes d'une matrice commence à 0. Si B est une matrice $k \times n$, on note $B[i,j]$ le coefficient i,j de la matrice B (avec $0 \leq i < k$ et $0 \leq\ j < n$). Une matrice est booléenne si ses coefficients sont 0 ou 1.

Soient $k > 0$ un entier positif. On note $S^{(k)}$ la matrice booléenne $2^k \times k$ telle que pour tout $a \in \{0, \ldots, 2^{k-1}\}$ d'écriture binaire $a = \overline{a_{k-1} \cdots a_1 a_0}^2$, la ligne a de $S^{(k)}$ est $(a_0, a_1, \ldots, a_{k-1})$, i.e., $S^{(k)}[a,j] = a_j$ pour $0 \leq j < k$.

[2] **a)** Soit B une matrice $k \times n$ arbitraire. Montrer que l'on peut calculer la matrice $S^{(k)} \cdot B$ en temps $\mathcal{O}(2^k n)$.

[2] **b)** Soit A une matrice $m \times k$ *booléenne* et soit B une matrice $k \times n$ arbitraire. Montrer que l'on peut calculer la matrice $A \cdot B$ en temps $\mathcal{O}(mk + mn + 2^k n)$.

[1] **c)** Soit A une matrice $m \times (\ell k)$ *booléenne* et soit B une matrice $(\ell k) \times n$ arbitraire. Montrer que l'on peut calculer la matrice $A \cdot B$ en temps $\mathcal{O}(\ell(mk + mn + 2^k n))$.

[1] **d)** Soit A une matrice $n \times n$ *booléenne* et soit B une matrice $n \times n$ arbitraire. Montrer que l'on peut calculer la matrice $A \cdot B$ en temps $\mathcal{O}(\frac{n^3}{\log_2 n})$.

3 Ensemble

On définit un type abstrait T par :

- Donnée : un sous-ensemble X de $\{1, \ldots, N\}$.
- Opérations :

$$\begin{aligned} \texttt{créer} &: \quad \text{Int} \to \text{T} \\ \texttt{prendre} &: \quad \text{T} \to \text{T} \times \text{Int} \\ \texttt{mettre} &: \text{T} \times \text{Int} \to \text{T} \end{aligned}$$

- `T(N) X` crée le sous-ensemble $X = \emptyset$ de $\{1, \ldots, N\}$.
- `X.prendre()` supprime un élément arbitraire de X et le retourne.
  L'ensemble X doit être non vide.
- `X.mettre(i)` ajoute l'élément i à l'ensemble X.
  On *doit* avoir $i \in \{1, \ldots, N\}$ et on *peut* avoir $i \in X$.

[2] **a)** Donner une implémentation complète de ce type abstrait pour laquelle les opérations `prendre` et `mettre` sont réalisées en temps constant.
Attention à bien gérer le fait qu'on peut appeler `X.mettre(i)` alors que $i \in X$.

4 Équivalence pour les automates

Soit $\mathcal{A} = (Q, \Sigma, \delta, F)$ un automate déterministe complet avec Q un ensemble fini d'états, Σ un ensemble fini de lettres (l'alphabet), $\delta : Q \times \Sigma \to Q$ la fonction de transitions, et $F \subseteq Q$ l'ensemble des états acceptants (on ne précise pas d'état initial). La fonction δ est étendue aux mots par récurrence en posant pour tout $q \in Q$:

$$\delta(q, \varepsilon) = q \text{ (ε dénote le mot vide)} \quad \text{et} \quad \delta(q, va) = \delta(\delta(q, v), a) \text{ pour } v \in \Sigma^* \text{ et } \alpha \in \Sigma.$$

Si $q \in Q$, on note $\mathcal{L}(\mathcal{A}, q) = \{v \in \Sigma^* \mid \delta(q, v) \in F\}$ l'ensemble des mots acceptés par $\mathcal{A}$ en prenant q comme état initial. Deux états $p, q \in Q$ engendrent le même langage si $\mathcal{L}(\mathcal{A}, p) = \mathcal{L}(\mathcal{A}, q)$, on dit alors que p et q sont Nerode-équivalents.

On admettra (dans un premier temps) que deux états $p, q \in Q$ sont Nerode-équivalents si et seulement si il existe une relation d'équivalence $\sim$ sur Q telle que $p \sim q$ et pour tous $p', q' \in Q$ tels que $p' \sim q'$, on a

$$p' \in F \iff q' \in F \quad \text{et} \quad \delta(p', a) \sim \delta(q', a) \text{ pour tout } a \in \Sigma.$$

On suppose dans la suite que $Q = \{1, \ldots, n\}$. On considère l'algorithme ci-dessous qui utilise principalement une *partition* `P` de l'ensemble Q et un ensemble `E` de couples d'éléments de Q représenté par une *file*.

```
fonction testEquiv (p,q:entiers) : booléen
PRE : p,q ∈ Q et p ≠ q
Début
   Partition P.créer(n)
   File E.créer()
   E.enfiler((p,q))
   tant que E.nonVide() faire
      (x,y) <- E.défiler()
      si non (x ∈ F ⟺ y ∈ F) alors retourner Faux fsi
      x' <- P.find(x); y' <- P.find(y)
      si x' ≠ y' alors
         P.union(x',y')
         pour tout a ∈ Σ faire E.enfiler(δ(x,a),δ(y,a)) fpour
      fsi
   ftq
   retourner Vrai
Fin
```

On considère la propriété (Inv1) :

$$\forall (x,y) \in E,\ \exists v \in \Sigma^* \text{ tel que } \{x,y\} = \{\delta(p,v), \delta(q,v)\}.$$

[2] **a)** Montrer que (Inv1) est un invariant de la fonction `testEquiv`.
Remarque : Il n'est pas demandé ici de faire une preuve de Hoare.

[1] **b)** Montrer que si la fonction `testEquiv` retourne `Faux` alors p et q ne sont pas Nerode-équivalents.

[1] **c)** Montrer que la boucle `tant que` de la fonction `testEquiv` termine.

[1] **d)** Montrer que la complexité en temps de la fonction `testEquiv` est au pire *presque* linéaire, plus précisément, $\mathcal{O}(mn\alpha(n,n))$ où $m = |\Sigma|$ est la taille de l'alphabet et α est *l'inverse de la fonction d'Ackermann* définie en cours.
Remarque : la taille de $\mathcal{A}$ est au moins mn puisqu'il faut représenter la fonction δ.

On note $\sim_P$ la relation d'équivalence définie par la partition P : $x \sim_P y$ si x et y sont dans une même classe de P. On note $\sim_E$ la plus petite relation d'équivalence contenant les couples de E. On note $\sim_{P,E}$ la plus petite relation d'équivalence contenant $\sim_P$ et $\sim_E$. Par exemple, si $x_0 \sim_P x_1 \sim_E x_2 \sim_P x_3$ alors $x_0 \sim_{P,E} x_3$.

On définit la propriété (Inv2) :

$$\forall (i,j) \in Q^2 \text{ tel que } i \sim_P j, \text{ on a } \begin{cases} i \in F \iff j \in F \\ \text{et } \delta(i,a) \sim_{P,E} \delta(j,a) \text{ pour tout } a \in \Sigma \end{cases}$$

[3] **e)** Montrer que (Inv2) est un invariant de la boucle `tant que` (si on n'exécute pas l'instruction `retourner Faux`).
Remarque : Il n'est pas demandé ici de faire une preuve de Hoare.

[1] **f)** Montrer que si la fonction `testEquiv` retourne `Vrai` alors p et q sont Nerode-équivalents.

[Bonus] **g)** Montrer que la caractérisation admise en début d'exercice pour "p et q sont Nerode-équivalents" est correcte.
Question subsidiaire à traiter seulement si vous avez fait tout le reste, ce qui m'étonnerait. ;)

Algorithmique

Partiel du 12 novembre 2009

durée 3 heures

Le polycopié du cours est le seul document autorisé.

Les exercices sont indépendants.

Toutes les réponses devront être correctement justifiées. La rigueur des raisonnements, la clarté des explications, et la qualité de la présentation influeront sensiblement sur la note.

Le chiffre en regard d'une question est une indication sur sa difficulté ou sa longueur.

1 Complexité

[3] **a)** Donner un algorithme pour trouver le minimum et le maximum de $n \geq 2$ valeurs contenues dans un tableau t en faisant au maximum $\lceil \frac{3}{2}n - 2 \rceil$ comparaisons.

On considère un tableau t contenant n entiers (positifs ou négatifs) et vérifiant $t[1] < t[2] < \cdots < t[n]$.
On souhaite trouver, pourvu qu'il existe, un indice $1 \leq i \leq n$ tel que $t[i] = i$.

[2] **b)** Donner un algorithme pour résoudre ce problème avec une complexité au pire en $\mathcal{O}(\log n)$.

[2] **c)** Montrer qu'on ne peut pas faire mieux en utilisant uniquement des comparaisons entre des éléments du tableau et des entiers.

2 Ensemble

On définit un type abstrait T par :

- Donnée : un sous-ensemble X de $\{1, \ldots, N\}$.
- Opérations :

$$\begin{aligned} \texttt{créer} &: \quad \text{Int} \to \text{T} \\ \texttt{prendre} &: \quad \text{T} \to \text{T} \times \text{Int} \\ \texttt{mettre} &: \text{T} \times \text{Int} \to \text{T} \end{aligned}$$

- `T(N) X` crée le sous-ensemble $X = \emptyset$ de $\{1, \ldots, N\}$.
- `X.prendre()` supprime un élément arbitraire de X et le retourne.
  L'ensemble X doit être non vide.
- `X.mettre(i)` ajoute l'élément i à l'ensemble X.
  On *doit* avoir $i \in \{1, \ldots, N\}$ et on *peut* avoir $i \in X$.

[3] **a)** Donner une implémentation complète de ce type abstrait pour laquelle les opérations `prendre` et `mettre` sont réalisées en temps constant.
Attention à bien gérer le fait qu'on peut appeler `X.mettre(i)` alors que $i \in X$.

3 Partitions

On définit une variante du type abstrait `partition` comme suit :

- Donnée : une partition P de $\{1, \ldots, N\}$, chaque classe $X \in P$ étant désignée par un entier de $\{1, \ldots, M\}$, *son nom*.
  Attention : le nom d'une classe n'est pas nécessairement un élément de la classe et deux classes distinctes ne peuvent avoir le même nom.
- Opérations :

$$\begin{aligned} \texttt{créer} &: \text{Int} \times \text{Int} \to \texttt{Partition} \\ \texttt{find} &: \texttt{Partition} \times \text{Int} \to \text{Int} \\ \texttt{union} &: \texttt{Partition} \times \text{Int} \times \text{Int} \to \texttt{Partition} \\ \texttt{renommer} &: \texttt{Partition} \times \text{Int} \times \text{Int} \to \texttt{Partition} \end{aligned}$$

- `Partition(N,M) P` crée une partition P composée des singletons de $\{1, \ldots, N\}$ et dont l'ensemble de noms est $\{1, \ldots, M\}$.
  Il faut $N \leq M$ et initialement, le nom du singleton $\{i\}$ est i.
- `P.find(i)` retourne le nom de la classe de P qui contient i.
- `P.union(x,y,z)` fusionne les classes de noms x et y en une classe de nom z.
  Ne fait rien s'il n'y a pas de classe de nom x ou pas de classe de nom y ou s'il y a déjà une classe de nom z et que $z \neq x$ et $z \neq y$.
- `P.renommer(x,y)` change le nom de la classe x en y.
  Ne fait rien s'il n'y a pas de classe de nom x ou s'il y a déjà une classe de nom y.

[5] **a)** Décrire une structure de données concrète pour implémenter très efficacement le type abstrait `partition` et donner les algorithmes pour les 4 opérations sur ce type. Les opérations `union` et `renommer` devront s'exécuter en temps constant. L'opération `créer` devra être en temps $\mathcal{O}(N+M)$. Le coût d'une suite d'opérations comportant $n-1$ `unions` et m `find` sera en $\mathcal{O}(n + m\alpha(n))$ où $\alpha(n)$ est l'inverse de la fonction d'Ackerman vue en cours.
Remarque : on peut bien sûr utiliser les théorèmes du cours.

On souhaite maintenant résoudre le problème des minima hors-ligne. La donnée est une suite d'entiers 2 à 2 distincts et d'instructions `extraire-min`, notées E dans la suite. Par exemple : $\sigma = 5, 4, 8, 2, E, 7, E, 1, 6, E, E, 3$.
Chaque instruction E doit afficher le plus petit entier qui précède et qui n'a pas déjà été affiché. Pour l'exemple ci-dessus on affichera 2, 4, 1, 5.
On cherche à résoudre ce problème avec un algorithme *hors-ligne*, i.e., on peut lire *toute* la donnée σ *avant* de commencer à afficher le résultat.

[5] **b)** Donner un algorithme pour résoudre le problème des minima hors-ligne lorsque les entiers de la suite σ forment une permutation de $\{1, \ldots, N\}$ et que σ contient au plus N instructions `extraire-min` (E). L'algorithme devra s'exécuter en temps $\mathcal{O}(n\alpha(n))$.
Indication : on pourra commencer par créer une partition dont la partie de nom $i \geq 1$ contient les entiers qui se trouvent entre le $(i-1)$-ème E et le i-ème E. Avec l'exemple ci-dessus, nous aurons les parties $X_1 = \{5, 4, 8, 2\}$, $X_2 = \{7\}$, $X_3 = \{1, 6\}$, $X_4 = \emptyset$, et $X_5 = \{3\}$.

Une question supplémentaire pour ceux qui s'ennuieraient.

[5] **c)** Montrer que la complexité d'une suite de n unions par taille *suivie* d'une suite de m find avec compression des chemins est au pire en $\mathcal{O}(n + m)$.
Remarque : il est important pour cette question de faire *toutes* les unions *avant* les find.

Algorithmique

Partiel du 13 novembre 2008

durée 3 heures

Le polycopié du cours est le seul document autorisé.

Les exercices sont indépendants.

Toutes les réponses devront être correctement justifiées. La rigueur des raisonnements, la clarté des explications, et la qualité de la présentation influeront sensiblement sur la note.

1 Terminaison

On considère la fonction de McCarthy :

```
fonction f(n:int) : int
Début
   si n > 100 alors retourner n-10 fsi
   retourner f(f(n+11))
Fin
```

a) Montrer que la fonction termine pour toute donnée entière et exprimer *directement* la valeur $f(n)$ en fonction de n.

On note $g(n)$ le nombre d'additions et de soustractions utilisées lors du calcul de $f(n)$ par le programme ci-dessus.

b) Calculer $g(98)$, $g(91)$ et $g(76)$.

c) Exprimer *directement* $g(n)$ en fonction de n.

2 Complexité

On souhaite déterminer la résistance à la chute de bocaux. Plus précisément, on souhaite déterminer la hauteur maximale de laquelle on peut faire tomber un bocal sans qu'il se casse. Pour cela, on dispose d'une échelle dont les échelons sont équirépartis et on cherche à déterminer l'entier n tel que si on lâche un bocal de l'échelon n il se casse alors qu'il ne se casse pas si on le lâche de l'échelon $n-1$.

Si on dispose d'un seul bocal, on peut clairement déterminer l'entier n en faisant n expériences : on lâche successivement le bocal des échelons 1, 2, 3, ... et lorsqu'il se casse on a déterminé n.

a) On dispose maintenant de 2 bocaux. Donner un algorithme A_2 qui permet de déterminer l'entier n avec un nombre d'expériences $f_2(n)$ sous-linéaire, i.e., $\lim_{n\to+\infty} \frac{f_2(n)}{n} = 0$.

b) Pour $k > 2$, donner un algorithme A_k qui permet de déterminer l'entier n en utilisant au plus k bocaux et en faisant un nombre d'expériences $f_k(n)$ tel que $\lim_{n\to+\infty} \frac{f_k(n)}{f_{k-1}(n)} = 0$.

3 Relais pour téléphones mobiles

On considère une route *droite* sur laquelle se trouvent n maisons. On cherche à placer des relais pour téléphones mobiles le long de la route de telle sorte que chaque maison se trouve à une distance au plus K d'un relais. On veut bien sûr minimiser le nombre de relais utilisés.

Formellement, une donnée du problème est une suite de réels $x_1 < x_2 < \cdots < x_n$. Une solution est une suite $y_1, y_2, \ldots, y_m$ de réels telle que

$$\forall\ 1 \leq i \leq n,\ \exists\ 1 \leq j \leq m, \qquad |x_i - y_j| \leq K. \tag{3}$$

L'objectif est de trouver une solution optimale en temps linéaire.

Bien lire toutes les questions avant de commencer.

a) Proposer un algorithme pour résoudre le problème et montrer qu'il fonctionne en temps $\mathcal{O}(n)$.

b) Prouver avec la méthode de Hoare que les solutions fournies par votre algorithme sont correctes, i.e., qu'elles vérifient (3).

c) Prouver que les solutions fournies par votre algorithme sont optimales, i.e., qu'elles utilisent un nombre minimal de relais.

4 Sélection optimale de tâches

Un travailleur indépendant doit choisir les tâches qu'il va réaliser pour chacune des semaines à venir. Chaque semaine, on lui propose une tâche particulièrement éprouvante qu'il ne peut réaliser que si la semaine précédente il était en vacances ; et une tâche facile qu'il peut réaliser quoi qu'il ait fait la semaine précédente. Chaque semaine il peut réaliser au plus une tâche et bien sûr, la tâche éprouvante est mieux rémunérée que la facile. Il va chercher à maximiser son revenu.

Formellement, la donnée du problème est une suite de couples $(l_1, h_1), (l_2, h_2), \ldots, (l_n, h_n)$ décrivant pour chaque semaine $i \in \{1, \ldots, n\}$ la rémunération l_i pour la tâche facile et la rémunération h_i pour la tâche éprouvante. On suppose $l_i \leq h_i$ pour tout $1 \leq i \leq n$.

Une solution est une suite de choix $c_1, c_2, \ldots, c_n$ avec $c_i \in \{v, l, h\}$ pour chaque $1 \leq i \leq n$ et telle que si $1 < i \leq n$ et $c_i = h$ alors $c_{i-1} = v$.

a) Donner la solution optimale et le revenu associé pour la suite de couples

i	1	2	3	4
l_i	5	2	12	7
h_i	9	8	31	17

b) Même question pour la suite de couples

i	1	2	3	4	5	6	7
l_i	5	2	12	7	9	3	7
h_i	9	8	31	17	15	14	13

c) Donner un algorithme qui calcule en temps linéaire la solution optimale.

5 Points les plus proches

On se donne n points $P_1, \ldots, P_n$ 2 à 2 distincts dans un plan et on note (x_i, y_i) les coordonnées du point P_i. On note $d(P, Q)$ la distance euclidienne entre les deux points P et Q du plan.

Le problème est de déterminer un couple de points (P_i, P_j) avec $i \neq j$ dont la distance est minimale, i.e., $d(P_i, P_j) \leq d(P_k, P_\ell)$ pour tous $1 \leq k, \ell \leq n$ avec $k \neq \ell$.

On peut facilement résoudre ce problème en temps $\mathcal{O}(n^2)$. L'objectif est de le résoudre en temps $\mathcal{O}(n \log n)$.

a) Soit $E \subseteq \{1, \ldots, n\}$ les indices d'un sous-ensemble de points et soit $m = |E|$. Soit A la suite des éléments de E triée par abscisses croissantes. On peut voir A comme un tableau tel que $E = \{A[1], \ldots, A[m]\}$ et $x_{A[1]} \leq \cdots \leq x_{A[m]}$. De même, soit B un tableau représentant la suite des éléments de E triée par ordonnées croissantes.

Étant donnés les tableaux A et B, montrer que l'on peut construire, *en temps* $\mathcal{O}(m)$, des tableaux B_g et B_d tels que, en notant $k = \lfloor \frac{m}{2} \rfloor$ on ait

— $\{B_g[1], \ldots, B_g[k]\} = \{A[1], \ldots, A[k]\}$,
— $\{B_d[1], \ldots, B_d[m-k]\} = \{A[k+1], \ldots, A[m]\}$,
— et les tableaux B_g et B_d sont triés par ordonnées croissantes :

$$y_{B_g[1]} \leq \cdots \leq y_{B_g[k]}$$
$$y_{B_d[1]} \leq \cdots \leq y_{B_d[m-k]}$$

b) On suppose que 3 points P_1, P_2 et P_3 sont dans une bande verticale de largeur δ, que la distance minimale entre ces points est au moins δ et qu'ils sont triés par ordonnées croissantes, i.e.,

— $|x_i - x_j| \leq \delta$ pour tous $1 \leq i, j \leq 3$,
— $d(P_i, P_j) \geq \delta$ si $i \neq j$, et
— $y_1 \leq y_2 \leq y_3$.

Montrer que $y_3 - y_1 \geq \delta \frac{\sqrt{3}}{2}$.

c) Soit B un tableau représentant les points $P_1, \ldots, P_n$ triés par ordonnées croissantes : $\{B[1], \ldots, B[n]\} = \{1, \ldots, n\}$ et $y_{B[1]} \leq \cdots \leq y_{B[n]}$. Soit $L \in \mathbb{R}$ et $\delta > 0$. On suppose $\{1, \ldots, n\}$ partitionné en deux ensembles E et F tels que $L - \delta \leq x_i \leq L \leq x_j \leq L + \delta$ pour tous $i \in E$ et $j \in F$. On suppose de plus que $d(P_i, P_j) \geq \delta$ pour tous $(i, j) \in E^2 \cup F^2$ tels que $i \neq j$. Montrer que l'on peut déterminer en temps $\mathcal{O}(n)$ s'il existe un couple de points vérifiant $0 < d(P_i, P_j) < \delta$ et dans l'affirmative calculer, toujours en temps linéaire, un tel couple de distance minimale.

d) Montrer que l'on peut résoudre en temps $\mathcal{O}(n \log n)$ le problème initial, i.e., trouver un couple de points (P_i, P_j) vérifiant $0 < d(P_i, P_j) \leq d(P_k, P_\ell)$ pour tous $1 \leq k, \ell \leq n$ avec $k \neq \ell$.

Algorithmique

Partiel du 15 novembre 2007
durée 3 heures

Les notes de cours et de TD sont autorisées, mais pas les livres.

Les exercices sont indépendants.

Toutes les réponses devront être correctement justifiées. La rigueur des raisonnements, la clarté des explications, et la qualité de la présentation influeront sensiblement sur la note.

1 Preuve de programme

On considère la fonction :

```
fonction f(a:int, b:int) : (int, int)
HYP : a ≥ b > 0
SPEC : ???
   r <- a mod b
   si r = 0 alors retourner (1, 1 - (a div b)) fsi
   (c,d) <- f(b,r)
   retourner (d, c - d · (a div b))
```

a) Quel est le résultat de $f(39, 15)$?

b) Donner la spécification formelle de cette fonction.

c) Prouver avec la méthode de Hoare que cette fonction satisfait sa spécification.

d) Prouver que cette fonction termine toujours.

2 Arbres optimaux

On considère un ensemble de n clés $c_1 < c_2 < \cdots < c_n$. La clé c_i a la probabilité p_i d'être tirée, avec bien sûr $p_1 + \cdots + p_n = 1$. On veut construire un arbre binaire de recherche (ABR) qui contient ces n clés et qui optimise le temps moyen de recherche. Le temps de recherche d'une clé dans un ABR est le nombre de nœuds visités lors de cette recherche.

Soit t un ABR contenant les n clés. On note $\text{prof}_t(c_i)$ la profondeur du nœud de t qui contient la clé c_i. La profondeur de la racine est 1. Le temps moyen de recherche (coût) pour l'arbre t est donc :

$$\text{coût}(t) = \sum_{i=1}^{n} p_i \cdot \text{prof}_t(c_i).$$

Un arbre optimal pour $[p_1, \ldots, p_n]$ est un ABR de coût minimal.

a) Pour $n = 3$, donner tous les ABR possibles et pour chacun d'eux le coût en fonction de p_1, p_2 et p_3. Application numérique : Calculer les coûts des ABR pour $[6, 3, 4]$.

b) Un ABR de hauteur minimale est-il nécessairement optimal ? Prouver que oui ou donner un contre-exemple.

c) On considère l'algorithme (glouton) suivant : on trie les clés par probabilités décroissantes et on construit un ABR, à partir de l'arbre vide, par insertions successives des clés par probabilités décroissantes (on utilise la procédure d'insertion usuelle, i.e., aux feuilles).
Donner un exemple dans lequel l'arbre obtenu par cet algorithme n'est pas de hauteur minimale mais dont le coût est inférieur aux coûts des arbres de hauteur minimale.
Cet algorithme donne-t-il nécessairement un arbre optimal ? Prouver que oui ou donner un contre-exemple.

Dans ce qui précède, nous avons uniquement considéré les recherches fructueuses. En général, il faut aussi considérer les recherches infructueuses. On note toujours p_i la probabilité de tirer la clé c_i. Pour $0 \leq i \leq n$, notons q_i la probabilité de tirer une clé dans l'intervalle $]c_i, c_{i+1}[$ (avec la convention $c_0 = -\infty$ et $c_{n+1} = +\infty$). Nous avons maintenant

$$p_1 + \cdots + p_n + q_0 + \cdots + q_n = 1.$$

Pour $0 \leq i \leq n$, on note d_i une clé factice prise dans l'intervalle $]c_i, c_{i+1}[$. La clé d_i représente une recherche infructueuse dans l'intervalle associé $]c_i, c_{i+1}[$.

Soit t un ABR dont les nœuds internes contiennent les n clés $c_1, \ldots, c_n$ et dont les feuilles contiennent les $n+1$ clés factices $d_0, \ldots, d_n$. Le coût de la recherche de c_i dans t est toujours $\text{prof}_t(c_i)$. Le coût d'une recherche infructueuse dans l'intervalle $]c_i, c_{i+1}[$ est $\text{prof}_t(d_i) - 1$. Le -1 est dû au fait que l'arbre réel ne contient pas la clé factice d_i et que le nombre de nœuds visités lors de cette recherche infructueuse est donc la profondeur du père de d_i dans t. Finalement, le coût de l'arbre t est maintenant :

$$\text{coût}(t) = \sum_{i=1}^{n} p_i \cdot \text{prof}_t(c_i) + \sum_{i=0}^{n} q_i \cdot (\text{prof}_t(d_i) - 1)$$

et un arbre optimal pour $[p_1, \ldots, p_n \mid q_0, \ldots, q_n]$ est un ABR de coût minimal.

Dans cette définition, il est inutile de supposer que les p_i et q_i soient des probabilités, i.e., que $p_1 + \cdots + p_n + q_0 + \cdots + q_n = 1$. Dans la suite, on abandonne donc cette restriction et on parlera de *poids* au lieu de *probabilités* et on supposera les poids positifs ou nuls.

d) Est-ce qu'un arbre optimal pour $[p_1, \ldots, p_n \mid q_0, \ldots, q_n]$ est aussi un arbre optimal pour $[q_0, p_1, q_1, \ldots, p_n, q_n]$, i.e., si on considère $d_0, \ldots, d_n$ comme de vraies clés ? Prouver que oui ou donner un contre-exemple.

e) Montrer qu'il y a un unique arbre optimal pour $[1, \ldots, 1 \mid 0, \ldots, 0]$ lorsque $n = 2^k - 1$ $(k > 0)$ et calculer son coût.

Dans la suite, on fixe les poids $[p_1, \ldots, p_n \mid q_0, \ldots, q_n]$. Pour $0 \leq i \leq j \leq n$, on note $T(i, j)$ l'ensemble des ABR dont les nœuds internes contiennent les clés $c_{i+1}, \ldots, c_j$ et dont les feuilles contiennent les clés factices $d_i, \ldots, d_j$. On remarque que $T(i, i)$ est réduit à l'arbre contenant uniquement la clé factice d_i. On note aussi $\text{Opt}(i, j)$ l'ensemble des arbres optimaux pour $[p_{i+1}, \ldots, p_j \mid q_i, \ldots, q_j]$, i.e., l'ensemble des arbres de coût minimal dans $T(i, j)$.

f) On suppose $0 \leq i < j \leq n$. Soit $t \in \mathrm{Opt}(i,j)$ un arbre de racine c_k et de sous-arbres gauche et droit t_g et t_d. Montrer que $t_g \in \mathrm{Opt}(i,k-1)$ et $t_d \in \mathrm{Opt}(k,j)$.

Pour $0 \leq i \leq j \leq n$, on note $w(i,j)$ le poids total $p_{i+1} + \cdots + p_j + q_i + \cdots + q_j$, en particulier $w(i,i) = q_i$. Pour $0 \leq i \leq j \leq n$, on note $C(i,j)$ le coût optimal pour les poids $[p_{i+1}, \ldots, p_j \mid q_i, \ldots, q_j]$, i.e., le coût d'un arbre quelconque de $\mathrm{Opt}(i,j)$, en particulier $C(i,i) = 0$.

g) Prouver que pour $i < j$ on a

$$C(i,j) = w(i,j) + \min_{i<k\leq j} C(i,k-1) + C(k,j).$$

h) En déduire un algorithme permettant de calculer le tableau C des coûts optimaux en temps $\mathcal{O}(n^3)$. Prouver que votre algorithme réalise bien cette complexité.

i) Écrire une fonction `arbreOpt` ayant deux paramètres entiers telle que `arbreOpt`(i,j) retourne un arbre de $\mathrm{Opt}(i,j)$.

La suite du problème vise à calculer les coûts optimaux en temps quadratique. Lorsque $0 \leq i < j \leq n$, on note $R(i,j)$ l'ensemble des entiers k tels que c_k est racine d'un arbre de $\mathrm{Opt}(i,j)$. Soient A et B deux ensembles d'entiers. On note $A \leq B$ si $\forall a \in A$, $\forall b \in B$, $b < a$ implique $b \in A$ et $a \in B$. Pour $1 < s \leq n$, on considère la propriété :

$$\forall 1 \leq i+1 < j \leq n, \quad j - i \leq s \Longrightarrow R(i,j-1) \leq R(i,j) \leq R(i+1,j) \qquad P(s)$$

j) Montrer $P(2)$.

Dans la suite, on suppose que $P(n)$ est vraie (la preuve de ce résultat n'est pas simple). Pour $0 \leq i < j \leq n$, on note $r(i,j) = \min R(i,j)$, i.e., $r(i,j)$ correspond à la racine minimale d'un arbre optimal pour $[p_{i+1}, \ldots, p_j \mid q_i, \ldots, q_j]$.

k) Pour $1 \leq i+1 < j \leq n$, montrer que $r(i,j)$ est l'entier k qui réalise le minimum de $C(i,k-1) + C(k,j)$ avec $r(i,j-1) \leq k \leq r(i+1,j)$.

l) Soit $1 < s \leq n$. Montrer que

$$\sum_{i=0}^{n-s} 1 + r(i+1,i+s) - r(i,i+s-1) \leq 2n.$$

m) Donner un algorithme *quadratique* qui calcule le tableau C des coûts optimaux et le tableau r des racines minimales des arbres optimaux. Prouver que l'algorithme est correct et qu'il est bien quadratique.

Quelques questions supplémentaires pour que personne ne s'ennuie.

n) Caractériser les arbres optimaux pour $[1, \ldots, 1 \mid 0, \ldots, 0]$ lorsque $n > 0$ est arbitraire et calculer le coût optimal.

o) Donner un arbre optimal pour $[5, 1, 1, \ldots, 1 \mid 0, \ldots, 0]$ lorsque $n = 40$ et calculer le coût optimal.

p) Montrer $P(3)$.

q) Montrer $P(n)$.

Algorithmique

Magistère STIC

Partiel du 16 novembre 2006
durée 3 heures

Les notes de cours et de TD sont autorisées, mais pas les livres.

Les exercices sont indépendants.

Toutes les réponses devront être correctement justifiées. La rigueur des raisonnements, la clarté des explications, et la qualité de la présentation influeront sensiblement sur la note.

1 Arbres AVL (Adelson-Velskii et Landis, 1962)

Un AVL est un arbre binaire de recherche (ABR) tel que pour chaque nœud de l'arbre, la différence de hauteur entre le sous-arbre gauche et le sous-arbre droit est d'au plus 1.

a) Soit n le nombre de nœuds et h la hauteur d'un AVL (la hauteur d'un arbre réduit à sa racine est 1). Montrer que $F_{h+2} - 1 \leq n \leq 2^h - 1$ où F_k est le k-ième nombre de Fibonacci avec $F_0 = 0$, $F_1 = 1$ et $F_{k+2} = F_{k+1} + F_k$ pour $k \geq 0$. Montrer que ces bornes sont atteintes.

Montrer que pour $k \geq 0$ on a $F_{k+2} \geq \phi^k$ où $\phi = (1 + \sqrt{5})/2$.
En déduire que $\log_2(n+1) \leq h \leq \log_\phi(n+1)$ et que la recherche dans un AVL se fait au pire en temps $O(\log(n))$.

Pour les algorithmes, un nœud x d'un AVL sera représenté par une structure comportant
- une clé, notée $x \cdot \text{cle}$ d'un type totalement ordonné (par exemple de type entier) ;
- un sous-arbre gauche, noté $x \cdot g$ et un sous-arbre droit, noté $x \cdot d$;
- la hauteur de l'arbre, notée $x \cdot h$.

L'arbre vide est noté `NULL`. On notera $h(x)$ la hauteur d'un arbre x avec $h(\texttt{NULL}) = 0$. On notera $n(x)$ le nombre de nœuds d'un arbre x.

b) Le but de cette question est d'écrire une procédure `équilibrer`(x) pour transformer en AVL en temps constant un ABR de racine x en supposant que ses deux sous-arbres sont des AVL et que la différence de hauteur entre les deux sous-arbres est d'au plus 2.

Proposer un rééquilibrage dans le cas où $h(a \cdot g) - h(a \cdot d) = 2$. On pourra distinguer les cas $h(a \cdot g \cdot g) \geq h(a \cdot g \cdot d)$ et $h(a \cdot g \cdot g) < h(a \cdot g \cdot d)$. Illustrer les transformations sur des dessins. Écrire la procédure `équilibrer`(x).

c) Écrire un algorithme pour insérer une clé c dans un AVL x en temps au pire $O(1+h(x))$. Si x' est l'arbre obtenu, comparer $h(x')$ et $h(x)$. Justifier la correction de l'algorithme (on ne demande pas une preuve de programme). Montrer que cet algorithme engendre au plus un rééquilibrage.

d) Écrire un algorithme pour supprimer une clé c dans un AVL x en temps au pire $O(1 + h(x))$. Justifier la correction de l'algorithme (on ne demande pas une preuve de programme). Combien de rééquilibrages peuvent être nécessaires ?

e) Écrire un algorithme qui réalise la fusion de deux AVL x et y et d'une clé c en supposant que toutes les clés de x sont strictement inférieures à c et que toutes les clés de y sont strictement supérieures à c. Cet algorithme devra fonctionner en temps $O(1+|h(x)-h(y)|)$. Justifier.

f) Écrire un algorithme qui réalise la scission d'un AVL x en deux AVL y et z contenant respectivement les clés de x inférieures ou égales à c pour y et strictement supérieures à c pour z. Cet algorithme devra fonctionner en temps $O(1 + |h(x)|)$. Justifier.

2 Recherche de motif et automates

Dans cet exercice, l'alphabet A est fixé. Le mot vide est noté ε.

On note $t = t[1] \cdots t[n]$ le texte de longueur n et $x = x[1] \cdots x[m]$ le motif de longueur m. On notera $t[i \cdot\cdot j] = t[i] \cdots t[j]$ le facteur de t commençant à la position i et se terminant à la position j. Par convention, ce facteur est vide si $j < 1$ ou $i > n$ ou $j < i$.

a) On suppose donné un automate déterministe complet $\mathcal{A} = (Q, A, \delta, q_0, F)$ qui reconnaît le langage $A^* \cdot \{x\}$ des mots qui se terminent par x. Écrire un algorithme, basé sur l'automate $\mathcal{A}$, qui affiche toutes les positions des occurrences de x dans t en temps $O(|t|)$.

Le but des questions **(b)** à **(g)** est de calculer l'automate minimal du langage $A^* \cdot \{x\}$ en temps $O(|A| \cdot |x|)$. On rappelle que la fonction de transition δ s'étend à A^* en posant pour tout $q \in Q$, $\delta(q, \varepsilon) = q$ et $\delta(q, va) = \delta(\delta(q, v), a)$ pour $v \in A^*$ et $a \in A$. On rappelle aussi que l'automate $\mathcal{A}$ est minimal si pour tous $p, q \in Q$ avec $p \neq q$ il existe un mot $v \in A^*$ tel que $\delta(p, v) \in F$ et $\delta(q, v) \notin F$ ou le contraire.

On utilisera les notions de *bord* et de *bord disjoint* définies en cours. On note $\mathrm{Bord}(v)$ le plus long bord de v et si v est un préfixe de x on note $\mathrm{BD}_x(v)$ le plus long bord de v disjoint dans x (on pose $\mathrm{BD}_x(v) = \perp$ si v n'a pas de bord disjoint dans x).

Le motif x étant fixé, pour $1 \leq j \leq m$, on note $b(j) = |\mathrm{Bord}(x[1 \cdot\cdot j])|$ et $bd(j) = |\mathrm{BD}_x(x[1 \cdot\cdot j])|$ avec la convention $bd(j) = -1$ si $x[1 \cdot\cdot j]$ n'a pas de bord disjoint dans x. On rappelle qu'on peut calculer en temps $O(m)$ les fonctions (tableaux) b et bd.

Si $v \in A^*$, on note $\mathrm{Pref}(v)$ l'ensemble des préfixes de v. On note aussi $\mathrm{Suff}(v)$ l'ensemble des suffixes de v. Noter que $|\mathrm{Pref}(v)| = |\mathrm{Suff}(v)| = |v| + 1$. Le motif x étant toujours fixé, on définit la fonction $f : A^* \to \mathrm{Pref}(x)$ par $f(v) = \max(\mathrm{Pref}(x) \cap \mathrm{Suff}(v))$.

b) Soit $v \in A^*$ et $a \in A$. Montrer que $f(va) = f(f(v)a)$.

c) Soit $v \in \mathrm{Pref}(x)$ et $a \in A$. Montrer que $f(va) = \begin{cases} va & \text{si } va \in \mathrm{Pref}(x) \\ \mathrm{Bord}(va) & \text{sinon.} \end{cases}$

d) Soit $v \in \mathrm{Pref}(x)$ et $a \in A$. Montrer que $f(va) = \begin{cases} va & \text{si } va \in \mathrm{Pref}(x) \\ f(\mathrm{Bord}(v)a) & \text{sinon.} \end{cases}$

e) Soit $v \in \mathrm{Pref}(x)$ et $a \in A$. Montrer que $f(va) = \begin{cases} va & \text{si } va \in \mathrm{Pref}(x) \\ f(\mathrm{BD}_x(v)a) & \text{si } \mathrm{BD}_x(v) \neq \perp \\ \varepsilon & \text{sinon.} \end{cases}$

On définit l'automate des occurrences $\mathcal{A}_x = (\mathrm{Pref}(x), A, \delta, \varepsilon, \{x\})$ où l'ensemble des états est $\mathrm{Pref}(x)$, l'état initial est le mot vide ε, l'état final est le mot x et la fonction de transition est définie par $\delta(v, a) = f(va)$ pour $v \in \mathrm{Pref}(x)$.

f) Montrer que $\mathcal{A}_x$ est un automate déterministe complet qui reconnaît $A^* \cdot \{x\}$. Montrer que A_x est minimal.

g) Écrire un algorithme qui calcule la fonction $\tilde{\delta} : \{0, \ldots, m\} \times A \to \{0, \ldots, m\}$ définie par $\tilde{\delta}(i, a) = |\delta(x[1 \cdot\cdot i], a)|$ en temps $O(|A| \cdot |x|)$.
Remarque : la fonction $\tilde{\delta}$ code la fonction δ.

Dans la suite, on veut éviter le facteur induit par l'alphabet dans le codage et le calcul de l'automate $\mathcal{A}_x$. On cherche donc un codage de δ que l'on puisse calculer en temps $O(|x|)$ (et qui utilise donc un espace $O(|x|)$).

Une transition $p \xrightarrow{a} q$ de $\mathcal{A}_x$ est dite *passive* si $q = \varepsilon$ et active sinon. Une transition active est une *flèche avant* si $q = pa$ et c'est une *flèche arrière* sinon, i.e., si $\varepsilon \neq q \neq pa$. Clairement, il y a donc m flèches avant dans $\mathcal{A}_x$. On va montrer qu'il y a au plus m flèches arrière.

On dit qu'un entier $p > 0$ est période d'un mot w si $w[i] = w[i+p]$ pour tout $1 \leq i \leq |w|-p$. On remarque que si $w \neq \varepsilon$ alors $|w|$ est période de w. On note $\mathrm{per}(w)$ la plus petite période de w.

h) Soit $w \neq \varepsilon$. Montrer que $|w| = \mathrm{per}(w) + |\mathrm{Bord}(w)|$.

i) Soient $u \xrightarrow{a} f(ua)$ et $v \xrightarrow{b} f(vb)$ deux flèches arrière. Montrer que $ua \neq vb$ implique $\mathrm{per}(ua) \neq \mathrm{per}(vb)$. En déduire que l'automate $\mathcal{A}_x$ comporte au maximum m flèches arrière.

Pour chaque entier i avec $0 \leq i \leq m$, soit $\bar{\delta}(i)$ la liste des flèches actives issues de $x[1 \cdot\cdot i]$ ordonnée par longueurs décroissantes. Chaque flèche est représentée par un couple (a, j) où j est la longueur de l'état d'arrivée : $x[1 \cdot\cdot i] \xrightarrow{a} x[1 \cdot\cdot j]$. Noter que si (a, j) est dans la liste $\bar{\delta}(i)$ alors $a = x[j]$ puisque la flèche représentée est active. Il n'est donc pas utile en fait de mémoriser les lettres. Si $i < m$, le premier élément de la liste $\bar{\delta}(i)$ est $(x[i+1], i+1)$.

j) Expliquer pourquoi $\bar{\delta}$ est un codage de $\tilde{\delta}$ (et donc de δ). Soit $i \in \{1, \ldots, m\}$ tel que $bd(i) \neq -1$. Comparer $\bar{\delta}(i)$ et $\bar{\delta}(bd(i))$. En déduire un algorithme qui calcule $\bar{\delta}$ en temps $O(|x|)$. Justifier cette complexité.

k) Écrire l'algorithme de recherche de motif qui s'en déduit. Cet algorithme est appelé algorithme de Simon.

l) Montrer que si $v \xrightarrow{a} ua$ est une flèche arrière alors il existe k tel que $u = \mathrm{BD}_x^k(v)$. En déduire que l'algorithme de Simon ne fait pas plus de comparaisons de caractères que l'algorithme de Knuth, Morris et Pratt. Quelle est la complexité de l'algorithme de Simon ?

Algorithmique

Magistère STIC

Partiel du 15 décembre 2005
durée 3 heures

Les notes de cours et de TD sont autorisées.

Les exercices sont indépendants.

Toutes les réponses devront être correctement justifiées. La rigueur des raisonnements, la clarté des explications, et la qualité de la présentation influeront sensiblement sur la note.

1 Insertion à la racine d'un ABR

On s'intéresse dans cette partie à des arbres binaires de recherche (ABR). Dans certains cas, les éléments que l'on a le plus de chances de rechercher sont ceux qui ont été insérés le plus récemment. Dans ce cas l'algorithme classique d'insertion aux feuilles n'est pas très adapté.

Soit t un ABR et x une clé. Lorsqu'on insère x dans t à la racine, la profondeur d'un nœud augmente d'au plus 1 et bien sûr la clé x se retrouve à la racine du nouvel ABR t'. Donc l'avant dernier élément inséré est un fils de la racine, et l'avant avant dernier élément inséré est un fils ou un petit-fils de la racine.

a) Dessiner les arbres obtenus après chaque insertion à partir de l'arbre vide par insertions à la racine des clés 7, 4, 9, 5, 6, 1, 3, 2, 8.

b) Écrire un algorithme d'insertion à la racine dont la complexité en temps est proportionnelle à la hauteur de l'arbre.
Donner le ou les invariants (de boucle ou de récursivité) et prouver la correction de l'algorithme. Justifier aussi sa complexité.

c) Ordonner les entiers de 1 à 7 de telle façon que la création d'un ABR par insertions successives à la racine de ces entiers dans cet ordre donne l'ABR parfait (toutes les branches ont la même longueur).

d) Donner un algorithme qui, étant donné un entier k, affiche les entiers de 1 à $2^k - 1$ dans un ordre permettant d'obtenir un arbre parfait par insertion à la racine de ces entiers à partir d'un arbre vide.
Prouver la correction de l'algorithme et calculer sa complexité.

e) Comparer l'arbre obtenu à partir d'un arbre vide par insertion à la racine d'une suite d'entiers 2 à 2 distincts et l'arbre obtenu à partir de l'arbre vide par insertion aux feuilles de cette suite d'entiers dans l'ordre inverse.

2 Recherche de motif par duels

On note $t = t[1] \cdots t[n]$ le texte de longueur n et $x = x[1] \cdots x[m]$ le motif de longueur m. On notera $t[i \cdot\cdot j] = t[i] \cdots t[j]$ le facteur de t commençant à la position i et se terminant à la position j. Par convention, ce facteur est vide si $j < 1$ ou $i > n$ ou $j < i$.

On rappelle qu'un *bord* d'un mot w est un préfixe strict de w qui est aussi suffixe de w. On note $\mathrm{Bord}(w)$ le plus long bord de w.

Le motif x étant fixé, on note $b(j) = |\mathrm{Bord}(x[1 \cdot\cdot j])|$ la longueur du plus long bord de $x[1 \cdot\cdot j]$. On rappelle que la fonction $b : \{1, \ldots, m\} \to \{0, \ldots, m-1\}$ peut se calculer en temps linaire. Plus précisément, on peut calculer en temps $\mathcal{O}(m)$ un tableau B défini par $B[i] = b(i)$ pour $1 \leq i \leq m$.

Le motif x étant toujours fixé, on définit la fonction $p : \{1, \ldots, m\} \to \{0, \ldots, m-1\}$ par

$$p(i) = \max\{k \mid 0 \leq k \leq m - i \text{ et } x[i+1 \cdot\cdot i+k] \text{ est préfixe de } x\}.$$

a) Calculer les fonctions b et p pour le motif $x = abcabaabcababc$.

Les questions **(b)** à **(i)** visent à élaborer un algorithme qui calcule le tableau représentant la fonction p en temps linéaire.

b) Montrer que pour $1 \leq j \leq m$ on a $b(j) \leq p(j - b(j))$.

c) Montrer que pour $1 \leq i \leq m$ on a $p(i) \leq b(i + p(i))$.

Pour $1 \leq i \leq m$ on définit $E(i) = \{j \mid i < j \leq m \text{ et } j - b(j) = i\}$ et

$$f(i) = \begin{cases} 0 & \text{si } E(i) = \emptyset \\ \max\{b(j) \mid j \in E(i)\} & \text{sinon.} \end{cases}$$

d) Calculer la fonction f pour le motif $x = abcabaabcababc$. Pour $1 \leq i \leq m$, montrer que

- $f(i) \leq p(i)$ et
- $E(i) \neq \emptyset$ si et seulement si $f(i) > 0$.

e) Montrer que si $j \in E(i)$ et $j \leq j' \leq i + p(i)$ alors $j' \in E(i)$. En déduire que $E(i) \neq \emptyset$ implique $f(i) = p(i)$.

f) Montrer que si $p(i') > 0$ alors il existe i tel que $E(i) \neq \emptyset$ et $i \leq i' < i' + p(i') \leq i + p(i)$.

g) Montrer que si $i < i'$, $E(i) \neq \emptyset$ et $E(i') \neq \emptyset$ alors $i + p(i) < i' + p(i')$.

h) Montrer que si $i < i' < i + p(i)$ et $E(i) \neq \emptyset$ et $E(i'') = \emptyset$ pour tout $i < i'' \leq i'$ alors $p(i') = \min(p(i' - i),\ p(i) - (i' - i))$.

i) Déduire des questions précédentes un algorithme qui calcule en temps $\mathcal{O}(m)$ le tableau P définissant la fonction p. On justifiera la correction de l'algorithme à l'aide des questions précédentes et on prouvera qu'il s'exécute bien en temps linéaire.

Nous passons maintenant à l'algorithme de recherche de motif par duels. On définit

$$q(i) = \begin{cases} 1 + p(i) & \text{si } i + 1 + p(i) \leq m \\ 0 & \text{sinon.} \end{cases}$$

j) Calculer les fonctions p et q pour le motif $x = bababba$.

On veut trouver les positions $i \in \{0, \ldots, n-m\}$ du texte telles que $x = t[i+1 \cdot\cdot i+m]$. Noter que dans un tel alignement le motif commence à la position $i+1$, ce qui justifie l'intervalle considéré pour i.

Soient $i, i' \in \{0, \ldots, n-m\}$ deux positions du texte. On dit qu'il y a *duel* entre i et i' si $0 < |i'-i| < m$ et $q(|i'-i|) \neq 0$. Sinon les positions i et i' sont dites *compatibles*. Lorsqu'il y a duel entre $i < i'$ alors i' gagne le duel si $x[q(i'-i)] = t[i' + q(i'-i)]$, sinon c'est i qui gagne le duel.

k) Soient $i < i'$ deux positions en duel. Montrer que si i gagne le duel alors $x \neq t[i'+1 \cdot\cdot i'+m]$ et que si i' gagne le duel alors $x \neq t[i+1 \cdot\cdot i+m]$.

l) Soient $i < i' < i''$ des positions du texte telles que i et i' d'une part et i' et i'' d'autre part sont compatibles. Montrer que i et i'' sont compatibles.

On considère l'algorithme suivant :

```
s <- pile vide
pour i <- n-m à 0 faire
  i1 <- i
  tant que s.nonVide() faire
    i2 <- s.dépiler()
    si i1 et i2 sont compatibles alors
      s.empiler(i2)
      sortir de la boucle tant que
    finsi
    i1 <- le gagnant du duel entre i1 et i2
  fin tant que
  s.empiler(i1)
fin pour
marquer toutes les positions contenues dans la pile s
```

m) Exécuter cet algorithme sur le texte $t = abcbababbababbabac$ et le motif $x = bababba$.

n) Montrer que les positions marquées sont deux à deux compatibles et que si $x = t[i+1 \cdot\cdot i+m]$ alors la position i est marquée.

Une position $1 \leq j \leq n$ du texte est *bonne* s'il existe une position marquée i telle que $i < j \leq i+m$ et $t[j] = x[j-i]$.

o) Donner un algorithme qui construit en temps $\mathcal{O}(n)$ un tableau de booléens $g[1 \cdot\cdot n]$ tel que $g[j] = V$ si et seulement si la position j est bonne.

p) Montrer que $x = t[i+1 \cdot\cdot i+m]$ si et seulement si i est une position marquée et $g[i+1 \cdot\cdot i+m]$ ne contient que des V. En déduire un algorithme de recherche de motif qui fonctionne en temps linéaire.