Table des matières

Préface **iii**

1 Parcours de tableaux **1**

2 Jouer avec les mots **3**

3 Stratégies gloutonnes **5**
3.1 Réservation SNCF 5
3.2 Chaîne maximum d'une permutation 8
3.3 Remplissage optimal ou quasi-optimal d'un camion 11
3.4 Transport de marchandises 12
3.5 Le voleur intelligent 15
3.6 Organisation 16

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

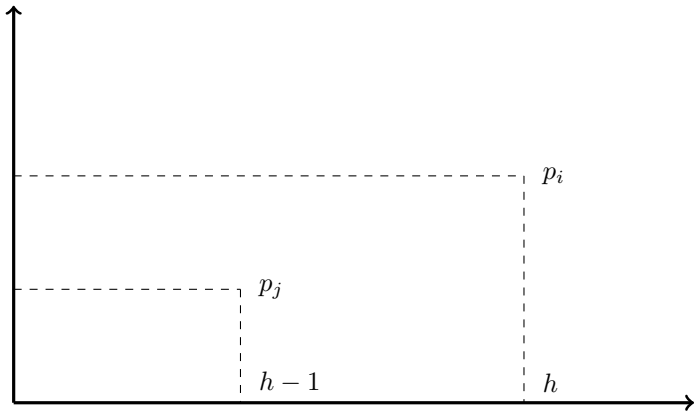
Réflexivité, antisymétrie et transitivité sont immédiates.

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.

⋆⋆⋆⋆⋆⋆⋆⋆

3.3 Remplissage optimal ou quasi-optimal d'un camion

On dispose de n marchandises, de poids respectifs a_1, a_2, ..., a_n, où les a_i sont des entiers strictement positifs ordonnés dans le sens croissant ($a_1 \leq a_2 \leq ... \leq a_n$). On dispose d'un camion dont la charge maximale autorisée est P. On désire le charger avec certaines marchandises, de manière à obtenir le plus grand poids possible inférieur ou égal à P. On cherche donc la plus grande valeur inférieure ou égale à P que l'on puisse obtenir en sommant les éléments d'un sous-ensemble de $\{a_1, a_2, ..., a_n\}$.

Question 1

Écrire une fonction OCaml résolvant le problème (on demande seulement le poids maximal et non l'ensemble correspondant). Quelle est la complexité de votre fonction dans le pire cas ?

Question 2

Pour obtenir un résultat plus rapidement, on va simplifier le problème : on se fixe une valeur $\varepsilon > 0$ donnée, et on cherche maintenant un sous-ensemble de $\{a_1, a_2, ...a_n\}$ dont la somme S des éléments est inférieure à P et telle que si S^* est la somme correspondant à la solution optimale, alors $S \geq S^*(1 - \varepsilon)$. Proposer un programme OCaml résolvant ce problème. Pourquoi est-il plus efficace que le précédent ?

CORRIGÉ

Question 1

On donne la fonction récursive suivante :

```
let poids_max (objets:int list) (poids_max:int) : int =
  let rec aux objets poids = match objets with
    | x::q -> if poids + x <= poids_max
      then max (aux q (poids + x)) (aux q poids)
```

```
      else aux q poids
    | [] -> poids
  in aux objets 0
```

Dans le pire cas, la somme des a_i est inférieure à P et on construit un arbre de décision de taille 2^n. On a donc une complexité exponentielle.

Il n'est pas difficile de modifier la fonction pour renvoyer l'ensemble d'objets correspondant au poids maximal.

Question 2

Il suffit de modifier la fonction précédente de manière à ne considérer un élément que si l'élément précédemment ajouté est inférieur à cet élément multiplié par $(1-\varepsilon/n)$. On montre alors par récurrence sur i que pour tout σ appartenant à l'ensemble des sommes des parties de $a_1, ..., a_i$, il existe un λ appartenant aux objets déjà ajoutés tel que $(1-\varepsilon/n)^i\sigma \leq \lambda \leq \sigma$. On en déduit donc que si S est le résultat donné par l'algorithme et S^* l'optimum, que $(1-\varepsilon/n)^n S^* \leq S$, ce qui donne $(1-\varepsilon)S^* \leq S$. L'incidence sur la complexité provient du fait qu'à tout moment, le rapport entre deux éléments consécutifs considérés est supérieur ou égal à $1/(1-\varepsilon/n)$. Le nombre maximum d'éléments à ajouter est donc majoré par le plus grand k tel que $1/(1-\varepsilon/n)^k \leq P$, ce qui donne $k \approx (n \log P)/\varepsilon$.

⋆⋆⋆⋆⋆⋆⋆⋆

3.4 Transport de marchandises

On dispose de divers modèles de camions (on considère tout d'abord que l'on dispose d'un nombre illimité de camions de chaque modèle !) d'une contenance de 1 tonne, 2 tonnes, 5 tonnes et 10 tonnes. On cherche de combien de façons on peut charger n tonnes ($n > 1$) en utilisant ces camions. Par exemple, si $n = 6$, il y a 5 façons possibles :

- utiliser un camion de 5 tonnes et un camion de 1 tonne
- utiliser 3 camions de 2 tonnes
- utiliser 2 camions de 2 tonnes et 2 camions d'1 tonne
- utiliser 1 camion de 2 tonnes et 4 camions d'1 tonne
- utiliser 6 camions d'une tonne

Question 1

Écrire une fonction OCaml qui calcule le nombre de possibilités.

Question 2

Même chose en supposant qu'on ne dispose pas de plus de `max10` camions de 10 tonnes, `max5` camions de 5 tonnes, `max2` camions de 2 tonnes et `max1` camions d'une tonne.

Question 3

Le stock de camions est à nouveau illimité dans chaque catégorie. Écrire une fonction OCaml qui donne la façon de charger n tonnes qui utilise le moins de camions. Justifier votre réponse. Votre réponse marcherait-elle encore si la contenance des camions était différente ? Que dire si les contenances sont 1, p, p^2, ..., p^k, où p est un entier supérieur ou égal à 2 ?

CORRIGÉ

Question 1

On donne la fonction suivante (où on compte implicitement le nombre de camions de 1 tonne) :

```
let nb_possibilites (n:int) : int =
  let nb = ref 0 in
  for nb10 = 0 to (n / 10) do
    let reste10 = n - 10*nb10 in
    for nb5 = 0 to (reste10 / 5) do
      let reste5 = reste10 - 5*nb5 in
      for nb2 = 0 to (reste5 / 2) do
        incr nb
      done
    done
  done;
  !nb
```

Question 2

Il suffit de changer les bornes des boucles « for » et de vérifier qu'on ne dépasse pas le nombre de camions de 1 tonne.

```
let nb_possibilites_contrainte (n:int) : int =
  let nb = ref 0 in
  for nb10 = 0 to min (n / 10) max10 do
    let reste10 = n - 10*nb10 in
    for nb5 = 0 to min (reste10 / 5) max5 do
      let reste5 = reste10 - 5*nb5 in
      for nb2 = 0 to min (reste5 / 2) max2 do
        let poids_restant = reste5 - 2*nb2 in
        if poids_restant <= max1 then
          incr nb
      done
    done
  done;
  !nb
```

Question 3

On peut bien sûr employer l'algorithme de la question 1 et mémoriser à chaque

fois le nombre de camions mais cette approche est peu efficace car de complexité proportionnelle au nombre de possibilités.

On utilise donc plutôt un algorithme glouton qui utilise le plus possible de camions de 10 tonnes (soit $\lfloor n/10 \rfloor$), puis le plus possible de camions de 5 tonnes (c'est à dire $\lfloor (n \bmod 10)/5 \rfloor$) et ainsi de suite.

```
let nb_camions (n:int) : (int * int * int * int) =
  let n10 = n / 10 in
  let reste = n mod 10 in
  let n5 = if reste >= 5 then 1 else 0 in
  let reste = reste - n5*5 in
  let n2 = reste / 2 in
  let n1 = reste mod 2 in
  (n1, n2, n5, n10)
```

Montrons que cet algorithme donne bien les valeurs recherchées. Pour ceci, montrons d'abord que le nombre de camions de 10 tonnes d'une solution optimale est nécessairement $\lfloor n/10 \rfloor$. Soit une solution optimale :

- cette solution utilise au plus un camion de 5 tonnes (sinon on obtiendrait une meilleure solution en remplaçant 2 camions de 5 tonnes par un camion de 10 tonnes) ;
- cette solution utilise au plus 2 camions de 2 tonnes (sinon on pourrait remplacer 3 camions de 2 tonnes par un de 5 tonnes + un de 1 tonne) ;
- cette solution utilise au plus 1 camion de 1 tonne (sinon on pourrait remplacer 2 camions de 1 tonne par 1 camion de 2 tonnes).

Donc la charge totale constituée par les camions qui ne sont pas des camions de 10 tonnes est au plus de $1 \times 5 + 2 \times 2 + 1 = 10$ tonnes. Elle ne peut pas être exactement de 10 tonnes, sinon on pourrait remplacer le tout par un camion de 10 tonnes, elle est donc au plus de 9 tonnes. Donc le nombre de camions de 10 tonnes est bien de $\lfloor n/10 \rfloor$ et ce qui reste est bien n mod 10. On poursuit le raisonnement sans difficulté avec les camions de 5 puis de 2 tonnes.

Cet algorithme ne donne pas forcément la solution optimale avec d'autres modèles de camions : si on a des camions de 18, 7 et 1 tonnes, et si $n = 21$ tonnes, alors l'algorithme glouton donne 1 camion de 18 tonnes et 3 camions de 1 tonne, alors que la meilleure solution est 3 camions de 7 tonnes.

Si les valeurs des camions sont 1, p, p^2, ..., p^k, où p est un entier supérieur ou égal à 2, l'algorithme glouton donne toujours une solution optimale. On montre comme précédemment que dans une solution optimale :

- il y a au plus $(p-1)$ camions de p^{k-1} tonnes (sinon on pourrait remplacer p camions de p^{k-1} tonnes par un camion de p^k tonnes) ;
- il y a au plus $(p-1)$ camions de p^{k-2} tonnes...

Donc la charge totale des camions de charge strictement inférieure à p^k est au plus $\sum_{i=0}^{k-1}(p-1)p^k = p^k - 1$, donc le nombre de camions de charge p^k doit être $\lfloor n/p^k \rfloor$, etc.

⋆⋆⋆⋆⋆⋆⋆⋆

3.5 Le voleur intelligent

Un cambrioleur entre par effraction dans une maison et désire emporter quelques-uns des objets de valeur qui s'y trouvent. Il n'est capable de porter que X kilos : il lui faudra donc choisir entre les différents objets, suivant leur valeur (il veut bien entendu amasser la plus grande valeur possible).

Question 1

On suppose que les objets sont des matières fractionnelles (on peut en prendre n'importe quelle quantité, c'est le cas d'un liquide ou d'une poudre). Il y a M matières différentes, la i-ième matières vaut un prix `p.(i)` par kilo, et la quantité disponible (en kilos) de cette matière `q.(i)`. On suppose que tous les prix `p.(i)` sont différents deux à deux. Donner un algorithme qui donne un choix optimal pour le voleur.

Question 2

On suppose maintenant que les objets sont non fractionnables (c'est le cas d'une chaise ou d'un téléviseur). Le i-ième objet vaut un prix `p.(i)` (à l'unité, pas au kilo !) et pèse un poids `q.(i)`. Proposer une méthode dérivée de celle de la question 1. Donne-t-elle un choix optimal ? Proposer un algorithme qui donne la valeur optimale que le voleur peut espérer emporter (aide : on construira au fur et à mesure des tableaux qui à l'étape i de l'algorithme contiendront, pour chaque sous-ensemble de l'ensemble des i premiers objets dont la somme des poids est inférieure à X, la somme des poids et la somme des valeurs de ces objets).

CORRIGÉ

Question 1

On utilise un algorithme glouton. On repère la matière la plus précieuse. On en prend le plus possible (jusqu'à ce qu'on en ait X kilos ou que l'on ait épuisé cette matière). Si on peut encore prendre des choses, on continue avec la matière la plus précieuse parmi celles restantes, et ainsi de suite. Justification : supposons que la distribution optimale soit différente. Soit q la quantité de la matière la plus précieuse dans cette distribution et q' celle donnée par notre algorithme. Par construction de notre algorithme $q' \geq q$. Si jamais $q' > q$, on obtient visiblement une meilleure distribution que celle supposée en remplaçant $q' - q$ kilos de n'importe quelle autre matière par $q' - q$ kilos de la matière la plus précieuse, donc $q' = q$. On se ramène au même problème avec $X - q$ kilos transportables et toutes les autres matières sauf la matière la plus précieuse.

Question 2

Dans ce cas, on peut sans difficulté proposer une méthode gloutonne, mais elle ne donne plus une distribution optimale. On peut même être très éloigné de l'optimum : supposons $X = 1$, et 3 objets de poids respectifs $1/2 + \varepsilon$, $1/2 + \varepsilon/2$ et $1/2 - \varepsilon/2$ et de valeurs égales à leur poids. L'optimum consiste bien entendu à prendre les 2ème et 3ème objets, on aura alors une valeur de 1, tandis que la méthode gloutonne conduira

à prendre le premier, on aura une valeur de $1/2 + \varepsilon$. L'algorithme obtenu est le même que celui du problème « remplissage d'un camion ».

Ce problème est connu sous le nom de « problème du sac à dos ». Il est possible de gagner en efficacité par rapport à l'algorithme exhaustif en utilisant un algorithme de type séparation et évaluation (*branch and bound*), maximisant la valeur des objets dans le sac.

⋆⋆⋆⋆⋆⋆⋆⋆

3.6 Organisation

Un organisateur de tournoi sportif désire utiliser au mieux le gymnase locale lors de la journée « portes ouvertes ». Il y a n évènements E_1, ..., E_n, chaque évènement commençant à l'heure d_i et se finissant à l'heure f_i. Autrement dit, l'évènement E_i requiert le gymnase durant l'intervalle de temps $[d_i, f_i[$. Le problème est de planifier le nombre maximal d'évènements parmi les n dans le gymnase.

Question 1

Indiquer comment modéliser la situation.

Question 2

Proposer un algorithme et écrire une fonction OCaml qui résout le problème. On pourra supposer que $f_1 \leq f_2 \leq ... \leq f_n$.

Question 3

Prouver que votre solution conduit bien au nombre maximal d'évènements. Que pensez-vous de votre solution ? Pouvez-vous l'améliorer ?

CORRIGÉ

Question 1

On utilise deux tableaux d'entiers pour stocker les dates de début et de fin, et un tableau d'entiers pour stocker les numéros d'évènements sélectionnés.

Question 2

L'idée est d'utiliser un algorithme glouton. Soit j le dernier évènement ajouté à la liste d'évènements A. Alors $f_j = \max\{f_k \ ; \ k \in A\}$, et donc le gymnase est libre à partir de l'heure f_j. On ajoute un évènement i à la liste si et seulement si $d_i \geq f_j$:

```
let organise (deb:int array) (fin:int array) =
  let n = Array.length deb in
```

```
let rec aux i j acc =   (* j = dernier ajouté *)
  if i=n then acc else
  if deb.(i) >= fin.(j) then
    aux (i+1) i (i::acc)
  else
    aux (i+1) j acc
in List.rev (aux 1 0 [0])
```

Question 3

L'algorithme est en $O(n)$ si les dates de fin son déjà triées. Il faut montrer que le nombre d'évènements obtenu est optimal :

- on montre d'abord qu'il existe une solution optimale qui commence avec E_1. Soit A une solution optimale et soit k l'indice de la première activité de A. Si $k \neq 1$, soit $B = (A \cup E_1) \setminus \{E_k\}$. Comme k est le premier évènement de A, tous les autres commencent après f_k. Comme $f_1 \leq f_k$, B est une solution possible, optimale comme A. D'où le résultat.
- Si A est une solution optimale commençant par E_1, alors $A' = A \setminus \{E_1\}$ est une solution optimale pour le problème où les évènements sont les $\{E_j,\ 2 \leq j \leq n,\ d_j \geq f_1\}$. Sinon, on pourrait trouver une solution B' meilleure que A' pour ce dernier problème, et alors $B' \cup \{E_1\}$ serait meilleur que A !

Par récurrence, on a le résultat.

Une autre solution est d'écrire un algorithme qui calcule m_i, itérativement pour $i = 1$, 2, ..., n, où m_i est le nombre maximal d'évènements compatibles dans E_1, ..., E_i. Le lecteur consciencieux qui résoudra ce problème vérifiera que cette approche est plus coûteuse.

Pour terminer, mentionnons que l'algorithme présenté est un algorithme glouton. D'une manière générale. un algorithme glouton effectue à une étape donnée le meilleur choix qui se présente. Bien sûr, cette stratégie d'optimisation locale n'est pas toujours globalement optimale. C'est le cas dans notre problème, mais considérons pour nous en convaincre le problème des pièces de monnaie : comment obtenir une somme S avec le moins de pièces possibles, les pièces pouvant valoir 10, 5 et 1 centime. Un algorithme glouton proposera d'utiliser $p_{10} = \lfloor S/10 \rfloor$ pièces de 10 centimes, puis $p_5 = \lfloor (S - 10 \cdot p_{10})/5 \rfloor$ pièces de 5 centimes, puis le reste en pièces de 1 centime. Pour ce jeu de pièces, il se trouve que l'algorithme glouton est optimal (heureusement pour notre vie quotidienne !), mais ce n'est pas vrai en général. Ainsi, avec des pièces de 11, 5 et 1 centime : pour obtenir $S = 15$, l'algorithme glouton propose une pièce de 11 centimes et quatre pièces de 1 centime, alors que trois pièces de 5 centimes est le meilleur choix.

⋆⋆⋆⋆⋆⋆⋆