Table des matières

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.* »

3.1 Réservation SNCF

On suppose que n personnes veulent prendre le train en un jour donné. La personne i veut prendre le train `p.(i)` où `p` est un tableau d'entiers. Les trains sont numérotés de 0 à $k-1$, et partent dans l'ordre de leur numéro. Chaque train peut contenir au plus c personnes.

Question 1

Écrire une fonction qui teste si tout le monde peut prendre le train de son choix.

Question 2

On suppose maintenant que, si le train `p.(i)` est trop plein pour que la personne i puisse le prendre, elle est prête à prendre le train suivant, soit le `p.(i) + 1`, lorsqu'il existe (c'est-à-dire lorsque `p.(i) < k-1`). Existe-t-il toujours une façon de remplir les trains pour faire voyager tout le monde ? Proposer un algorithme pour répartir les gens dans les trains lorsque cela est possible. Écrire la fonction OCaml correspondante. On pourra par exemple renvoyer un tableau de taille k tel que l'élément d'indice i corresponde au train que prendra la personne i.

Question 3

On suppose maintenant que la personne i, si elle ne peut prendre le train `p.(i)` parce qu'il est trop plein, souhaite prendre un train le plus tôt possible après `p.(i)` (s'il y en a un). Proposer un algorithme pour affecter chaque personne à un train lorsque c'est possible. Écrire la fonction OCaml correspondante.

Question 4

Dans cette question, on suppose que les demandes de réservation se font l'une après l'autre et doivent être traitées immédiatement : il faut donner une réservation à la personne i sans savoir ce que les clients $i+1$, $i+2$, ... vont demander et sans pouvoir changer les réservations données aux personnes 0, 1, 2, ..., $i-1$. L'entrée `p.(i)` dénote maintenant le train demandé à l'instant i par la i-ième personne au guichet. Écrire une fonction qui lui donne une réservation dans le train `p.(i)` s'il n'est pas plein, et dans le premier train non plein après `p.(i)` sinon. Qu'en pensez-vous ?

CORRIGÉ

Question 1

Il suffit de remplir un tableau `t` à k entrées tel que la i-ième entrée soit égale au nombre de gens souhaitant prendre le train i.

```
let choix_satisfiables (p:int array) (k:int) (c:int) : bool =
  let n = Array.length p in
  let t = Array.make k 0 in
  try
    for i=0 to n-1 do
      let choix = p.(i) in
      t.(choix) <- t.(choix) + 1;
      if t.(choix) > c then
        raise Exit
    done;
    true
  with Exit -> false
```

Question 2

Il n'est clairement pas toujours possible de faire voyager tout le monde : par exemple, ce n'est pas possible si plus de c personnes veulent prendre le dernier train. On propose un algorithme de type « glouton », qui remplit les trains un par un dans l'ordre.
Pour remplir le train i, on regarde d'abord tous les voyageurs qui souhaitaient prendre le train $i-1$ mais qui n'y ont pas été affectés, et on les met dans le train i (s'il n'y a pas assez de places, on arrête l'algorithme). Puis, parmi les voyageurs qui veulent prendre le train i, on satisfait le maximum de requêtes possibles.
Après avoir rempli les trains 0, 1, 2, ..., $k-1$, on vérifie qu'il ne reste pas de voyageurs sans affectation.

Cet algorithme garantit que pour tout i, le maximum de gens voyagent dans les trains 0, 1, ..., i, et trouve donc toujours une affectation lorsque cela est possible.

```
let repartition (p:int array) (k:int) (c:int) : int array =
  let n = Array.length p in
  let affectation = Array.make n (-1) in

  let report = ref 0 in (* personnes prenant le train i+1 *)
```

```
  for i=0 to k-1 do (* numéro du train *)
    let nb_voyageurs = ref !report in
    report := 0;

    for j=0 to n-1 do (* numéro du voyageur *)
      if p.(j) = i then
        if !nb_voyageurs = c then (
          affectation.(j) <- i+1;
          report := !report + 1;
          if !report > c then failwith "impossible"
        )
        else (
          affectation.(j) <- i;
          nb_voyageurs := !nb_voyageurs + 1
        )
    done
  done;
  if !report > 0 then failwith "impossible";
  affectation
```

Question 3

On utilise la même idée que la question 2 : celle d'un algorithme glouton. On traite tour à tour les personnes désirant prendre le train 0, puis celles désirant prendre le train 1, etc. Chaque personne est affectée au premier train non rempli à partir de celui qu'elle désire prendre.

```
let repartition2 (p:int array) (k:int) (c:int) : int array =
  let n = Array.length p in
  let affectation = Array.make n (-1) in

  let train_libre = ref 0 in
  for i=0 to k-1 do (* numéro du train *)
    let nb_voyageurs = ref 0 in

    for j=0 to n-1 do (* numéro du voyageur *)
      if p.(j) = i then
        if !train_libre < i then (
          train_libre := i;
          nb_voyageurs := 0
        );

        if !train_libre = k then
          failwith "impossible";

        affectation.(j) <- !train_libre;
        nb_voyageurs := !nb_voyageurs + 1;
```

```
        if !nb_voyageurs = c then (
          train_libre := !train_libre + 1;
          nb_voyageurs := 0
        )
      done
    done;
  affectation
```

Question 4

Ce problème fait partie de la classe de problèmes dits « en ligne », où les données ne sont pas toutes connues dès le départ mais arrivent une à une au cours du temps. Les algorithmes en-ligne forment un domaine actif de la recherche actuelle. Dans cette question, tous les trains se remplissent à peu près en même temps et donc il est nécessaire d'avoir un tableau donnant à chaque instant le nombre de personnes dans chaque train. C'est en fait ici une variante de la question 1, sauf que si un train est plein, au lieu d'arrêter l'algorithme, on cherche un train libre pour le client.

```
let repartition_en_ligne (p:int array) (k:int) (c:int) : int array =
  let n = Array.length p in
  let affectation = Array.make n (-1) in
  let t = Array.make k 0 in

  for i=0 to n-1 do (* numéro du voyageur *)
    let j = ref p.(i) in
    (* trouver le premier train libre *)
    while t.(!j) = c && !j < n-1 do incr j done;

    if t.(!j) < c then (
      affectation.(i) <- !j;
      t.(!j) <- t.(!j) + 1
    )
    else failwith "impossible"
  done;
  affectation
```

⋆⋆⋆⋆⋆⋆⋆⋆

3.2 Chaîne maximum d'une permutation

On considère une permutation p de 0, ..., $n-1$. On note $p_i = (i,\ p(i)) \in \mathbb{N}^2$ et on dit que p_i domine p_j si $i \geq j$ et $p(i) \geq p(j)$.

Question 1

Montrer que la relation de domination est une relation d'ordre partiel. On la note $\geq$, et on note $p_i > p_j$ pour $p_i \geq p_j$ et $p_i \neq p_j$. Une chaîne est une suite de points $p_{i_1} > p_{i_2} > ... > p_{i_k}$. On définit la hauteur de p_i comme étant la longueur maximale d'une chaîne dans l'ensemble $\{p_j$ tels que $p_i \geq p_j\}$ et on note S_h l'ensemble des points de hauteur h.

Question 2

Montrer que si p_i est de hauteur $h > 1$, alors il existe un point p_j, $j < i$, qui appartient à S_{h-1} et est dominé par p_i. Montrer qu'alors le point de S_{h-1} le plus à droite (c'est-à-dire d'abscisse maximum) parmi ceux qui sont à gauche de p_i (c'est-à-dire d'abscisse strictement inférieure à celle de p_i) est aussi dominé par p_i.

Question 3

Proposer un algorithme pour calculer les hauteurs des éléments d'un ensemble de n points.

Question 4

Écrire une fonction qui prend en entrée un tableau p de n entiers, codant la permutation, et donne en sortie la hauteur maximale des p_i et un tableau `hauteur` tel que `hauteur.(i)` contient la hauteur du point p_i.

Question 5

Proposer un algorithme pour trouver une chaîne de p de taille maximale.

CORRIGÉ

Question 1

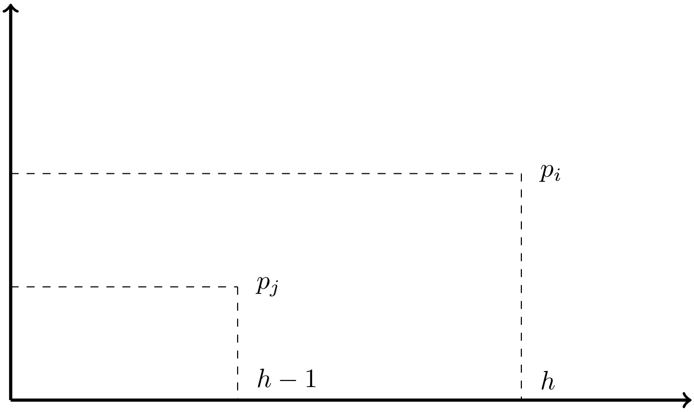
Réflexivité, antisymétrie et transitivité sont immédiates.

Question 2

Comme p_i est de hauteur h, il existe une chaîne $c : p_i > p_{i_2} > p_{i_3} > ... > p_{i_h}$. Soit $p_j = p_{i_2}$. La chaîne c prouve que p_j est de hauteur au moins $h - 1$. Par ailleurs, si p_j était de hauteur h, alors en ajoutant p_j à sa chaîne maximale, on obtiendrait une chaîne de hauteur $h + 1$ pour p_i : impossible. Donc p_j appartient à S_{h-1}.

Supposons les points de S_{h-1} triés par abscisse croissante. Alors leurs ordonnées sont triées par ordre décroissant : en effet, sinon il existerait deux points p et q de S_{h-1} tels que $x(p) \leq x(q)$ et $y(p) < y(q)$, et donc $p < q$: mais p étant de hauteur $h - 1$, q serait alors de hauteur au moins h, ce qui est impossible. L'ensemble des points de S_{h-1} d'abscisse inférieure à ou égale à celle de p_i est non vide, puisque p_i domine au moins un point p_j de hauteur $h - 1$. Soit donc q le point de S_{h-1} d'abscisse maximale parmi ceux à gauche de p_i. L'abscisse de q est supérieure à celle de p_j, donc son ordonnée est inférieure. Par conséquent : $x(q) \leq x(p_i)$ et $y(q) \leq y(p_j) \leq y(p_i)$. Donc p_i domine q.

Voir la figure suivante :



Question 3

On parcourt la liste des points de gauche à droite en déterminant leur hauteur au fur et à mesure, par un algorithme glouton. On va utiliser un tableau `hauteur` tel que `hauteur.(i)` contienne la hauteur de p_i, et un tableau `adroite` tel que `adroite.(i)=j` si p_j est le point le plus à droite parmi les points de hauteur h et `adroite.(i)=-1` s'il n'y a pas de point de hauteur h. De plus, une variable `hmax` donne la hauteur maximale trouvée jusqu'à présent. Supposons que p_0, p_1, ..., p_{i-1} aient déjà été traités, et soit $hmax$ leur hauteur maximale. On veut déterminer la hauteur de p_i. Si p_i domine le point d'indice `adroite.(hmax)`, alors p_i est de hauteur $hmax+1$, sinon, on parcourt le tableau `adroite` dans l'ordre décroissant pour trouver le plus grand $l < i$ tel que p_i domine le point d'indice `adroite.(l)` : p_i est alors de hauteur $l+1$; si p_i ne domine aucun point du tableau `adroite`, alors p_i est de hauteur 1. Il ne reste plus qu'à mettre à jour les tableaux `hauteur` et `adroite` ainsi que la variable `hmax`.

Question 4

On donne la fonction suivante :

```
let hauteurs (p:int array) : (int array * int) =
  let n = Array.length p in
  let hauteurs = Array.make n 1 in
  let adroite = Array.make n (-1) in
  adroite.(0) <- 0;
  let hmax = ref 1 in

for i=1 to n-1 do
  (* calculer la hauteur de pi *)
  let h = ref !hmax in
  while !h > 0 && p.(i) < p.(adroite.(!h-1)) do
    decr h
  done;
  hauteurs.(i) <- !h+1;
  adroite.(!h) <- i;
```

```
  if !h+1 > !hmax then
    hmax := !h+1
  done;
  hauteurs, !hmax
```

Question 5

Il suffit de prendre les points d'abscisses `adroite.(hmax-1)`, `adroite.(hmax-2)`, ..., `adroite.(0)` pour avoir une chaîne de taille maximale.

⋆⋆⋆⋆⋆⋆⋆⋆