Table des matières

Préface **iii**

1 Parcours de tableaux **1**

2 Jouer avec les mots **3**

3 Stratégies gloutonnes **5**

4 Arborescences **7**
4.1 Tas 7

Préface

Les exercices de cet ouvrage ont été proposés aux oraux des concours d'entrée des Écoles Normales Supérieures de Lyon et de la rue d'Ulm en 1994, 1995 et 1996. Beaucoup sont donnés dans leur forme originelle, d'autres ont été légèrement modifiés pour être un peu plus attractifs et accessibles.

Ces exercices étaient généralement précédés de l'avertissement suivant :

« Le but de cet épreuve est de déterminer votre aptitude à
- mettre en forme et analyser un problème
- maîtriser les méthodes logiques propres à l'informatique
- organiser et traiter des informations
- rechercher, concevoir et mettre en forme un ou des algorithmes
- construire méthodiquement un ou des programmes clairs
- exposer de manière synthétique, claire et concise votre travail.

Le texte de l'épreuve est relativement succinct. Il vous est demandé, suivant votre convenance, de le compléter, pour décrire aussi précisément que possible, les limites d'utilisation de vos algorithmes et programmes. »

Les exercices sont regroupés par thème. L'étudiant pourra ainsi découvrir les grandes classes d'algorithmes et l'enseignant pourra trouver facilement des exemples pour illustrer un cours ou des travaux dirigés.

Mis à part ceux de la rue d'Ulm, les exercices sont proposés avec un corrigé qui ne se veut pas un modèle du genre, qui propose une solution à une question posée. Les algorithmes proposés dans les corrigés sont écrits en OCaml mais aucun exercice n'est dépendant de ce langage de programmation ; ils peuvent tous facilement être retranscrits dans un autre langage. De plus, les parties en OCaml ont juste la prétention d'offrir une mise en forme lisible et rigoureuse des algorithmes proposés. Les fonctions proposées ne sont pas toujours orthodoxes, en particulier pour éviter certaines lourdeurs pouvant rendre la lecture difficile.

Cet ouvrage est destiné à un public que l'on souhaite le plus large possible. Tout étudiant de classe préparatoire ou d'université désireux de s'exercer à l'algorithmique devrait y trouver satisfaction.

Les auteurs.

⋆⋆⋆⋆⋆⋆⋆⋆

1

Parcours de tableaux

« *Ordre, Permutations, Jeux.* »

2

Jouer avec les mots

« *Reconnaissance, Cosntruction, Codage.* »

3

Stratégies gloutonnes

« *Le meilleur du moment, pour trouver le meilleur.* »

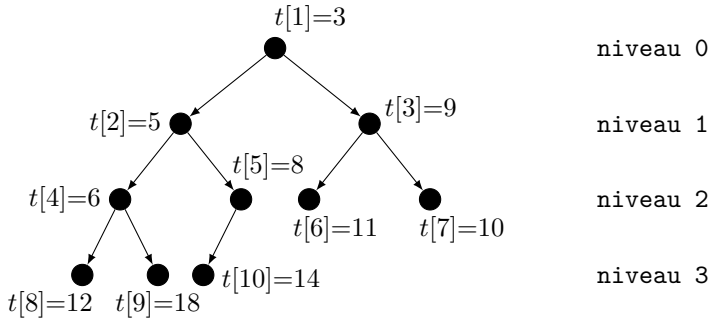
4

Arborescences

« *Un père, des fils. Une racine, des noeuds, des feuilles.* »

4.1 Tas

Un arbre binaire complet est un graphe comme celui de la figure : tous les niveaux sont remplis de gauche à droite, sauf éventuellement le dernier niveau. On numérote les sommets de 1 à n, niveau par niveau, et pour chaque niveau, de gauche à droite. S'il y a une flèche de i vers j, on dit que i est le père de j, et que j est un fils de i. Le sommet d'en haut qui n'a pas de père (niveau 0), est la racine. Les sommets du dernier niveau, qui n'ont pas de fils, sont des feuilles. À chaque sommet numéro i, on associe un attribut entier. On représente alors un arbre binaire complet de taille n par un tableau de taille $n + 1$ dont la première case contient le nombre d'éléments du tas.



Question 1

Pour i donné, $1 \leq i \leq n$, quel est son père, et quels sont ses fils (s'ils existent) ? Un tas est un arbre binaire complet dans lequel tout père est plus petit que ses fils (comme

pour la figure). Le minimum est donc à la racine. Écrire une fonction OCaml qui vérifie si un arbre binaire complet est un tas.

Question 2

Écrire une fonction OCaml qui supprime la racine d'un tas à n sommets et qui renvoie un tas composé des $n-1$ sommets restants (indication : on pourra mettre la dernière feuille à la place de la racine et la faire descendre).

Question 3

Comment insérer un nouvel élément dans un tas à n sommets ?

CORRIGÉ

Question 1

Le père du sommet i est $\lfloor i/2 \rfloor$ pour $2 \leq i \leq n$ (le père du sommet 1 n'est pas défini).
Les fils de i sont $2i$ et $2i+1$ si ces valeurs sont $\leq n$. En particulier :

- si n est le nombre de sommets et $2i = n$, alors i n'a qu'un fils, à savoir n
- si $i > \lfloor n/2 \rfloor$, alors i est une feuille

Pour la fonction demandée, on vérifie les attributs des fils des sommets qui ne sont pas des feuilles, et on sort dès qu'il y a un échec :

```
let verif (t:int array) : bool =
  let n = t.(0) in
  try
    for i=1 to n/2 do
      if t.(i) > t.(2*i) then raise Exit;
      if 2*i+1 <= n && t.(i) > t.(2*i+1) then raise Exit;
    done;
    true
  with Exit -> false
```

Question 2

Une idée naturelle serait d'essayer de remonter le plus petit des fils de la racine à sa place, puis de continuer ainsi... mais on aboutit à un déséquilibre de l'arbre, qui n'est plus complet. Comme l'indication le suggère, il est plus simple de placer le dernier élément à la racine puis de le descendre à la bonne place (percolation basse) :

```
let fils_min (t:int array) (i:int) : int =
  let n = t.(0) in
  let a_fils_droit = 2*i+1 <= n in
  if a_fils_droit then
    if t.(2*i) < t.(2*i+1) then
      2*i else 2*i+1
  else 2*i
```

```
let rec percolation_basse (t:int array) (i:int) : unit =
  let n = t.(0) in
  let a_fils_gauche = 2*i <= n in
  if a_fils_gauche then
    let fils = fils_min t i in
    if t.(fils) < t.(i) then (
      let tmp = t.(i) in
      t.(i) <- t.(fils);
      t.(fils) <- tmp;
      percolation_basse t fils
    )

let supprimer_racine (t:int array) : int =
  let n = t.(0) and rac = t.(1) in
  t.(1) <- t.(n);
  t.(0) <- n-1;
  percolation_basse t 1;
  rac
```

La procédure a une complexité au pire proportionnelle au nombre de niveaux de l'arbre soit $\lceil \log_2 n \rceil$.

Question 3

Comme précédemment, une idée simple est d'ajouter la nouvelle valeur en dernière position, puis de la remonter tant que besoin est (percolation haute) :

```
let rec percolation_haute (t:int array) (i:int) : unit =
  let parent = i/2 in
  if i <> 1 && t.(parent) > t.(i) then (
    let tmp = t.(i) in
    t.(i) <- t.(parent);
    t.(parent) <- tmp;
    percolation_haute t parent
  )

let ajouter (t:int array) (e:int) : unit =
  assert(Array.length t > t.(0) + 1);
  t.(0) <- t.(0) + 1;
  let n = t.(0) in
  t.(n) <- e;
  percolation_haute t n
```

Ici encore, la fonction a une complexité proportionnelle au nombre de niveaux de l'arbre, donc en $O(\lceil \log_2 n \rceil)$.

Signalons que la structure de tas est à la base d'une méthode de tri très performante : le tri par tas. L'idée est simple : on part du tableau `a` de n éléments à trier, et on construit un tas par ajout successif des n éléments. On retire ensuite toutes les racines et on obtient ainsi les éléments dans l'ordre :

```
let tri_par_tas (a:int array) : unit =
  let n = Array.length a in
  let tas = Array.make (n+1) 0 in
  Array.iter (ajouter tas) a;
  for i=0 to n-1 do
    a.(i) <- supprimer_racine tas
  done
```

Le coût de chaque insertion ou suppression est proportionnel à la hauteur du tas courant ($\log_2 p$ pour p éléments), et donc le coût total est de l'ordre de :

$$\sum_{p=1}^{n} \log_2 p = \log_2 n! = O(n \log_2 n)$$

(d'après la formule de Stirling ou par comparaison avec $\int_1^n \log x \, \mathrm{d}x$). Le tri par tas est donc asymptotiquement très rapide.

⋆⋆⋆⋆⋆⋆⋆⋆