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
```

⋆⋆⋆⋆⋆⋆⋆⋆