Passeport pour l'informatique

EXERCICES D'ALGORITHMIQUE

Oraux d'ENS

Ouvrage collectif — Coordination Jean-Claude Bajard

INTERNATIONAL™ THOMSON PUBLISHING

Jean-Claude Bajard, Hubert Comon, Claire Kenion,
Daniel Krob, Michel Morvan, Jean-Michel Muller,
Antoine Petit, Mattéo Rizza-Murgier et Yves Robert

Coordination : Jean-Claude Bajard

Exercices d'algorithmique

Oraux d'ENS

INTERNATIONAL THOMSON PUBLISHING FRANCE
An International Thomson Publishing Company

E-mail : *contact@itp.fr*
Listes de diffusion : *listserver@itp.fr*
World-Wide Web : *http ://www.itp.fr*

Paris • Albany • Belmont • Bonn • Boston • Cincinnati • Detroit • Johannesburg
Londres • Madrid • Melbourne • Mexico • New York • Singapour • Toronto • Tokyo

Table des matières

Préface *vii*

Exercices corrigés *1*

1 : Parcours de tableaux *3*

1.1 Jeu d'échecs 3
1.2 Médiane d'un tableau 5
1.3 Travail sur des tableaux 7
1.4 Deuxième plus grand élément 9
1.5 Tri d'un petit nombre d'éléments 11
1.6 Représentation de permutations 15
1.7 Permutations 19

2 : Jouer avec les mots *23*

2.1 Réécriture de mots 23
2.2 Pliage de papier 27
2.3 Plus longue sous-suite croissante 30
2.4 Recherche de motifs 32

2.5 Mots bien parenthésés 35
2.6 Plus longue sous-suite commune 38
2.7 Code correcteur de Hamming 40
2.8 Répétition d'un motif 45

3 : Stratégies gloutonnes *49*

3.1 Réservation SNCF 49
3.2 Chaîne maximum d'une permutation 52
3.3 Remplissage optimal ou quasi-optimal d'un camion 55
3.4 Transport de marchandises 56
3.5 Le voleur intelligent 59
3.6 Organisation 60

4 : Arborescences *63*

4.1 Tas 63
4.2 Arbre bicolore 66
4.3 Code préfixé : codage-décodage 71
4.4 Génération d'un code de Huffman 74
4.5 Tas binomial 77

5 : Graphes *81*

5.1 Fête de Noël sans conflit 81
5.2 La tournée du facteur 85
5.3 Un mariage stable 88
5.4 Coloriage d'un réseau de trains 90
5.5 Graphes de dépendances 91
5.6 Ordonnancement 94

6 : Géométrie et Images *99*

6.1 Pavage d'un rectangle par des dominos 99
6.2 Surface d'un polygone dessiné sur un réseau 102
6.3 Sommes de rectangles 106

6.4 Polygones convexes 111
6.5 Intersection d'un polygone et d'une fenêtre rectangulaire 115
6.6 Intersection de segments 118

7 : Arithmétique et calculs numériques *123*

7.1 Parties d'un ensemble 123
7.2 Conversion d'écriture romaine-décimale 126
7.3 Polynômes à trois variables 130
7.4 JPEG : Transformée en cosinus discrète 134

8 : Vers la récursivité *139*

8.1 L'angoisse de la panne sèche 139
8.2 Produit de plusieurs matrices 141
8.3 Les deux points les plus proches 142
8.4 Hanoï 147

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.

⋆⋆⋆⋆⋆⋆⋆⋆

Exercices corrigés

1

Parcours de tableaux

« *Ordre, Permutations, Jeux.* »

1.1 Jeu d'échecs

Sur un échiquier, on représentera chaque case par ses coordonnées (i, j), la case en bas à gauche étant de coordonnées $(0, 0)$. Sur un tel échiquier, en un coup, un cavalier peut se déplacer de la case (i, j) vers celles d'entre les 8 positions suivantes qui correspondent effectivement à une case de l'échiquier (abscisse et ordonnée comprises entre 0 et 7) : $(i-2, j+1)$, $(i-1, j+2)$, $(i+1, j+2)$, $(i+2, j+1)$, $(i+2, j-1)$, $(i+1, j-2)$, $(i-1, j-2)$ et $(i-2, j-1)$.

Question 1

Écrire une fonction OCaml qui donne toutes les cases accessibles en p coups au plus à partir d'une case (i_0, j_0).

Question 2

Écrire une fonction OCaml qui indique si toutes les cases sont accessibles à partir d'une case (i_0, j_0) donnée, et si oui, quel est le plus petit nombre de coups permettant d'atteindre à partir de cette case n'importe quelle autre case de l'échiquier.

CORRIGÉ

Question 1

Choisissons déjà la structure de données. Définissons un type

```
type case = int * int
```

et donnons-nous une fonction `est_valide (c : case) : bool` qui renvoie `true` si `c` est bien une case de l'échiquier (i.e $0 \leq$ `fst c`, `snd c` ≤ 7), et `false` sinon. Réalisons tout de suite que quel que soit le nombre de coups, on ne pourra jamais atteindre plus de 64 cases et définissons des tableaux

```
let nombre_max_coups = 256 (* nombre max de coups *)
let coups = Array.make_matrix nombre_max_coups 64 (0,0)
let ncases = Array.make nombre_max_coups 0
```

tel que `coups[i,j]` désigne la j-ième des cases atteignables en exactement i coups et `ncases[i]` le nombre de cases atteignables en exactement i coups et non atteignables en moins de coups. La seule difficulté de la fonction et de penser à « faire le ménage »en vérifiant, à chaque fois qu'on ajoute une case, si elle n'a pas déjà été atteinte. Il est à noter qu'il suffit de vérifier ceci pour des valeurs de i (numéros des coups) de même parité que le numéro courant : si on part d'une case blanche, en un nombre pair de coups, on sera forcément sur une case blanche, et en un nombre impair sur une case noire.

```
(* Retourne la k-ième case atteignable a partir de *)
(* la case c selon l'ordre arbitraire de l'énoncé *)
let case_suivante (c:case) (k:int) : case =
  assert (1 <= k && k <= 8);
  let i, j = c in
  let positions = [|
    (i-2, j+1); (i-1, j+2); (i+1, j+2); (i+2, j+1);
    (i+2, j-1); (i+1, j-2); (i-1, j-2); (i-2, j-1)
  |] in
  positions.(k-1)

(* Vérifie si la case c a déjà été atteinte. On est à *)
(* l'étape pcour, et on cherche c parmi les cases *)
(* atteintes à un coup i de même parité que pcour, *)
(* pcour compris *)
let existe_deja (c:case) (pcour:int) : bool =
  let i = ref (pcour mod 2) in
  try
    while !i <= pcour do
      for j=0 to ncases.(!i)-1 do
        if coups.(!i).(j) = c then raise Exit
      done;
      i := !i + 2
    done;
    false
  with Exit -> true

let accessibles (i0:int) (j0:int) (p:int) : unit =
  assert (p < nombre_max_coups);
  coups.(0).(0) <- (i0, j0);
  ncases.(0) <- 1;
```

```
for p_courant=1 to p do
  (* on ajoute les cases atteignables en p coups *)
  (* i.e en 1 coup depuis les cases atteignables en p-1 coups *)
  for idx = 0 to ncases.(p_courant-1)-1 do
    let case_courante = coups.(p_courant-1).(idx) in
    for k=1 to 8 do
      let cs = case_suivante case_courante k in
      if est_valide cs && not (existe_deja cs p_courant) then (
        (* On insère la nouvelle case *)
        let nb_cases = ncases.(p_courant) in
        coups.(p_courant).(nb_cases) <- cs;
        ncases.(p_courant) <- nb_cases + 1;
      )
    done
  done
done
```

Question 2

Il n'y a pas grand chose à modifier : il suffit de remarquer qu'on a forcément atteint toutes les cases atteignables dès que l'ajout d'un nouveau coup n'apporte rien. Il suffit de modifier très légèrement la fonction précédente pour remplacer la boucle « for » sur la variable `p_courant` par une boucle « while » où l'on s'arrête dès que `ncases[p_courant]` vaut zéro. Il est alors facile de voir si toutes les cases sont atteintes (il faut simplement additionner les valeurs des `ncases[i]` pour voir si on trouve 64, surtout ne pas tester pour toute case si elle est dans le tableau `coups`, ceci serait trop long). Le lecteur pourra essayer d'évaluer le coût de ces fonctions. Sans réfléchir, on trouve quelque chose d'exponentiel en le nombre de coups, sauf si on se souvient que l'échiquier n'a que 64 cases !

⋆⋆⋆⋆⋆⋆⋆⋆

1.2 Médiane d'un tableau

On dispose d'un tableau A de n entiers distincts.

Question 1

Écrire une fonction OCaml `echange (a:int array) (i:int) (j:int) : unit` qui échange les éléments d'indices i et j du tableau A.

Question 2

Soient g et d deux entiers, $1 \leq g \leq d \leq n$. Posons $\alpha = A[g]$. On désire effectuer une permutation des éléments de A d'indice compris entre g et d, qui soit telle qu'après la permutation il existe un entier *pivot*, $(g \leq pivot \leq d)$ vérifiant :

- $A[pivot] = \alpha$,
- pour tout i compris entre g et $pivot$, $A[i] \leq \alpha$,
- pour tout i compris entre $pivot + 1$ et d, $A[i] > \alpha$,
- les éléments de A d'indice strictement inférieur à g ou strictement supérieur à d restent inchangés.

Par exemple, si $n = 6$, si les éléments de A sont 5, 8, 7, 3, 9, 15, si $g = 1$ et $d = 5$, les éléments de A après la permutation seront 5, 3, 7, 8, 9, 15 ou 5, 7, 3, 8, 15, 9, ou... (il n'y a pas unicité des permutations possibles), et *pivot* sera égal à 3. Écrire une fonction OCaml qui effectue la permutation et donne la valeur de *pivot* sans utiliser un autre tableau que A. Combien d'affectations (c'est à dire d'instructions « `<-` ») et de tests nécessite-t-elle ?

Question 3

On appelle *médiane* de A un couple (i, α) tel que $1 \leq i \leq n$, $A[i] = \alpha$, et $E(n/2)$ éléments de A sont inférieurs strictement à α ($E(x)$ est la partie entière de x). Proposer une fonction de calcul de la médiane de A utilisant la fonction de la question 2.

CORRIGÉ

Question 1

La question 1 est élémentaire.

Question 2

Pour la fonction donnant la permutation (c'est une fonction dite de *partition*), on maintient deux indices i et j tels qu'à tout moment les éléments d'indice compris entre g et i sont tous inférieurs ou égaux à α, tandis que les éléments d'indice compris entre j et d sont tous supérieurs ou égaux à α.

```
let partition (a:int array) (g:int) (d:int) : int =
  let alpha = a.(g) in
  let i = ref g and j = ref d in
  while !i < !j do (
    while a.(!j) >= alpha && !j > g do
      decr j
    done;
    while a.(!i) <= alpha && !i < d do
      incr i;
    done;
    if (!i < !j)
      then echange a !i !j
  ) done;
  echange a !j g;
  !j (* on renvoie le pivot *)
```

Question 3

Partitionnons le tableau A complet à l'aide de la fonction précédente. Si $pivot < n/2$, c'est-à-dire s'il y a plus d'éléments de A supérieurs au pivot que d'éléments inférieurs au pivot, alors la médiane est à droite du pivot, il faut poursuivre la recherche à droite. Dans le cas contraire, il faut poursuivre la recherche à gauche du pivot.

```
let mediane (a:int array) : int =
  let n = Array.length a in
  let g, d = ref 0, ref (n-1) in
  while !d - !g > 1 do
    let pivot = partition a !g !d in
    if pivot < n/2
      then g := pivot+1
      else d := pivot-1
  done;
  !g
```

⋆⋆⋆⋆⋆⋆⋆⋆

1.3 Travail sur des tableaux

On se donne un tableau T d'entiers de taille N.

Question 1

On se donne un entier $p < N$ et un entier s, et on recherche dans le tableau T les indices k tels que $\sum_{i=k}^{p+k} T[i] \geq s$. Écrire une fonction OCaml qui effectue cette recherche. Donner en fonction de N et p un ordre de grandeur du nombre de tests et d'opérations arithmétiques effectuées lors de l'exécution de votre fonction. Pouvez-vous l'améliorer ?

Question 2

On suppose maintenant que $T[0] < T[1] < T[2] < ... < T[N-1]$. Pouvez-vous tenir compte de cette information pour obtenir une fonction plus rapide ?

Question 3

Proposer une fonction qui affiche les triplets d'entiers i, j, k avec $j < i < N$ tels que $i^2 + j^2 = k^2$. Pouvez-vous l'améliorer ?

CORRIGÉ

Question 1

Première idée : une fonction intuitive, mais quelque peu idiote

```
let recherche (t:int array) (p:int) (s:int) : unit =
  let n = Array.length t in
  for k=0 to n-p do
    let somme = ref 0 in
    for i=k to p+k do
      somme := !somme + t.(i)
    done;
    if !somme >= s then
      (print_int k; print_newline ())
  done
```

Si on réalise que lorsque l'on passe de l'étape k à l'étape $k+1$ dans l'algorithme précédent, deux termes seulement de la somme changent, on obtient la fonction suivante, plus rapide si p est plus grand que 2.

```
let recherche (t:int array) (p:int) (s:int) : unit =
  let n = Array.length t in
  let somme = ref 0 in
  for i=0 to p do
    somme := !somme + t.(i)
  done;
  if !somme >= s then print_endline "0";

  for k=1 to n-1-p do
    somme := !somme - t.(k-1) + t.(k+p);
    if !somme >= s then
      (print_int k; print_newline ())
  done
```

Question 2

L'idée est bien sûr de procéder par dichotomie pour savoir à partir de quel rang on a $\sum_{i=k}^{p+k} T[i] \geq s$. On peut procéder comme suit :

```
(* Renvoie l'indice k à partir duquel on a la propriété *)
let dichotomie (t: int array) (p:int) (s:int) : int =
  let n = Array.length t in
  let min = ref 0 and max = ref (n-p-1) in
  while !min <= !max do
    let m = (!min + !max) / 2 in
    if somme t m p >= s
      then max := m - 1
      else min := m + 1
  done;
  !max + 1
```

Plusieurs variantes permettant de faire moins d'additions en utilisant le « truc » de la question 1 sont possibles.

Question 3

La première idée (idiote !) est de faire une boucle sur i, j et k en utilisant le fait que $i^2 < k^2 = i^2 + j^2 < 2 \times i^2$ donc $i < k < \sqrt{2} \times i < 1,5 \times i$ d'où $i + 1 \leq k \leq \lfloor 1,5 \times i \rfloor$:

```
let triplets (n:int) =
  for i=1 to n-1 do
    for j=1 to i-1 do
      let i' = float_of_int i in
      let sup = int_of_float (1.5 *. i') in
      for k=i+1 to sup do
        if i*i + j*j = k*k then
          Printf.printf "%d %d %d\n" i j k
      done
    done
  done
```

on peut ensuite se dire que le seul k qui a une chance de convenir est $\sqrt{i^2 + j^2}$ (si c'est un entier !), ce qui donne :

```
let triplets (n:int) =
  for i=1 to n-1 do
    for j=1 to i-1 do
      let r = float_of_int (i*i + j*j) in
      let k = Float.round (sqrt r) in
      if k*.k = r then
        Printf.printf "%d %d %d\n" i j (int_of_float k)
    done
  done
```

⋆⋆⋆⋆⋆⋆⋆⋆

1.4 Deuxième plus grand élément

Soit T un tableau unidimensionnel d'entiers de taille $n \geq 1$. On cherche à déterminer l'indice du deuxième plus grand élément de T (celui qui viendrait en deuxième position si on rangeait les éléments de T par ordre décroissant).

Question 1

Écrire une fonction OCaml qui calcule l'indice du deuxième plus grand élément de T en parcourant une fois le tableau T.
Quel nombre de comparaisons entre éléments du tableau effectue votre fonction ?

Question 2

Pour imaginer une meilleure solution, penser à un tournoi de tennis. Le deuxième meilleur joueur n'est pas forcément le finaliste mais figure parmi les adversaires du

gagnant. Pourquoi ?
Donner un algorithme qui utilise cette analogie, et déterminer le nombre de comparaisons entre éléments du tableau qu'il effectue.
Écrire alors la fonction OCaml correspondante.

CORRIGÉ

Question 1

Première solution évidente :

```
let deuxieme (t:int array) : int =
  let n = Array.length t in
  let max_1 = ref (max t.(0) t.(1)) in
  let max_2 = ref (min t.(0) t.(1)) in
  for i=2 to n-1 do
    if t.(i) > !max_1 then
      (max_2 := !max_1; max_1 := t.(i))
    else if t.(i) > !max_2 then
      max_2 := t.(i)
  done;
  !max_2
```

Les $n-2$ comparaisons `t.(i) > !max_1` sont toujours effectuées. Les $n-2$ comparaisons `t.(i) > !max_2` le sont aussi dans le pire cas d'un tableau décroissant. Avec les comparaisons implicites de `min` et `max` (une seule suffirait mais serait moins lisible), on arrive à $2n-3$, ce qui est intuitif : il faut $n-1$ comparaisons pour trouver le plus grand parmi n (chacun sauf le plus grand doit avoir été comparé à plus grand que lui) et $n-2$ pour trouver le plus grand parmi les $n-1$ éléments restants (même argument).

Question 2

C'est clair : tout autre joueur que le gagnant et ses adversaires malheureux est de classement ≥ 3 puisqu'on connaît deux meilleurs joueurs que lui.

Pour l'algorithme, on simule un tournoi de tennis. On transforme les éléments du tableau en feuilles puis on les fusionne en mettant l'indice du plus grand élément des deux à la racine. En itérant jusqu'à n'obtenir qu'un seul arbre, on crée effectivement un arbre de tournoi dont l'indice du vainqueur est la racine.

On effectuera $n-1$ comparaisons pour trouver le gagnant (maximum) car chaque match élimine un joueur. On cherche alors le maximum parmi les $\lceil \log_2 n \rceil$ joueurs battus par le gagnant.

On commence par définir un type d'arbre et une fonction utilitaire :

```
type tree = Node of (tree * int * tree) | Leaf of int

let etiquette tree = match tree with
  | Node(_, x, _) | Leaf x -> x
```

```
let deuxieme (t:int array) : int =
  let n = Array.length t in
  (* liste de tous les joueurs *)
  let joueurs = List.init n (fun i -> Leaf i) in

  (* simulation des duels, retourne un arbre de tournoi *)
  let rec jouer jou gagn = match jou with
    | j1::j2::q ->
      let i1, i2 = etiquette j1, etiquette j2 in
      let g = if t.(i1) > t.(i2) then i1 else i2 in
      let res = Node (j1, g, j2) in
      jouer q (res::gagn)
    | [g] when gagn = [] -> g
    | l -> jouer (l@gagn) [] (* un seul joueur ou liste vide *)
  in
  let matchs = jouer joueurs [] in

  (* on descend dans l'arbre pour trouver *)
  (* le plus grand adversaire du premier *)
  let premier = etiquette matchs in
  let rec traverser res m = match res with
    | Leaf _ -> m
    | Node (g, gagn, d) ->
      if etiquette g = premier then
        let idx = etiquette d in
        let m = max m t.(idx) in
        traverser g m
      else
        let idx = etiquette g in
        let m = max m t.(idx) in
        traverser d m
  in traverser matchs min_int
```

⋆⋆⋆⋆⋆⋆⋆⋆

1.5 Tri d'un petit nombre d'éléments

Soit n un entier ≥ 2. Le but de cet exercice est d'étudier les algorithmes de tri d'un tableau de n éléments entiers, pour $n = 3, 4, 5$ et 10. On ne compte que les comparaisons entre éléments du tableau, et on note $Comp(n)$ leur nombre.

Question 1

Donner un algorithme, et écrire la fonction OCaml correspondante, pour trier un tableau de taille $n = 3$ avec $Comp(n) = 3$.
Même question avec $n = 4$ et $Comp(n) = 5$.
Même question avec $n = 5$ et $Comp(n) = 7$.

Question 2

On suppose que $n = 10$. On demande de préciser le nombre de comparaisons pour chacun des deux algorithmes basés sur les principes suivants (mais on ne demande aucune fonction OCaml dans cette question) :

Algorithme 1. Faire deux listes de 5 éléments, utiliser deux fois l'algorithme précédent pour $n = 5$ et fusionner les deux listes triées obtenues.

Algorithme 2. Faire 5 paires d'éléments et trier les 5 plus grands de chaque paire à l'aide de l'algorithme pour $n = 5$. Puis insérer les éléments restants dans la liste formée.

CORRIGÉ

Commençons par définir une fonction qui range dans l'ordre deux éléments d'un tableau `t` en positions `i` et `j` :

```
let ranger (t:int array) (i:int) (j:int) : unit =
  if t.(i) > t.(j) then (
    let tmp = t.(i) in
    t.(i) <- t.(j);
    t.(j) <- tmp
  )
```

Question 1

Très facile avec 3 éléments : on met le plus petit élément en position 1 en le comparant aux deux autres, puis on classe les deux éléments restants :

```
let tri_3 (t:int array) : unit =
  ranger t 0 1;
  ranger t 0 2;
  ranger t 1 2
```

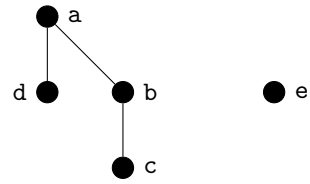
Pour 4 éléments : on classe les paires en position (0, 1) et (2, 3), puis on met les deux plus petits en position 0 et le plus grand des deux plus grands en position 3. Enfin, reste à classer les éléments en positions 1 et 2 :

```
let tri_4 (t:int array) : unit =
  ranger t 0 1;
  ranger t 2 3;
  ranger t 0 2;
  ranger t 1 3;
  ranger t 1 2
```

Une autre solution est de trier les 3 premiers éléments (3 comparaisons) puis d'insérer le quatrième, ce qui coûte 2 comparaisons supplémentaires en procédant par dichotomie, c'est-à-dire en commençant par l'élément du milieu.

Avec 5 éléments : trier les 4 premiers éléments puis insérer le cinquième par dichotomie est ici trop coûteux : il faut 3 comparaisons pour insérer un élément dans une liste de taille 4. Cette solution nous coûterait $5 + 3 = 8$ comparaisons.

Voici une solution en 7 comparaisons : on prend les 4 premiers éléments, on classe les paires (0, 1) et (2, 3) et on compare les deux plus grands éléments. La situation se résume à l'aide du diagramme (où $\texttt{a} \geq \texttt{b} \geq \texttt{c}$ et $\texttt{a} \geq \texttt{d}$) :



On insère alors le cinquième élément e dans la chaîne a, b, c de longueur 3, ce qui coûte 2 comparaisons. Enfin, on insère le quatrième élément d parmi les éléments inférieurs à a dans l'ensemble a, b, c, e ordonné, ce qui coûte deux comparaisons.

On commence par définir une fonction utilitaire :

```
let remplir_tableau (t:int array) (v,w,x,y,z) =
  t.(0) <- v; t.(1) <- w; t.(2) <- x; t.(3) <- y; t.(4) <- z

let tri_5 (t:int array) : unit =
  (* construction du diagramme *)
  ranger t 0 1; ranger t 2 3;
  let a, b, c, d = if t.(1) > t.(3)
    then t.(1), t.(3), t.(2), t.(0)
    else t.(3), t.(1), t.(0), t.(2)
  and e = t.(4) in

  (* insertion de e dans la liste, puis de d *)
  if e > b then
    if e > a then (* ordre c,b,a,e *)
      if d > b then
        remplir_tableau t (c,b,d,a,e)
      else
        if d > c then
          remplir_tableau t (c,d,b,a,e)
        else
          remplir_tableau t (d,c,b,a,e)
    else (* ordre c,b,e,a *)
      if d > b then
        if d > e then
          remplir_tableau t (c,b,e,d,a)
        else
          remplir_tableau t (c,b,d,e,a)
      else
        if d > c then
          remplir_tableau t (c,d,b,e,a)
        else
          remplir_tableau t (d,c,b,e,a)
  else
```

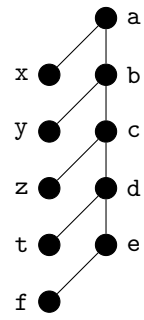
```
if e > c then (* ordre c,e,b,a *)
  if d > e then
    if d > b then
      remplir_tableau t (c,e,b,d,a)
    else
      remplir_tableau t (c,e,d,b,a)
  else
    if d > c then
      remplir_tableau t (c,d,e,b,a)
    else
      remplir_tableau t (d,c,e,b,a)
else (* ordre e,c,b,a *)
  if d > c then
    if d > b then
      remplir_tableau t (e,c,b,d,a)
    else
      remplir_tableau t (e,c,d,b,a)
  else
    if d > e then
      remplir_tableau t (e,d,c,b,a)
    else
      remplir_tableau t (d,e,c,b,a)
```

La fonction est volumineuse mais ne pose pas de difficulté particulière.

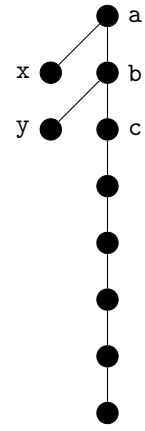
Question 2

Pour l'algorithme 1, le décompte est facile : on utilise deux fois la fonction `tri_5`, ce qui conduit à 14 comparaisons. Reste à fusionner deux listes triées de 5 éléments. Or la fusion de deux listes triées de tailles respectives m et n requiert $m+n-1$ comparaisons, soit ici 9 comparaisons. On obtient un total de 23 comparaisons.

Pour l'algorithme 2, après les 5 premières comparaisons et la fonction `tri_5` appliquée aux 5 plus grands, on a la situation suivante :



Reste à insérer les 4 éléments x, y, z et t dans la chaîne ordonnée de longueur 6 f, e, d, c, b, a. On commence par insérer z dans d, e, f (2 comparaisons) puis on insère t dans la liste qui comprend f, e et éventuellement z (encore deux comparaisons). On a :



Insérer x dans la chaîne de longueur 7 dont le plus grand élément est b requiert 3 comparaisons. Enfin, insérer y dans une chaîne de longueur au plus 7 (longueur 6 si $x > b$, longueur 7 sinon) requiert également 3 comparaisons.

Le nombre final de comparaisons est donc $5 + 7 + 2 + 2 + 3 + 3 = 22 = Comp(10)$.

Il se trouve que l'algorithme 2 est optimal. Bien sûr, c'est difficile à prouver ! D'une manière générale, la fonction $Comp\_min(n)$ qui donne le nombre minimal de comparaisons nécessaires pour trier n éléments est très mal connue : on ne connaît pas d'expression exacte, juste un équivalent asymptotique en $O(n \log_2 n)$.

⋆⋆⋆⋆⋆⋆⋆

1.6 Représentation de permutations

On dit qu'une permutation p des entiers 0, 1, ..., n est représentée sous forme normale si elle est stockée dans un tableau `p` tel que `p.(i)` contienne l'image de `i` par la permutation.

Question 1

Écrire une fonction qui prend comme entrée une permutation p sous forme normale et calcule le nombre de points fixes de p.

Question 2

Écrire une fonction qui prend comme entrée une permutation p sous forme normale et calcule le nombre de cycles de p.

Question 3

On dit qu'une permutation est stockée sous forme de cycles si elle est stockée dans un tableau `c` construit comme suit : on stocke la permutation cycle par cycle, on regarde le plus petit élément de chaque cycle, on ordonne les cycles par ordre décroissant de leur plus petit élément. Par exemple, la permutation :

i	0	1	2	3	4	5	6	7
`p.(i)`	6	1	0	7	3	4	2	5

a trois cycles, l'un réduit à 1, qui est un point fixe, l'autre de longueur 3, avec 0 qui a pour image 6 qui a pour image 2 qui a pour image 0, et le troisième de longueur 4, avec 4 qui a pour image 3 qui a pour image 7 qui a pour image 5 qui a pour image 4. Les plus petits éléments de ces cycles sont respectivement 1, 0 et 3. En écrivant d'abord le cycle qui contient 3, puis celui qui contient 1, puis celui qui contient 0, on obtient le tableau `c` :

i	0	1	2	3	4	5	6	7
`c.(i)`	3	7	5	4	1	0	6	2

Écrire une fonction qui construit `c` à partir de `p`.

Question 4

Montrer que l'application qui a un tableau `p` associe le tableau `c` est bijective. Proposer un algorithme qui prend comme entrée une permutation sous forme de cycles et donne en sortie la même permutation représentée sous forme normale. Écrire la fonction correspondante.

CORRIGÉ

Question 1

Fonction élémentaire.

```
let nb_points_fixes (p:int array) : int =
  let pts = ref 0 in
  Array.iteri (fun i e -> if i = e then incr pts) p;
  !pts
```

Question 2

On peut par exemple parcourir les cycles un à un, en marquant les nombres comme « vus » au fur et à mesure ; on repère qu'un cycle se termine lorsqu'on retombe sur le premier élément du cycle courant. Pour trouver le cycle suivant, on cherche simplement un nombre qui n'a pas encore été vu. La fonction requiert de manipuler deux boucles imbriquées, la boucle intérieure parcourant le cycle et la boucle extérieure allant de cycle en cycle.

```
let nb_cycles (p:int array) : int =
  let n = Array.length p in
  let vu = Array.make n false in
  let nb_cycles = ref 0 in

  for i=0 to n-1 do
    if not vu.(i) then ( (* nouveau cycle *)
      incr nb_cycles;
      vu.(i) <- true;
      let j = ref p.(i) in
      while !j <> i do (* parcours du cycle *)
        vu.(!j) <- true;
        j := p.(!j)
      done;
    )
  done;
  !nb_cycles
```

Question 3

Un algorithme possible est de parcourir les cycles comme dans la question 2, en remplissant en même temps un tableau « min » tel que `min[j]` contienne le plus petit élément du j-ième cycle. Puis on fait un deuxième parcours des cycles pour remplir le tableau de sortie : le premier cycle à parcourir sera celui tel que `min[j]` soit maximal, et ainsi de suite jusqu'à avoir écrit tous les cycles.

```
let transforme (p:int array) : int array =
  let n = Array.length p in
  let nb_cycles = ref 0 in
  let min = Array.make n 0 in
  let vu = Array.make n false in

  for i=0 to n-1 do
    if not vu.(i) then ( (* nouveau cycle *)
      min.(!nb_cycles) <- i;
      incr nb_cycles;
      vu.(i) <- true;
      let j = ref p.(i) in
      while !j <> i do (* parcours du cycle *)
        vu.(!j) <- true;
        j := p.(!j)
      done;
    )
  done;

  let sortie = Array.make n 0 in
  let k = ref 0 in (* première case libre du tableau sortie *)
  for j=0 to !nb_cycles-1 do (* écrire les cycles successivement *)
```

```
    (* recherche du max *)
    let idx_max = ref 0 in
    for i=1 to !nb_cycles-1 do
      if min.(i) > min.(!idx_max) then
        idx_max := i
    done;

    let max = min.(!idx_max) in
    min.(!idx_max) <- 0;
    sortie.(!k) <- max;
    incr k;
    let j = ref p.(max) in
    while !j <> max do (* parcours du cycle *)
      sortie.(!k) <- !j;
      incr k;
      j := p.(!j)
    done;
  done;
  sortie
```

Question 4

La seule question pour savoir si la transformation de la question 3 est réversible est de décider comment séparer le tableau en cycles : ensuite, chaque cycle est écrit dans l'ordre (i, $p(i)$, $p(p(i))$, etc.), donc il est facile de reconstruire p. Pour faire la séparation des cycles, il suffit de remarquer que l'ordre dans lequel on a écrit les cycles, chaque cycle commençant par son plus petit élément et les cycles étant écrits dans l'ordre décroissant, assure que dans un tableau `c` donnant une permutation sous forme de cycles, $j = c(i)$ est le début d'un nouveau cycle si et seulement si j est minimal parmi $c(0)$, $c(1)$, ..., $c(i)$. Donc l'application est bien inversible.

La fonction de passage à une forme normale est en fait nettement plus simple que celle de la question 3. Il suffit de parcourir `c` en gardant en mémoire le minimum des valeurs vues jusqu'à présent, et en remarquant que $p(c(i)) = c(i+1)$ sauf en bout de cycle.

```
let inverse (c:int array) : int array =
  let n = Array.length c in
  let p = Array.make n 0 in
  let min = ref c.(0) in
  for i=0 to n-2 do (* trouver l'image de c(i) *)
    if c.(i+1) > !min
      then p.(c.(i)) <- c.(i+1)
      else (
        p.(c.(i)) <- !min;
        min := c.(i+1)
      )
  done;
  p.(c.(n-1)) <- !min; p
```

⋆⋆⋆⋆⋆⋆⋆⋆

1.7 Permutations

On considère un ensemble $E = \{0, ..., N-1\}$. On représente une permutation p de E par un tableau `p` de taille N tel que l'image d'un élément i de E est `p.(i)`.

Question 1

Écrire un algorithme qui, étant donné un tableau `p`, vérifie que `p` représente effectivement une permutation de E.

Question 2

Écrire un algorithme qui décompose une permutation en cycles.

Question 3

On ordonne les permutations par ordre lexicographique (l'ordre du dictionnaire). Par exemple, si $N = 4$, $0123 < 0213 < 1230 < 2013$. Écrire un algorithme qui associe à chaque permutation `p` la permutation suivante dans l'ordre lexicographique (quand elle existe). On pourra utiliser le plus grand entier i tel que $p(i) < p(i+1)$.

Question 4

Écrire un algorithme qui énumère toutes les permutations de E.

CORRIGÉ

Question 1

Il serait particulièrement maladroit de tester successivement si tous les nombres de 0 à $N-1$ sont dans `p`. Une solution simple consiste à marquer les images rencontrées.

```
let est_permutation (p:int array) : bool =
  let n = Array.length p in
  let vu = Array.make n false in
  Array.iter (fun e -> if 0 <= e && e < n then vu.(e) <- true) p;
  Array.fold_left (&&) true vu
```

Question 2

Comme d'habitude, la bonne idée est de procéder comme on le ferait « à la main ». On part de 0 et on regarde ses images successives par la permutation en marquant tous les entiers rencontrés. On s'arrête lorsque l'on retombe sur 0. Puis on recommence avec le premier entier non marqué.

Exemple : Soit la permutation $\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 4 & 1 & 0 & 5 & 2 & 3 \end{pmatrix}$. On trouve un premier cycle (0 4 2). Le premier élément non marqué est 1, on trouve un second cycle (1). Le premier élément non marqué est 3, on trouve le troisième et dernier cycle (3 5).

```
let afficher_cycles (p:int array) : unit =
  let n = Array.length p in
  let vu = Array.make n false in

  for i=0 to n-1 do (* recherche du premier élément non vu *)
    if not vu.(i) then (
      vu.(i) <- true;
      Printf.printf "(%d" i;
      let j = ref p.(i) in
      while !j <> i do (* parcours du cycle *)
        vu.(!j) <- true;
        Printf.printf " %d" !j;
        j := p.(!j)
      done;
      print_endline ")"
    )
  done
```

Question 3

La difficulté est de comprendre comment on calcule le suivant d'une permutation dans l'ordre lexicographique.

Exemple : le suivant de 0 1 3 2 est 0 2 1 3
le suivant de 0 2 1 3 est 0 2 3 1
le suivant de 1 2 3 0 est 1 3 0 2

Il faut donc chercher le plus grand entier i tel que $p(i)$, ..., $p(N-1)$ contienne un élément plus grand que $p(i)$. Il est très facile de voir que cela revient à chercher, comme indiqué dans l'énoncé, le plus grand entier i tel que $p(i) < p(i+1)$. Le suivant est alors obtenu en remplaçant $p(i)$ par le plus petit élément qui lui soit supérieur parmi $p.(i+1)$, ..., $p.(N-1)$. On complète la permutation en triant par ordre croissant les éléments restants.

Exemple : Pour $p = 0\ 1\ 3\ 2$, le plus grand élément tel que $p(i) < p(i+1)$ est 1. On remplace 1 par le plus petit élément plus grand que 1 parmi 3 et 2, soit 2. On trie les nombres restants (2 et 3) par ordre croissant. On obtient ainsi 0 2 1 3.

Pour éviter d'avoir une fonction de tri à écrire, on peut également marquer les éléments rencontrés dans la recherche du plus grand i tel que $p(i) < p(i+1)$.

Le programme de calcul de la permutation q suivant dans l'ordre lexicographique une permutation p fixée va suivre la méthode décrite ci-dessus. La fonction `max_croit` renvoie le plus grand entier i tel que $p(i) < p(i+1)$ si un tel entier existe, -1 sinon.

```
let max_croit (p:int array) : int =
  let n = Array.length p in
  let i = ref (n-2) in
  while !i >= 0 && p.(!i) >= p.(!i+1) do
    decr i;
  done;
  !i
```

La fonction `inf` renvoie le plus petit entier parmi `p.(k+1)`, ..., `p.(N-1)` qui est plus grand que `p.(k)`.

```
let inf (p:int array) (k:int) : int =
  let n = Array.length p in
  let min = ref max_int in
  for i=k+1 to n-1 do
    if p.(i) < !min && p.(i) > p.(k) then
      min := p.(i)
  done;
  !min
```

On a finalement la fonction calculant la permutation suivant `p` :

```
let suivante (p:int array) : (int array) =
  assert (est_permutation p);

  let n = Array.length p in
  let q = Array.make n 0 in
  let seuil = max_croit p in
  if (seuil < 0) then [||] (* pas de permutation suivante *)
  else

  (* recopier les éléments avant le seuil *)
  (* en les marquant comme vus *)
  let vu = Array.make n false in
  for i=0 to seuil-1 do
    q.(i) <- p.(i);
    vu.(p.(i)) <- true
  done;

  let inf = inf p seuil in
  q.(seuil) <- inf;
  vu.(inf) <- true;

  (* complèter q avec les éléments non vus *)
  let idx = ref (seuil+1) in
  for i=0 to n-1 do
    if not vu.(i) then (
      q.(!idx) <- i;
      vu.(i) <- true;
      incr idx
    )
  done;
  q
```

Question 4

Il suffit d'appliquer l'algorithme de la question précédente de manière itérative à partir de la permutation identité.

```
let affiche_perm (p:int array) : unit =
  let n = Array.length p in
  print_string "(";
  Array.iteri (fun i e -> if i = n-1
    then Printf.printf "%d)\n" e
    else Printf.printf "%d " e
  ) p

let enumere (n:int) : unit =
  (* Initialisation de la permutation identité *)
  let identite = Array.init n (fun i -> i) in
  let cur = ref identite in

  while !cur <> [||] do
    affiche_perm !cur;
    cur := suivante !cur
  done
```

****

2

Jouer avec les mots

« *Reconnaissance, Construction, Codage.* »

2.1 Réécriture de mots

On considère les mots écrits sur l'alphabet {a, b, A, B}, tel que w =abbaBabAA par exemple. Soit ε le mot vide. On dit que deux mots u et v sont en relation si on peut réécrire des parties de u de façon à obtenir v après une suite de transformations effectuées en suivant les règles suivantes :

$$\begin{array}{rcl} \text{aA} & \to & \varepsilon \\ \text{Aa} & \to & \varepsilon \\ \varepsilon & \to & \text{aA} \\ \varepsilon & \to & \text{Aa} \\ \text{bB} & \to & \varepsilon \\ \text{Bb} & \to & \varepsilon \\ \varepsilon & \to & \text{bB} \\ \varepsilon & \to & \text{Bb} \\ \text{aab} & \to & \text{baa} \\ \text{baa} & \to & \text{aab} \\ \text{bba} & \to & \text{abb} \\ \text{abb} & \to & \text{bba} \end{array}$$

Par exemple :

$$\begin{array}{rl} & \text{aababaabAAABBB} \\ \to & \text{aababbaaAAABBBB} \\ \to & \text{aababbABBB} \\ \to & \text{aababbABBB} \\ \to & \text{aabbbaABBB} \\ \to & \text{aa} \end{array}$$

Question 1

Montrer que cette relation est une relation d'équivalence. Montrer que aa et AA commutent avec toutes les lettres.

Question 2

Montrer que tout mot w est équivalent à un mot xyz (formé de trois mots x, y et z mis bout à bout), où : x est de la forme aa...aa ou AA...AA et de longueur paire, y est de la forme bb...bb ou BB...BB et de longueur paire, z de la forme ababa...bab, ou ba...bab, ou ab...aba, ou ba...ba, c'est-à-dire alterne les lettres a et b, commençant par a ou b et finissant par a ou b. Un mot de la forme xyz est dit canonique.

Question 3

Proposer un codage des mots canoniques sous forme de triplets d'entiers.

Question 4

Écrire une fonction `forme3 (w:string) : bool` qui prend en entrée un mot w et teste si w est un mot alternant les lettres a et b, c'est-à-dire de la forme de z.

Question 5

Écrire une fonction `ajouter_a (iu, ju, ku: int * int * int) : (int * int * int)` qui prend en entrée un mot canonique u, codé par un triplet `(iu, ju, ku)`, et donne en sortie le codage `(iv, jv, kv)` du mot $v = u$a.

Question 6

Écrire une fonction `representant_canonique (w:string) : (int * int * int)` qui prend pour entrée un mot quelconque w et donne en sortie un triplet `(i, j, k)` codant un mot canonique équivalent à w.

CORRIGÉ

Question 1

Réflexivité : il suffit de prendre une suite de transformations réduite à l'ensemble vide.
Symétrie : pour chaque règle, la transformation inverse est dans l'ensemble des règles donc toute suite de transformations peut être inversée.
Transitivité : évidente.
On a donc une relation d'équivalence.

Commutation de aa avec toutes les lettres : aa commute évidemment avec a.
aaA $\to$ a $\to$ Aaa, donc aa commute avec A.
aab $\to$ baa est une règle, donc aa commute avec b.
aaB $\to$ BbaaB $\to$ BaabB $\to$ Baa, donc aa commute avec B.

Commutation de AA avec toutes les lettres : AAa $\to$ A $\to$ aAA, donc AA commute avec a.
AA commute évidemment avec A.
AAb $\to$ AAbaaAA $\to$ AAaabAA $\to$ bAA, donc AA commute avec b.
AAB $\to$ AABaaAA $\to$ AAaaBAA $\to$ BAA, donc AA commute avec B.

Remarquons que, de façon symétrique, bb et BB commutent également avec toutes les lettres.

Question 2

Pour transformer w, on commence par bouger toutes les suites de deux lettres consécutives identiques et les mettre au début du mot, les aa et AA précédant les bb et BB, et par simplifier toutes les occurrences de aA, Aa, bB ou Bb. On se retrouve avec un mot commençant par des aa et AA, continuant avec des bb et BB, et se terminant avec un mot qui alterne des a ou A avec des b ou B. On remplace alors chaque A par AAa et chaque B par BBb, puis on ramène les AA et les BB plus au début : on se retrouve avec une suite de aa et de AA, suivie d'une suite de bb et de BB, suivie d'un mot de la forme de z. En simplifiant les aaAA, les AAaa, les bbBB et les BBbb, on obtient mot xyz de la forme requise. Notons qu'il n'est pas demandé ici de montrer l'unicité de cette écriture.

Question 3

On peut coder x par un entier i, positif si x contient des aa et négatif s'il contient des AA, et de valeur absolue égale au nombre de lettres de x. De même, y peut être codé par un entier j. On peut coder z par un entier k, positif si z commence par un a et négatif si z commence par un b, et de valeur absolue égale au nombre de lettres de z.

Question 4

Supposons la longueur de w supérieure ou égale à 1. On regarde la première lettre de w, puis on teste si les suivantes alternent, en s'arrêtant dès qu'on trouve une erreur ou qu'on a parcouru tout le mot.

```
let forme3 (w:string) : bool =
  if w.[0] <> 'a' && w.[0] <> 'b' then false else

  let last = ref w.[0] and res = ref true in
  let n = String.length w in
  for i=1 to n-1 do
    if !last = 'a' && w.[i] <> 'b' || !last = 'b' && w.[i] <> 'a'
      then res := false;
    last := w.[i]
  done;
  !res
```

Question 5

La fonction n'est pas difficile mais demande un peu d'attention pour ne pas faire d'étourderie : il est particulièrement recommandé d'écrire d'abord l'algorithme en détail. Si z est non vide et se termine par un b : `ku` strictement positif et pair ou strictement négatif et impair ; alors z s'allonge d'une lettre : `ku` augmente de 1 dans le premier cas et diminue de 1 dans le deuxième cas. Si z est non vide et se termine par un a : `ku` strictement positif et impair ou strictement négatif et pair ; alors z raccourcit d'une lettre et x change : `ku` diminue de 1 dans le premier cas et augmente de 1 dans le second cas, et `iu` augmente de 2 s'il était positif et diminue de 2 s'il était strictement négatif. Si z est vide : $\texttt{ku} = 0$; alors `ku` devient égal à 1. D'où la fonction suivante :

```
let ajouter_a (iu, ju, ku: int * int * int) : (int * int * int) =
  if ku = 0 then (iu, ju, 1) else (
    if ku > 0 && ku mod 2 = 0 then (iu, ju, ku+1)
    else if ku < 0 && ku mod 2 = 1 then (iu, ju, ku-1)
    else if ku > 0 && ku mod 2 = 1 then (
      if iu = 0 then (iu+2, ju, ku-1) else (iu-2, ju, ku-1)
    )
    else ( (* ku < 0 && ku mod 2 = 0 *)
      if iu = 0 then (iu+2, ju, ku+1) else (iu-2, ju, ku+1)
    )
  )
```

Question 6

On peut supposer qu'on dispose des fonctions `ajouter_b`, `ajouter_A` et `ajouter_B` similaires à celle de la question 5. Il est alors facile d'ajouter des lettres une par une pour construire un mot canonique pour chaque préfixe de w et finalement pour w.

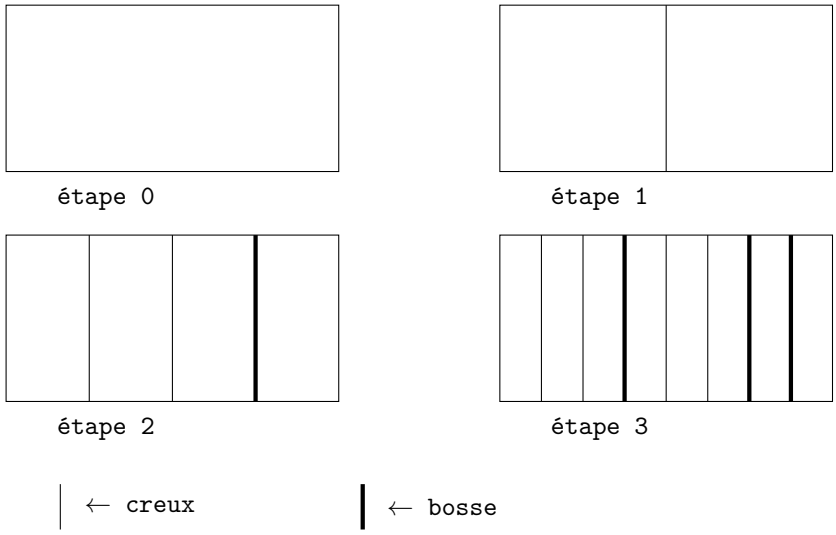
```
let representant_canonique (w:string) : (int * int * int) =
  let n = String.length w in
  let mot = ref (0, 0, 0) in
  for i=0 to n-1 do
    if w.[i] = 'a' then mot := ajouter_a !mot else
    if w.[i] = 'b' then mot := ajouter_b !mot else
    if w.[i] = 'A' then mot := ajouter_A !mot else
    mot := ajouter_B !mot
  done;
  !mot
```

La relation d'équivalence étudiée dans cet exercice correspond à un groupe donné par une permutation finie : les éléments sont les classes d'équivalence de la relation définie à la question 1, et on peut facilement vérifier que cette relation est compatible avec la concaténation. L'étude faite ici consiste en la construction d'une « structure automatique » pour le groupe, permettant de calculer un représentant distingué de la classe du produit de deux éléments donnés. L'étude de groupes automatiques est un domaine de recherche récent et actif en mathématiques.

⋆⋆⋆⋆⋆⋆⋆⋆

2.2 Pliage de papier

On prend une feuille de papier et on la plie n fois dans le sens vertical, en repliant à chaque fois la moitié droite sur la gauche. Les plis de la feuille, une fois redépliée, sont une suite de creux et de bosses. Voir figure :



Question 1

Combien y a-t-il de plis à la n-ième étape ?

Question 2

On représente chaque étape du pliage par un mot. Un creux est codé par un 0 et une bosse par un 1. Ainsi : $w_0 = \varepsilon$ (mot vide)
$w_1 = 0$
$w_2 = 001$
$w_3 = 0010011$
...
Montrer que w_i est toujours un préfixe de w_{i+1}, c'est à dire que le début de w_{i+1} coïncide avec w_i.

Question 3

Donner un algorithme de construction de w_n à partir de w_{n-1}. Écrire une fonction de calcul de w_n. On prendra comme entrée n et on renverra une liste `w` d'entiers remplie adéquatement.

Question 4

Écrire une fonction qui prend pour entrée un entier n et renvoie une liste contenant la représentation binaire de n, poids fort en tête.

Question 5

Les mots de la suite de pliage étant préfixes les uns des autres, on peut considérer le mot infini w dont ils sont tous préfixes. Proposer un algorithme qui prend pour entrée la représentation binaire de n et renvoie le n-ième bit de w.

CORRIGÉ

Question 1

Si on considère les intervalles entre les plis (y compris les côtés de la feuille), on a 1 intervalle, puis 2, puis 4, etc. ; à chaque étape, chaque intervalle est coupé en deux et le nombre d'intervalles double. Après n étapes, il y a donc 2^n intervalles, et le nombre de plis est $2^n - 1$.

Question 2

Déchirons le papier le long du pli de la première étape et débarrassons-nous de la moitié de droite : la k-ième étape de pliage du papier initial est comme la $(k-1)$-ième étape de pliage du demi-papier, et donc w_{k-1} forme la moitié gauche de w_k.

Question 3

Le pli du milieu est un creux et donc donne un 0 au milieu de w_n. Le demi-papier droit est plié, après la première étape, comme le demi-papier gauche, sauf qu'il est tourné dans l'autre sens. Soit $\mathrm{r}(w)$ le mot obtenu à partir d'un mot w en lisant les chiffres de droite à gauche, et c(w) le mot obtenu à partir de w en remplaçant chaque 0 par un 1 et chaque 1 par un 0 : on a la relation de récurrence

$$w_n = w_{n-1}\, 0\, \mathrm{c}(\mathrm{r}(w_{n-1}))$$

De cette relation, se déduit facilement la fonction permettant de construire w_n.

```
let rec motdepliage (n:int) : (int list) =
  if n = 0 then [0] else
  let pred = motdepliage (n-1) in
  let pred' = List.map (fun e -> 1-e) (List.rev pred) in
  pred @ (0::pred')
```

Question 4

Question classique et accessible à tous.

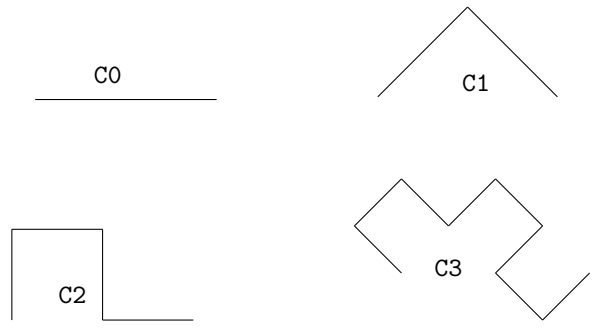
```
let binaire (n:int) : (int list) =
  let rec aux n acc =
    if n = 0 then acc
    else aux (n / 2) ((n mod 2) :: acc)
  in
  if n = 0 then [0] else aux n []
```

Question 5

On peut trouver un autre point de vue pour construire w_k par récurrence à partir de w_{k-1} : on intercale un nouveau pli entre chaque paire de plis consécutifs de w_{k-1}, et ce pli est alternativement 0 ou 1, le premier étant 0. Ainsi : $w_k = 0x1x0x...x1$, si $w_{k-1} = xxx...xx$. Donc le n-ième bit de w est facile à trouver si n est impair : c'est 0 si n est congru à 1 modulo 4 et c'est 1 si n est congru à 3 modulo 4. Si n est pair, on divise n par deux et on remarque que le n-ième bit de w_k est le $(n/2)$-ième bit de w_{k-1}, donc si $n/2$ est impair, il suffit encore une fois de tester si $n/2$ est congru à 1 ou à 3 modulo 4. Si $n/2$ est est pair, on redivise par deux, et ainsi de suite. Lorsque n est écrit en binaire, on regarde son bit le plus à droite (celui de poids le plus faible), puis le bit immédiatement à sa gauche, et ainsi de suite jusqu'à trouver un bit égal à 1. Soit x le n-ième bit de w. On a :
Si $n = 1\ 0\ 0\ ...\ 0$, alors $x = 0$.
Si $n = *\ *\ ...\ *\ 0\ 1\ 0\ 0\ ...\ 0$, alors $x = 0$.
Si $n = *\ *\ ...\ *\ 1\ 1\ 0\ 0\ ...\ 0$, alors $x = 1$.
Car un nombre se terminant par 0 1 est congru à 1 modulo 4 et un nombre se terminant par 1 1 à 3 modulo 4. Ceci donne un programme extrêmement simple.

```
let bit (nbin:int list) : int =
  let rev = List.rev nbin in
  let rec aux bin = match bin with
    | 1::1::_ -> 1
    | 1::0::_ | [1] -> 0
    | 0::q -> aux q
    | _ -> failwith "entrée invalide"
  in aux rev
```

Des études du genre de celle faite dans cet exercice peuvent servir à montrer des propriétés d'algébricité ou de transcendance du nombre étudié. C'est actuellement un domaine actif de recherche en France. On obtient une suite de courbes (C_n) approximant une courbe de Peano (c'est-à-dire une courbe qui remplit le plan), définie à partir de (w_n), en dessinant un segment de longueur $1/2^{n/2}$ pour chaque bit lu, et en tournant à droite (de $+\pi/2$) ou à gauche (de $-\pi/2$) à chaque pas, selon que le bit lu est 0 ou 1. Le premier segment est dessiné dans la direction $\pi/4$. La figure suivante illustre les premiers pas :



★★★★★★★

2.3 Plus longue sous-suite croissante

Une suite finie d'entiers $(a_i)_{1\leq i\leq n}$ est représentée par un tableau d'entiers `a` en OCaml.

On cherche la longueur de la (ou les) plus longue(s) sous-suite(s) croissante(s) (au sens large) de la suite d'entiers. Par exemple, si $n = 9$, et si les termes de la suite sont 1, 2, 6, 4, 5, 11, 9, 12, 9, la longueur cherchée est 6, et elle est atteinte par les sous-suites croissantes :

- 1, 2, 4, 5, 11, 12
- 1, 2, 4, 5, 9, 12
- 1, 2, 4, 5, 9, 9

Question 1

Écrire une fonction donnant la longueur de la plus longue sous-suite croissante de la suite `a`. Estimer le nombre de comparaisons que demande votre fonction. Pourriez-vous l'améliorer ?

Question 2

Modifiez votre fonction de manière à pouvoir afficher la plus longue (ou une des plus longues s'il y en a plusieurs de longueur maximale) sous-suites croissantes de `a`.

CORRIGÉ

Question 1

Il suffit de procéder par étapes, en ajoutant les éléments de `a` au fur et à mesure. Définissons un tableau `l` tel que `l.(p)` est la longueur de la plus longue sous-suite se terminant par `a.(p)`. Une plus longue sous-suite croissante se terminant par `a.(p)` est soit formée uniquement de `a.(p)` si pour tout $i < p$, on a `a.(i) > a.(p)`, soit obtenue en ajoutant `a.(p)` à la plus longue des plus longues sous-suites se terminant par un `a.(i)` tel que $i < p$ et `a.(i) < a.(p)`. À la fin, il ne reste plus qu'à chercher le plus grand des `l.(i)`, $1 \leq i \leq n$. Ceci nous donne la fonction suivante :

```
let longueur_sous_suite (a:int array) : int =
  let n = Array.length a in
  let l = Array.make n 0 in
  l.(0) <- 1;
  for p=1 to n-1 do
    let max = ref 0 in
    for j=0 to p-1 do
      if (a.(j) <= a.(p)) && (l.(j)+1 >= !max)
        then max := l.(j) + 1
    done;
    l.(p) <- !max
  done;
```

```
  (* Trouver le max *)
  let longueur = ref 1 in
  for i=1 to n-1 do
    if l.(i) > !longueur then
      longueur := l.(i)
  done;
  !longueur
```

On fait 2 fois $1 + 2 + ... + (n-1)$, soit $n(n-1)$ tests plus les $n-1$ derniers tests de la dernière boucle « for ». Ceci donne $(n+1)(n-1)$ tests. On peut gagner un peu de temps en classant `l` non par ordre croissant des valeurs de `p` mais par ordre croissant des valeurs de `a.(p)` et en simulant une structure dynamique pour insérer un nouvel élément sans devoir tout déplacer. Ce sera un peu lourd !

Question 2

On peut dans un premier temps créer un matrice auxiliaire `s` telle que `s.(p).(i)` contient le i-ième élément d'une plus longue sous-suite croissante se terminant par `a.(p)` (note : pas besoin d'une « marque de fin », on sait que le dernier élément est `a.(p)` et on connaît la longueur d'une telle sous-suite, c'est `l.(p)`). Il est plus malin de remarquer que si l'élément qui précède `a.(p)` dans une telle sous-suite est `a.(i)`, alors la sous-suite correspondante s'obtient en ajoutant `a.(p)` à la plus longue sous-suite qui se termine par `a.(i)`. Il suffit donc de mémoriser i. On obtient alors, en déclarant un tableau `precedent` :

```
let afficher_sous_suite (a:int array) : unit =
  let n = Array.length a in
  let l = Array.make n 0 in
  let precedent = Array.make n 0 in
  l.(0) <- 1;
  for p=1 to n-1 do
    let max = ref 0 and prec = ref 0 in
    for j=0 to p-1 do
      if (a.(j) <= a.(p)) && (l.(j)+1 >= !max)
        then (
          max := l.(j) + 1;
          prec := j
        )
    done;
    l.(p) <- !max; precedent.(p) <- !prec
  done;
  (* Trouver le max *)
  let longueur = ref 1 and pmax = ref 0 in
  for i=1 to n-1 do
    if l.(i) > !longueur then (
      longueur := l.(i);
      pmax := i
    )
  done;
```

```
Printf.printf "%d" a.(!pmax);
for i=1 to !longueur-1 do
  pmax := precedent.(!pmax);
  Printf.printf " %d" a.(!pmax);
done;
```

Cette fonction affichera une plus longue sous-suite à l'envers. L'afficher à l'endroit n'est pas difficile.

⋆⋆⋆⋆⋆⋆⋆⋆

2.4 Recherche de motifs

Un mot u est une suite $(u_1, u_2, ..., u_n)$ de lettres, l'entier n est la longueur du mot u. La suite vide correspond au mot vide, noté ε, de longueur 0. La concaténation de deux mots $u = (u_1, u_2, ..., u_n)$ et $v = (v_1, v_2, ..., v_k)$, notée uv, est simplement le mot $(u_1, u_2, ..., u_n, v_1, v_2, ..., v_k)$. Il est facile de vérifier que la concaténation est une opération associative pour laquelle le mot vide est élément neutre à gauche et à droite. Le mot v est dit facteur ou motif d'un mot u s'il existe des mots g et d tels que $u = gvd$. Si le mot g (resp. d) est la suite vide, on dit que v est un préfixe (resp. suffixe) de u.

Question 1

Modéliser la situation décrite. Écrire un algorithme qui, deux mots u et v, vérifie si v est un facteur de u. Quel est le nombre maximum de comparaisons effectuées par votre algorithme ?

Question 2

Si v n'est pas le mot vide, on note $Bord(v)$ le plus grand mot distinct de v qui soit à la fois préfixe et suffixe de v. Par exemple, si $v = abacaba$, $Bord(v) = aba$. Soit $v = (v_1, v_2, ..., v_k)$. Pour tout i entre 1 et k, on définit $B(i)$ comme étant la longueur de $Bord(v_1...v_k)$.

Utiliser cette fonction B pour écrire un nouvel algorithme vérifiant si un mot v est facteur d'un mot u. Quel est le nombre maximum de comparaisons effectuées par ce nouvel algorithme ?

Question 3

Soit $v = (v_1, v_2, ..., v_k)$ et a une lettre. On admet que $Bord(va)$ est le plus long préfixe de v qui soit dans $\{\varepsilon, Bord(v)a, Bord^2(v)a, ..., Bord^k(v)a\}$ (en posant $Bord(\varepsilon) = \varepsilon$). Déduire de ce résultat un algorithme qui, étant donné un mot $v = (v_1, v_2, ..., v_k)$, calcule successivement $B(1), ..., B(k)$.

CORRIGÉ

Question 1

La modélisation suggérée consiste à représenter les mots par des chaînes de caractères.

L'algorithme naïf, mais naturel, consiste à comparer v avec les facteurs de u de même longueur que v en commençant en position 0, puis 1, 2... jusqu'au succès ou à la fin de u.

```
let est_facteur (u:string) (v:string) : bool =
  let n = String.length u in
  let k = String.length v in

  try
    for i=0 to n-k do
      let sub = String.sub u i k in
      if String.equal sub v then
        raise Exit
    done;
    false
  with Exit -> true
```

Chaque appel à la fonction `String.equal` amène à au plus k comparaisons. Le nombre d'appels à `String.equal` étant d'au plus $(n-k+1)$, le nombre total de comparaisons effectuées par cet algorithme est au plus $k(n-k+1)$. Le maximum est atteint pour les mots de la forme $u = a^{n-1}b$ et $v = a^{k-1}b$.

Question 2

La primitive Caml `equal` compare les chaînes de caractère passées en argument octet par octet (caractère par caractère). On donne la code OCaml équivalent suivant :

```
let equal (s1:string) (s2:string) : bool =
  let n = String.length s1 in
  let k = String.length s2 in
  if n <> k then false else
  try
    for i=0 to n-1 do
      if s1.[i] <> s2.[i]
        then raise Exit
    done;
    true
  with Exit -> false
```

Supposons que l'appel à `String.equal` renvoie false lors de la comparaison entre u_{i+j-1} et v_j. On a alors la situation suivante :

$$\begin{array}{ccccc} u_i & u_{i+1} & ... & u_{i+j-2} & u_{i+j-1} \\ = & = & & = & \neq \\ v_1 & v_2 & ... & v_{j-1} & v_j \end{array}$$

On va donc ensuite comparer la chaîne de caractères $(u_{i+1}, u_{i+2}, ..., u_{i+j})$ à v. Pour que cet appel renvoie *true*, il faut en particulier que $v_1 = v_2$, ..., $v_{j-2} = v_{j-1}$. Ces égalités

signifient exactement que $Bord(v_1...v_{j-1}) = v_1...v_{j-2}$. Ainsi, si $Bord(v_1...v_{j-1}) \neq v_1...v_{j-2}$, il est inutile d'appeler `String.equal` avec la chaîne de caractères (u_{i+1}, u_{i+2}, ..., u_{i+j}). De façon identique, on voit que si $Bord(v_1...v_{j-1}) \neq v_1...v_{j-3}$, l'appel à `String.equal` avec la chaîne de caractères $(u_{i+2}, u_{i+3}, ..., u_{i+j+1})$ est inutile. En répétant le raisonnement, il apparaît que le premier appel qui ne soit pas voué à l'échec est celui où u_{i+j-1} est comparé à $v_{1+B(j-1)}$. Cette remarque conduit alors au nouvel algorithme suivant, où on suppose la fonction `b` correspondant à B définie :

```
let est_facteur_bis (u:string) (v:string) : bool =
  let n = String.length u in
  let k = String.length v in

  let i = ref 0 and j = ref 0 in
  while !i < n && !j < k do
    if u.[!i] = v.[!j] then (
      incr i; incr j
    ) else (
      if !j = 0 then incr i
      else i := b v (!j-1)
    )
  done;
  !j = k
```

Le nombre maximal de comparaisons peut être évalué comme suit. Appelons « positif » un test $u_i = v_j$ évalué à *true* et négatif un test $u_i = v_j$ évalué à $false$. Lors d'un test positif, i augmente, il y a donc au plus n tests positifs. Lors d'un test négatif, la quantité $i - j$ augmente : elle vaut 0 au commencement de l'algorithme et au plus n lorsque l'algorithme se termine. Ainsi, il y a également au plus n tests négatifs. En conclusion, ce second algorithme effectue donc au plus $2n$ comparaisons.

Question 3

On a bien sûr $B(1) = 0$ et $B(2) = 1$ si $v_1 = v_2$, 0 sinon.

Supposons calculés $B(1)$, ..., $B(h-1)$. D'après l'indication donnée dans l'énoncé, $B(h)$ est la longueur du plus long préfixe de $v_1...v_h$ qui soit dans

$$\{\varepsilon, Bord(v_1...v_{h-1})v_h, Bord^2(v_1...v_{h-1})v_h, ...\}$$

Il est clair que la longueur des mots $Bord^i(v_1...v_{h-1})v_h$ décroît (au sens large) lorsque i croît. Ainsi, il faut d'abord tester si $Bord(v_1...v_{h-1})v_h$ est un préfixe de $v_1...v_h$, puis si ce n'est pas le cas, si $Bord^2(v_1...v_{h-1})v_h$ est un préfixe de $v_1...v_h$, etc. En remarquant que $Bord(v_1...v_{h-1})v_h$ est un préfixe de $v_1...v_h$ si et seulement si $v_h = v_{1+B(h-1)}$, on est amenés naturellement à l'algorithme suivant, renvoyant le tableau $[B(1), B(2), ..., B(k)]$:

```
let calcul (v:string) : int array =
  let k = String.length v in
  let b = Array.make k 0 in

  for h=1 to k-1 do
    let j = ref b.(h-1) in
```

```
    let fini = ref false in
    while not !fini do
      (* v_{j+1} = v_h *)
      if v.[!j] = v.[h] then (
        fini := true;
        b.(h) <- !j+1
      )
      else if !j = 0 then (
        fini := true;
        b.(h) <- 0
      )
      else j := b.(!j-1)
    done;
  done;
  b
```

En considérant la quantité $h - j$, on obtient par un argument similaire à celui de la question 2 que le nombre de comparaisons effectuées est au plus $2k$.

L'indication de l'énoncé : $Bord(va)$ est le plus long préfixe de v qui soit dans $\{\varepsilon, Bord(v)a, Bord^2(v)a, ..., Bord^k(v)a\}$ se démontre comme suit.
On dit que w est un bord de v si w est à la fois un préfixe et un suffixe de v. Soit z un bord de va. Si $z \neq \varepsilon$, $z = z'a$, où z' est un bord de v. Ou bien $z' = Bord(v)$, ou bien z' est un bord de $Bord(v)$. Par induction, il vient immédiatement que z' est de la forme $Bord^i(v)$ pour un certain i tel que $1 \leq i \leq k$. Ainsi, z est dans l'ensemble $\{\varepsilon, Bord(v)a, Bord^2(v)a, ..., Bord^k(v)a\}$.
Réciproquement, tout mot de cet ensemble est clairement suffixe de va, donc un bord de va s'il est préfixe de va.

⋆⋆⋆⋆⋆⋆⋆⋆

2.5 Mots bien parenthésés

On considère une parenthèse ouvrante « (» et une parenthèse fermante «) ». Un mot parenthésé u est une suite de parenthèses ouvrantes et fermantes. Le nombre de parenthèses utilisées est appelée la longueur du mot. La suite vide correspond au mot vide, noté ε. La concaténation de deux mots parenthésés u et v, notée uv, est simplement le mot parenthésé obtenu en mettant u et v bout à bout. Ainsi, si $u = (((()($ et $v =))$, $uv = ((()()))$. Un mot parenthésé v est dit facteur gauche d'un mot parenthésé u s'il existe un mot w tel que $u = vw$. Un mot parenthésé u est bien parenthésé s'il contient autant de parenthèses ouvrantes que de parenthèses fermantes et si tout facteur gauche v de u contient au moins autant de parenthèses ouvrantes que de parenthèses fermantes.

Question 1

Modéliser la situation décrite et écrire un algorithme qui, étant donné un mot parenthésé, vérifie s'il est bien parenthésé ou non.

Question 2

Montrer que, étant donné un mot bien parenthésé non vide u, il existe un unique couple (v, w) de mots bien parenthésés tels que $u = (v)w$. Écrire un algorithme qui, étant donné le mot u, calcule les mots v et w.

Question 3

Soit $N \in \mathbb{N}$. Écrire un algorithme qui énumère tous les mots bien parenthésés de longueur au plus N.

CORRIGÉ

Question 1

Une solution est de modéliser un mot parenthésé par une chaîne de caractères ne contenant que des parenthèses.

Pour tester si un mot u est bien parenthésé, il suffit de vérifier que u contient autant d'ouvrantes que de fermantes et que tout facteur gauche de u contient plus de parenthèses ouvrantes que de fermantes. On peut pour cela gérer un compteur que l'on incrémente (resp. décrémente) lorsque l'on trouve une parenthèse ouvrante (resp. fermante). On suppose ici que u est bien formé (i.e ne contient que des parenthèses).

```
let test (u:string) : bool =
  let n = String.length u in
  let cpt = ref 0 in
  try
    for i=0 to n-1 do
      if u.[i] = '(' then incr cpt
      else decr cpt;   (* u.[i] = ')' *)

      if !cpt < 0 then raise Exit
    done;
    !cpt = 0
  with Exit -> false
```

Question 2

Montrons tout d'abord l'unicité de la décomposition proposée. Supposons qu'un mot u se décompose en $u = (v)w = (v')w$ avec $v \neq v'$. Il est clair que si v et v' sont de même longueur alors $v = v'$. Par symétrie, il suffit de traiter le cas où v' est de longueur strictement plus grande que v. Ainsi $v' = v)v''$ où v'' est un mot parenthésé éventuellement vide. Comme v est bien parenthésé, il contient autant d'ouvrantes que de fermantes. Ainsi, le préfixe $v)$ de v' contient une fermante de plus que d'ouvrantes, ce qui est en contradiction avec le fait que v' soit bien parenthésé. On a ainsi montré l'unicité de la décomposition.

Soit u un mot bien parenthésé. Pour montrer l'existence d'une décomposition, considérons le plus petit facteur gauche non vide u' de u qui contienne autant d'ouvrantes que

de fermantes (u' existe puisque u a cette propriété). Dans la fonction de la question précédente, u' correspond au plus petit facteur gauche non vide pour lequel la variable `cpt` prend la valeur 0. Ainsi, il est facile de voir que u' commence par une ouvrante et finit par une fermante : $u' = (v)$. En utilisant le fait que u est bien parenthésé, on vérifie aisément que v et w, où w est défini par $u = u'w$, sont bien parenthésés.

Le calcul de v et w repose sur le remarque que (v) est le plus petit facteur de u qui contienne autant d'ouvrantes que de fermantes.

```
let decompose (u:string) : (string * string) =
  let n = String.length u in
  let cpt = ref 0 and idx = ref 0 in

  for i=0 to n-1 do
    if u.[i] = '(' then incr cpt
    else decr cpt;

    if !cpt = 0 && !idx = 0 then (
      idx := i
    )
  done;

  let v = String.sub u 1 (!idx-1) in
  let w = String.sub u (!idx+1) (n - !idx - 1) in
  (v, w)
```

Question 3

L'idée est bien entendu d'utiliser la question précédente. Si l'on connaît tous les mots bien parenthésés de longueur au plus L, où L est une certaine constante, on obtient un mot bien parenthésé de longueur $L + 2$ en prenant un mot bien parenthésé u de longueur k, un mot bien parenthésé de longueur $L - k$ et en construisant le mot $(u)v$. Il est donc nécessaire de garder en mémoire tous les mots parenthésés déjà construits.

Nous allons ici utiliser un tableau de listes tel que la liste à l'indice i contient tous les mots bien parenthésés de longueur $2i$.

```
let enumere_mots (lmax:int) : string list array =
  let mots = Array.make (lmax+1) [] in
  mots.(0) <- [""];
  for l=0 to lmax-1 do
    for k=0 to l do
      List.iter (fun u ->
        List.iter (fun v ->
          let mot = Printf.sprintf "(%s)%s" u v in
          mots.(l+1) <- mot::mots.(l+1)
        ) mots.(l-k)
      ) mots.(k)
    done
  done; mots
```

Remarque : Le nombre de mots bien parenthésés de longueur $2n$ est égal à $\frac{1}{n+1}\binom{2n}{n}$, le n-ième nombre de Catalan.

⋆⋆⋆⋆⋆⋆⋆⋆

2.6 Plus longue sous-suite commune

Notation : Si X est une chaîne de caractères, on désignera par X_l le préfixe de X de longueur l.

Une sous-suite du mot $A = a_1a_2...a_n$ est un mot obtenu en supprimant certaines lettres. Par exemple, *bbcdd* est une sous-suite de *aaabbbcccddd*. Une sous-suite commune à deux chaînes de caractères A et B et de longueur maximale est appelée *PLSC* (Plus Longue Sous-suite Commune) de A et B. Par exemple, si $A = abaab$ et $B = aabb$, on a deux *PLSC*, *aab* et *abb*.

Question 1

Étant donné deux mots $A = a_1a_2...a_m$ et $B = b_1b_2...b_n$ de longueurs respectives m et n, on demande de calculer la longueur d'une *PLSC* de A et B :

Justifier l'équation récurrente (à compléter) :

$$long(i,j) = \max(long(i-1,j-1) + [a_i = b_j],\ long(i,j-1),\ long(i-1,j))$$

où $[a_i = b_j]$ vaut 1 si $a_i = b_j$ et 0 sinon et où $long(i,j)$ désigne la longueur d'une *PLSC* de A_i et B_j.

Écrire une fonction OCaml pour calculer la longueur d'une *PLSC* de A et B.

Question 2

On s'intéresse maintenant au calcul effectif d'une *PLSC* de A et B. Écrire une fonction OCaml qui effectue ce calcul. Comment la modifier pour obtenir toutes les *PLSC* de A et B ?

CORRIGÉ

Question 1

Soit $A = a_1a_2...a_i$ et $B = b_1b_2...b_j$. Notons $Z = z_1...z_k$ une de leurs *PLSC* :

- Si $a_i = b_j$ alors $z_k = a_i = b_j$ et Z_{k-1} est une *PLSC* de A_{i-1} et B_{j-1} ;
- Si $a_m \neq b_n$ alors : $(z_k \neq a_m) \implies Z$ est une *PLSC* de A_{m-1} et B ;
- Si $a_m \neq b_n$ alors : $(z_k \neq b_n) \implies Z$ est une *PLSC* de A et B_{n-1}.

d'où la relation complète :

$$long(i,j) = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0 \\ long(i-1,j-1) + 1 & \text{si } i,j > 0 \text{ et } a_i = b_j \\ \max(long(i,j-1),\ long(i-1,j)) & \text{si } i,j > 0 \text{ et } a_i \neq b_j \end{cases}$$

Pour écrire la fonction, il suffit de générer toutes les valeurs de $long(i, j)$ dans un ordre compatible avec la relation de récurrence. On écrit donc un algorithme de *programmation dynamique* qui remplit un tableau ligne par ligne. On a $long(i, j) =$ `long.(i).(j)`.

```
let longueur (a:string) (b:string) : int =
  let n = String.length a in
  let m = String.length b in
  let long = Array.make_matrix (n+1) (m+1) 0 in

  for i=1 to n do
    for j=1 to m do
      if a.[i-1] = b.[j-1] then
        long.(i).(j) <- long.(i-1).(j-1) + 1
      else
        long.(i).(j) <- max long.(i).(j-1) long.(i-1).(j)
    done
  done;
  long.(n).(m)
```

La complexité est en $O(nm)$.

Question 2

Il suffit de créer une matrice `chemin` que l'on modifie à chaque fois qu'on met à jour `long.(i).(j)`. Celle-ci permettra de déterminer « d'où on vient » et donc de reconstruire une ou toutes les *PLSC*. La fonction devient :

```
let plsc_chemin (a:string) (b:string) : string array array =
  let n = String.length a in
  let m = String.length b in
  let long = Array.make_matrix (n+1) (m+1) 0 in
  let chemin = Array.make_matrix (n+1) (m+1) "" in

  for i=1 to n do
    for j=1 to m do
      if a.[i-1] = b.[j-1] then (
        long.(i).(j) <- long.(i-1).(j-1) + 1;
        chemin.(i).(j) <- "  "
      )
      else if long.(i-1).(j) >= long.(i).(j-1) then (
        long.(i).(j) <- long.(i-1).(j);
        chemin.(i).(j) <- "  "
      )
      else (
        long.(i).(j) <- long.(i).(j-1);
        chemin.(i).(j) <- "  "
      )
    done
  done;
  chemin
```

Grâce à la matrice `chemin`, on peut donc retrouver une *PLSC* de A et B :

```
let plsc (a:string) (b:string) : unit =
  let n = String.length a in
  let m = String.length b in
  let chemin = plsc_chemin a b in

  let rec aux i j =
    if i=0 || j=0 then ()

    else if chemin.(i).(j) = "  " then (
      aux (i-1) (j-1);
      print_char a.[i-1]
    )
    else if chemin.(i).(j) = "  " then
      aux (i-1) j
    else
      aux i (j-1)
  in aux n m;
  print_newline ()
```

Comme le montre la figure ci-dessous représentant les matrices `longueur` et `chemin` pour $A = abcbdab$ et $B = bdcaba$, on suit les flèches (chemin grisé) et on note les lettres correspondant aux flèches obliques :

	j	0	1	2	3	4	5	6
i		b_j	b	d	c	a	b	a
0	a_i	0	0	0	0	0	0	0
1	a	0	0 ↑	0 ↑	0 ↑	1 ↖	1 ←	1 ↖
2	b	0	1 ↖	1 ←	1 ←	1 ↑	2 ↖	2 ←
3	c	0	1 ↑	1 ↑	2 ↖	2 ←	2 ↑	2 ↑
4	b	0	1 ↖	1 ↑	2 ↑	2 ↑	3 ↖	3 ←
5	d	0	1 ↑	2 ↖	2 ↑	2 ↑	3 ↑	3 ↑
6	a	0	1 ↑	2 ↑	2 ↑	3 ↖	3 ↑	4 ↖
7	b	0	1 ↖	2 ↑	2 ↑	3 ↑	4 ↖	4 ↑

Cette fonction a une complexité en $O(n+m)$ car i ou j décroît à chaque itération.

Pour afficher toutes les *PLSC*, il faudrait traiter les cas où $long(i-1,j) = long(i,j-1)$, c'est-à-dire celui où le chemin montant et celui allant à gauche permettent tous les deux d'obtenir une *PLSC* et afficher les deux chaînes correspondantes à chaque fois.

⋆⋆⋆⋆⋆⋆⋆⋆

2.7 Code correcteur de Hamming

Nous voulons transmettre des messages composés de 0 et de 1 (chiffres binaires). Comme une erreur peut se produire lors de la transmission, nous utilisons un code correcteur

de Hamming(n, m) où $n = 2^r - 1$ et $m = n - r$ qui encode tout mot V de m chiffres binaires en un mot de code C composé de n chiffres binaires. Soit $V = v_1 v_2 ... v_m$ un mot du message et $C = c_1 c_2 ... c_r$ le mot de code correspondant. C est construit de la façon suivante :

Si $2^{k-1} < i < 2^k$, $(k > 0)$ alors $c_i = v_{i-k}$

Si $i = 2^k$ avec $k = 0...r$ alors $c_i = 0$ si la somme $\sum_{\substack{j=1 \\ j \neq 2^k \text{ et } j_k = 1}}^{n} c_j$ est paire, $c_i = 1$ sinon.

Avec $j = \sum_{k=0}^{r-1} j_k 2^k$ (écriture binaire de j).
Pour simplifier, nous prendrons des tableaux d'entiers pour stocker V et C.

Question 1

Le passage du message V au message encodé C revient à un produit matrice-vecteur tel que : $C = AV$, où A est une matrice $(n \times m)$. Écrire la matrice A pour $n = 31$ et $m = 26$.
Écrire une fonction OCaml `gen_mat` qui, étant donné deux nombres n et m, génère la matrice génératrice A du code de Hamming(n, m).

Question 2

Écrire une fonction OCaml qui, étant donné un mot V de m chiffres binaires et une matrice A, renvoie le mot de code C correspondant.

Question 3

À la réception du message encodé C' (C avec peut-être une erreur de transmission), nous désirons reconstituer le message V. Pour ceci, nous vérifions que le message C' est un mot du code défini à la question précédente. Construire, pour $m = 26$ et $n = 31$, la matrice (5×31) de contrôle de parité H telle que :

$H \times C' = S$ où S est un vecteur de dimension 5 tel que :

- Si C' est un mot du code alors S est le vecteur nul
- Sinon, S donne l'écriture binaire de l'indice de l'élément erroné de C'

Écrire une fonction OCaml `cont_mat` qui, étant donné deux nombres n et r génère la matrice de contrôle de parité H du code de Hamming$(n, n - r)$.

Question 4

Écrire une fonction OCaml `syndrome` qui, étant donné le mot de code C' et une matrice de contrôle de parité H du code de Hamming$(n, n - r)$, renvoie l'indice de l'élément erroné de C' ou -1 si C' est sans erreur.

Écrire une fonction OCaml `decode` qui, étant donné le mot de code C' et une matrice de contrôle de parité H du code de Hamming$(n, n - r)$, renvoie le message original V à partir de C' après avoir au besoin corrigé l'erreur détectée.

CORRIGÉ

Question 1

Pour $n = 31$ et $m = 26$, nous obtenons la matrice A suivante :

```
1 1 0 1 1 0 1 0 1 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
```

Nous proposons la fonction `gen_mat` suivante qui utilise le fait que la matrice contient beaucoup de 0 :

```
let gen_mat (n:int) (m:int) : int array array =
  let a = Array.make_matrix n m 0 in
  let p = ref 1 and k = ref 0 in    (* p=2^k *)

  (* Traitement des i compris entre deux puissances de 2 *)
  for i=1 to n do
    for j=1 to m do    (* Cas où i < 2^k *)
      if i <> !p && j = i - !k then
        a.(i-1).(j-1) <- 1
    done;
```

```
    if i = !p then (    (* Cas où i = 2^k, on passe à i < 2^(k+1) *)
      p := !p * 2; k := !k + 1
    )
  done;

  (* Traitement des i = 2^k *)
  p := 1; k := !k - 1;
  for i=0 to !k do
    let d = ref 1 and c = ref 0 in    (* d = 2^c *)
    for j=1 to n do
      if j = !d then (
        d := !d * 2; c := !c + 1
      )
      else if j mod (2 * !p) >= !p then
        a.(!p-1).(j - !c - 1) <- 1
    done;
    p := !p * 2
  done;
  a
```

Question 2

C'est un simple produit matrice-vecteur.

```
let encode (v:int array) (a:int array array) : int array =
  let n = Array.length a in
  let m = Array.length a.(0) in
  let c = Array.make n 0 in

  for i=0 to n-1 do
    for j=0 to m-1 do
      c.(i) <- (c.(i) + a.(i).(j) * v.(j)) mod 2
    done
  done;
  c
```

Question 3

La matrice (5×31) de contrôle de parité H pour $m = 26$ et $n = 31$ est :

```
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

La fonction `cont_mat` qui, étant donné deux nombres n et r génère la matrice de contrôle de parité H du code de Hamming$(n, n-r)$ peut s'écrire :

```
let cont_mat (n:int) (r:int) : int array array =
  let h = Array.make_matrix r n 0 in
  let p = ref 1 in

  for i=1 to r do
    for j=1 to n do
      if j mod (2 * !p) >= !p then
        h.(i-1).(j-1) <- 1
    done;
    p := !p * 2
  done;
  h
```

Question 4

Nous présentons ici une fonction *syndrome* qui, étant donné le mot de code C' et une matrice de contrôle de parité H du code de Hamming$(n, n-r)$, renvoie l'indice de l'élément erroné ou -1 si C' est sans erreur.

Nous avons là encore un simple produit matrice-vecteur :

```
let syndrome (c':int array) (h:int array array) : int =
  let n = Array.length c' in
  let r = Array.length h in
  let s = Array.make r 0 in

  for i=0 to r-1 do
    for j=0 to n-1 do
      s.(i) <- (s.(i) + h.(i).(j) * c'.(j)) mod 2
    done
  done;

  let d = ref 0 in let p = ref 1 in (* Conversion *)
  for i=0 to r-1 do
    d := !d + s.(i) * !p;
    p := 2 * !p;
  done;
  !d-1
```

La fonction `decode` définie dans l'énoncé peut s'écrire comme suit.

```
let decode (c':int array) (h:int array array) : int array =
  (* Correction d'erreur *)
  let d = syndrome c' h in
  if d <> (-1) then c'.(d) <- 1 - c'.(d);

  (* Décodage *)
  let n = Array.length c' in
  let m = n - Array.length h in
  let message = Array.make m 0 in
```

```
let p = ref 1 and r = ref 0 in
for i=1 to n do
  if i <> !p then
    message.(i - !r - 1) <- c'.(i-1)
  else
    ( p := !p * 2; r := !r + 1 )
done;
message
```

⋆⋆⋆⋆⋆⋆⋆⋆

2.8 Répétition d'un motif

Nous disposons d'un alphabet Σ. Nous nous intéressons aux mots ne possédant pas deux facteurs consécutifs égaux, autrement dit les mots sans facteurs carrés : ils ne peuvent pas s'écrire rs^2t avec $|s| \geq 1$.

Question 1

Montrer que si Σ est réduit à deux éléments alors tout mot de quatre lettres ou plus possède un facteur carré.

Question 2

Écrire une fonction booléenne qui renvoie `true` si le mot entré ne comporte pas de facteur carré et `false` sinon.

Préciser le nombre maximum d'itérations.

Question 3

On considère l'alphabet $\Sigma = \{0, 1\}$ et le morphisme σ défini par $\sigma(0) = 01$ et $\sigma(1) = 10$.

Nous avons $\sigma(mm') = \sigma(m)\sigma(m')$ où m et m' sont deux mots et mm' est la concaténation de m et m'.

Nous pouvons ainsi générer la suite : $0 \to 01 \to 0110 \to 01101001 \to ...$ en appliquant σ successivement. Le mot infini obtenu est appelé la suite de Thue-Morse.

Écrire une fonction permettant de générer une telle suite.

Montrer qu'il est impossible d'avoir une sous-suite de la forme $\omega\omega x$ où ω est un mot et x la première lettre de ω.

Question 4

Construire la suite des nombres de 1 compris entre deux 0 de la suite précédente.

Montrer que l'alphabet $\{0, 1, 2\}$ est suffisant pour l'écrire, et qu'elle ne possède pas de facteurs carrés.

Déterminer le morphisme correspondant.

CORRIGÉ

Question 1

Notons $\Sigma = \{a, b\}$. Tout mot de plus de trois lettres sur l'alphabet Σ et sans facteur carré possède l'un des deux préfixes suivants : *aba* ou *bab* (les six autres facteurs possibles comportent tous un carré). S'il possède au moins quatre lettres, le préfixe suivant sera l'un des quatre mots : *abaa*, *abab*, *baba*, *babb*, qui tous possèdent un facteur carré.

Question 2

Un facteur propre de $s = s_0...s_{n-1}$ est de la forme $s_i s_{i+1}...s_{j-1}$ avec $0 \leq i < j \leq n$. On les examine tous à la recherche d'un facteur carré.

On commence par définir une fonction testant si un facteur donné est un carré :

```
let est_carre (facteur:string) : bool =
  let n = String.length facteur in
  if n mod 2 <> 0 then false else
  let p = n/2 in
  try
    for i=0 to p-1 do
      if facteur.[i] <> facteur.[p+i] then
        raise Exit
    done;
    true
  with Exit -> false
```

Puis on teste tous les facteurs possibles :

```
let contient_carre (mot:string) : bool =
  let n = String.length mot in
  try
    for i=0 to n-1 do
      for j=i+1 to n do
        let fac = String.sub mot i (j-i) in
        if est_carre fac then
          raise Exit
      done
    done;
    false
  with Exit -> true
```

On peut majorer le nombre d'itérations par :

$$\sum_{i=0}^{n-1} \sum_{j=i+1}^{n} \frac{j-i}{2} = \frac{1}{2} \sum_{i=0}^{n-i} j = \frac{1}{4} \sum_{i=0}^{n-1} (n-i)(n-i+1) = \frac{1}{4} \sum_{i=1}^{n} i(i+1) = \frac{n(n+1)(n+2)}{12}$$

On peut réduire la complexité en remplaçant la comparaison effectuée dans `est_carre` par une comparaison de deux nombres. Il s'agit de considérer les lettres comme des chiffres et le cardinal de l'alphabet comme la base.

Si on se limite à une longueur de répétition équivalente au plus grand entier machine, le nombre d'itérations devient de l'ordre de n^2 au lieu de n^3.

Question 3

Il n'est pas utile de repartir du début. Chaque caractère d'indice i engendre deux nouveaux caractères en fin de mot (indices $2i$ et $2i+1$).

```
(* génère les 2n premiers chiffres de la suite *)
let thue_morse (n:int) : int array =
  let mot = Array.make (2*n) 0 in
  for i=0 to n-1 do
    if mot.(i) = 0 then (
      mot.(2*i) <- 0;
      mot.(2*i + 1) <- 1
    ) else (
      mot.(2*i) <- 1;
      mot.(2*i + 1) <- 0
    )
  done;
  mot
```

Montrons qu'il est impossible d'avoir une sous-suite de la forme $\omega\omega x$ où ω est un mot et x la première lettre de ω.

Nous pouvons rapidement vérifier que 111 et 000 n'ont pas d'antécédents.

Supposons que $\omega\omega x$ soit le plus petit mot possible de cette forme. Il suffit de rechercher les antécédents possibles et voir que ceux-ci sont de la même forme ou inexistants, ce qui contredit notre hypothèse.

Nous pouvons remarquer que cette suite de $\{0,1\}$ est identique à celle obtenue en regardant la parité du nombre de 1 dans l'écriture binaire des entiers :

entier	0	1	2	3	4	5	6
parité	0	1	1	0	1	0	0

Question 4

L'alphabet $\Sigma = \{0,1,2\}$ est suffisant car on ne peut avoir plus de deux 1 consécutifs, on aurait sinon un facteur de la forme $\omega\omega x$.

Supposons que la suite possède deux facteurs consécutifs égaux. Ces derniers ont pour origine un facteur de la forme $0\nu0\nu0$ où $\nu \in \Sigma^*$, autrement dit de la forme $\omega\omega x$.

Nous considérons $\Sigma = \{0,1,2\}$ et le morphisme ϕ tel que :
$0 \to (010 \to 0101) \to 1...$
et
$1 \to (010 \to 011001) \to 21...$
et
$2 \to (0110 \to 01101001) \to 210...$

⋆⋆⋆⋆⋆⋆⋆⋆

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

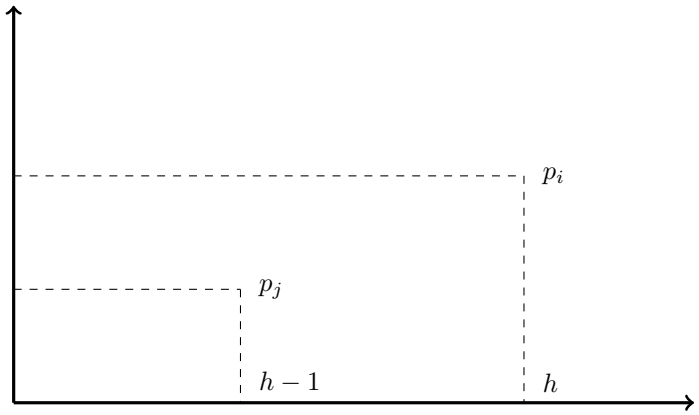
Réflexivité, antisymétrie et transitivité sont immédiates.

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 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 de 1 tonne
- utiliser 1 camion de 2 tonnes et 4 camions de 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 local 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) : int list =
  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.

⋆⋆⋆⋆⋆⋆⋆⋆

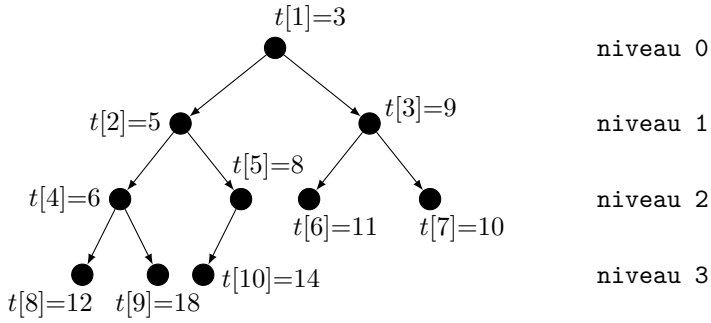
4

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 fonction 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.

⋆⋆⋆⋆⋆⋆⋆⋆

4.2 Arbre bicolore

Question 1

Un arbre binaire étiqueté est défini par une racine, des nœuds et des feuilles. Un nœud est défini par un père, un fils gauche, un fils droit, une étiquette (entier positif) et une couleur (noir ou rouge). La racine est un nœud sans père et une feuille est un nœud vide (autrement dit, fils vide d'un nœud interne).

Expliquer quelle structure de données utiliser pour représenter un arbre binaire étiqueté en OCaml.

Question 2

Un arbre binaire est de recherche (ABR) si chaque nœud est tel que : tout nœud de sa sous arborescence droite ne contient que des valeurs d'étiquettes supérieures et tout nœud de sa sous-arborescence gauche des valeurs d'étiquettes inférieures.

Donner une fonction permettant d'insérer une nouvelle valeur en conservant la structure d'ABR.

Question 3

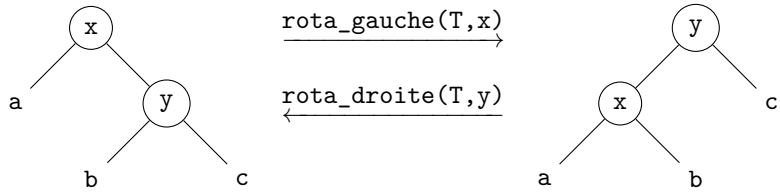
Un arbre bicolore est un arbre binaire de recherche qui vérifie les propriétés suivantes : chaque nœud est soit noir, soit rouge, chaque feuille est considérée comme noire, si un nœud est rouge alors ses deux fils sont noirs et tous les chemins d'un nœud (x) à une feuille de sa descendance contiennent le même nombre $\mathbf{pn}(x)$ de nœuds noirs (non-compris le nœud lui-même). Ce nombre est appelé la **profondeur noire** du nœud.

Nous définissons la **profondeur noire d'un arbre bicolore** comme étant la profondeur noire de sa racine.

Montrer que la profondeur (longueur de la plus longue branche, *racine* → *feuille*) d'un arbre bicolore composé de n nœuds internes (i.e qui ne sont pas des feuilles) est d'au plus $2\log_2(n+1)$.

Question 4

Nous désirons effectuer les rotations suivantes, où T représente l'arbre :



Écrire la fonction `rota_gauche`.

Montrer que cette fonction préserve la structure d'arbre binaire de recherche. En est-il de même de la structure d'arbre rouge-noir ?

Question 5

Comment insérer un nouvel élément dans un arbre bicolore en conservant sa structure ?

CORRIGÉ

Question 1

On représente un arbre bicolore avec le type récursif suivant :

```
type arn =
  | Nil
  | R of arn * int * arn
  | N of arn * int * arn
```

Question 2

Pour insérer un nouvel élément dans un arbre binaire de recherche en conservant la propriété fondamentale des ABR, nous parcourons l'arborescence en partant de la racine avec comme critère de choix pour le nœud suivant le fils droit si l'étiquette du nouvel élément est plus grande que celle du nœud courant et le fils gauche sinon.

En OCaml, on peut écrire :

```
let rec insere_arn (t:arn) (e1:int) : arn = match t with
  | Nil -> R (Nil, e1, Nil)
  | R (fg, e2, fd) ->
```

```
    if e1 > e2 then  R (fg, e2, insere_arn fd e1)
    else R (insere_arn fg e1, e2, fd)
  | N (fg, e2, fd) ->
    if e1 > e2 then  R (fg, e2, insere_arn fd e1)
    else R (insere_arn fg e1, e2, fd)
```

Question 3

L'arbre bicolore minimal pour une profondeur noire pn donnée n'est composé que de nœuds noirs. Comme chaque branche partant de la racine a exactement pn nœuds noirs, nous avons un arbre complet équilibré de profondeur pn. Chaque niveau i de cet arbre avant tout binaire contient 2^i nœuds.

Ainsi, un arbre bicolore de profondeur noire pn est composé d'au moins $2^{pn} - 1$ nœuds noirs.

Si n est le nombre de nœuds d'un arbre bicolore de profondeur noire pn, nous avons

$$n \geq 2^{pn} - 1$$

autrement dit,

$$\log_2(n+1) \geq pn$$

Si l'on tient compte du fait que, sur une branche, au plus un nœud sur deux est rouge, la profondeur d'un arbre bicolore de profondeur noire pn est au plus $2pn$ c'est à dire d'au plus $2\log_2(n+1)$.

Question 4

Pour alléger les fonctions et puisque la coloration n'apporte rien ici, on définit un type d'arbre binaire sans couleurs :

```
type abr = Nil | Noeud of abr * int * abr
```

On propose les fonctions de rotations gauche et droite autour de la racine de `t` (la transformation n'est pas effectuée si elle est impossible) :

```
let rotate_right (t:abr) : abr = match t with
  | Noeud(Noeud(t1, u, t2), v, t3) -> Noeud(t1, u , Noeud(t2, v, t3))
  | _ -> t

let rotate_left (t:abr) : abr = match t with
  | Noeud(t1, u, Noeud(t2, v, t3)) -> Noeud (Noeud(t1, u, t2), v, t3)
  | _ -> t
```

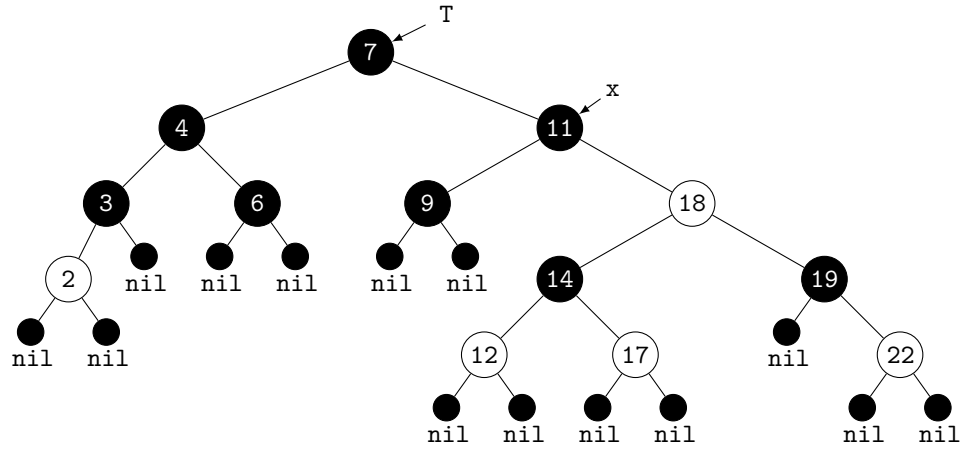
Si l'on souhaite effectuer une rotation autour d'un nœud en particulier, il suffit de descendre jusqu'à celui-ci de manière récursive puis de le remplacer, similairement à ce qui est fait pour l'insertion :

```
let rec rotate_node_left (t:abr) (x:int) : abr = match t with
  | Noeud(t1, y, t2) -> if y = x then rotate_left t
    else if x < y then Noeud(rotate_node_left t1 x, y, t2)
    else Noeud(t1, y, rotate_node_left t2 x)
  | _ -> t
```

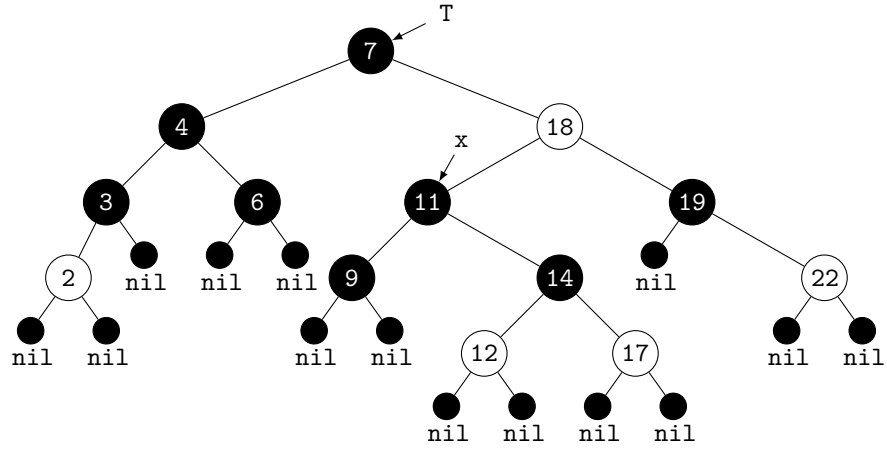
La fonction `rotate_node_right` s'écrit de manière analogue.

La structure d'arbre binaire de recherche est conservée : x et y sont du même côté par rapport au père de x et les étiquettes de la sous-arborescence gauche de y sont supérieures à celle de x et inférieures à celle de y, elle peut devenir la sous-arborescence droite de x sans que l'arbre ne perde sa propriété d'arbre binaire de recherche.

En revanche, la propriété d'arbre bicolore n'est pas forcément conservée. Si nous appliquons l'algorithme précédent à l'arbre ci-dessous où les nœuds noirs sont colorés en noir et les nœuds rouges sont représentés en blanc :



Nous obtenons l'arbre suivant :

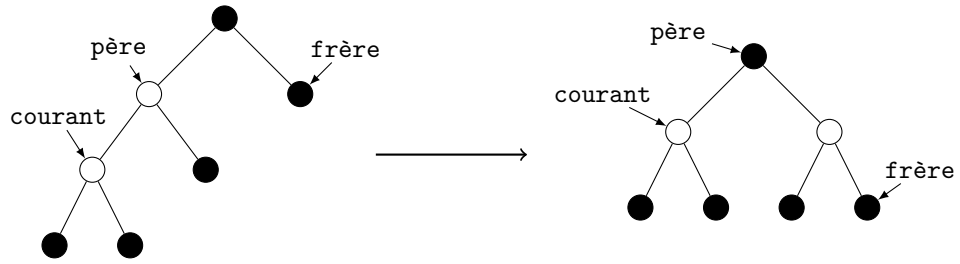


Cet arbre ne vérifie plus la propriété d'arbre bicolore, le nombre de nœuds noirs n'est plus le même sur toutes les branches partant de la racine.

Question 5

L'insertion dans un arbre bicolore peut se faire selon le schéma suivant :

On insère le nouvel élément dans l'ABR en utilisant la fonction de la question 2 ; on obtient donc un arbre binaire de recherche.
Pour que cet arbre vérifie la propriété de bicolore, nous donnons à ce nouveau nœud la couleur rouge. Il devient le nœud courant. Si son père est noir, l'arbre est bicolore et c'est terminé. Sinon, si le frère de ce père est rouge, ce père et son frère deviennent noirs et leur père devient rouge. On recommence avec le père commun. Par contre, si le frère du père du nœud courant est noir, il y a au moins un élément de plus du côté du nœud courant. Dans ce cas, nous plaçons le nœud courant et son père suivant le même type de filiation que le père et le grand-père : le père devient noir et le grand-père rouge puis nous effectuons une rotation dans le sens du père vers son grand père.
On représente ici l'une des quatre situations où une rotation est nécessaire :



Nous obtenons ainsi les fonctions OCaml suivantes où l'on n'effectue pas de rotation de manière « explicite » mais où on utilise une fonction de *pattern-matching* pour gérer les cas où deux nœuds rouges se suivent :

```
let corrige_rouge (t:arn) : arn = match t with
  | N (R (R (a, x, b), y, c), z, d)
  | N (R (a, x, R (b, y, c)), z, d)
  | N (a, x, R (R (b, y, c), z, d))
  | N (a, x, R (b, y, R (c, z, d)))
    -> R (N (a, x, b), y, N (c, z, d))
  | t -> t

let rec insere_aux (t:arn) (x:int) : arn =
  match t with
  | Nil -> R (Nil, x, Nil)
  | R (fg, y, fd) ->
    if x = y then t
    else if x > y then corrige_rouge (R(fg, y, insere_aux fd x))
    else corrige_rouge (R(insere_aux fg x, y, fd))
  | N (fg, y, fd) ->
    if x = y then t
    else if x > y then corrige_rouge (N(fg, y, insere_aux fd x))
    else corrige_rouge (N(insere_aux fg x, y, fd))
```

Notons que pour éviter de répéter deux fois la même chose, on pourrait définir une fonction constructeur comme suit :

```
let cons (t:arn) (fg:arn) (x:int) (fd:arn) : arn = match t with
  | N _ -> N(fg, x, fd) | R _ -> R(fg, x, fd ) | _ -> t
```

Nous disposons d'un arbre binaire de recherche dont la profondeur est d'au plus $2\log_2(n+1)$, avec des algorithmes d'insertion et de suppression qui parcourent un nombre donné de fois une branche de l'arborescence. Nous disposons ainsi d'une structure de données où toutes les opérations peuvent se faire en un temps logarithmique suivant la taille de cette base de données.

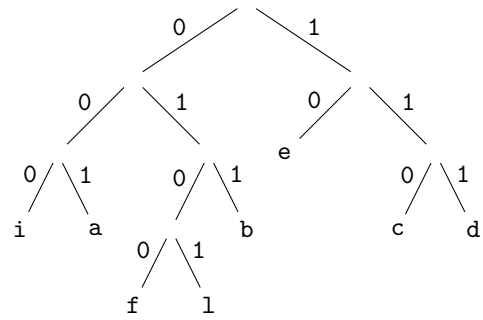
⋆⋆⋆⋆⋆⋆⋆⋆

4.3 Code préfixé : codage-décodage

Le code est stocké dans un arbre binaire de la manière suivante :

- on marque les arcs « fils gauche » avec des 0 et les arcs « fils droit » avec des 1.
- le chemin de la racine au caractère nous donne une suite de 0 et de 1 qui définit le code du caractère.

Exemple :



Question 1

Décoder le message suivant : 0100001110000010110.

Quelle est la particularité de ce codage ?

Question 2

On suppose l'arbre de codage connu. Les messages codés sont stockés dans des listes.

Écrire une fonction de décodage en précisant les structures de données utilisées.

Question 3

Encoder le mot « difficile ».

Écrire une fonction d'encodage en précisant les structures de données utilisées.

Question 4

Quel est l'intérêt de ce type de codage par rapport aux codages à longueur fixe comme le code ASCII ?

CORRIGÉ

Question 1

Le mot encodé est « facile » :

0100	001	110	000	0101	10
f	a	c	i	l	e

Nous constatons que les codes associés aux caractères n'ont pas tous la même longueur. Or, nous sommes capables de décoder un mot sans aucune ambiguïté, sans nous soucier de la longueur de chaque code. Aucun code n'est préfixe de l'autre, aucun caractère ne se trouve sur un nœud interne de l'arborescence, ils sont tous placés sur une feuille. Si l'on considère un code à longueur fixe comme le code ASCII, nous obtenons un arbre binaire complet équilibré.

Question 2

On peut représenter un arbre binaire par le type :

```
type arbre = Noeud of arbre * arbre | Feuille of char
```

Ce qui donne l'arbre suivant pour l'exemple :

```
let arbre1 =
Noeud(
  Noeud(
    Noeud(Feuille 'i', Feuille 'a'),
    Noeud(
      Noeud(Feuille 'f', Feuille 'l'),
      Feuille 'b'
    )
  ),
  Noeud(
    Feuille 'e',
    Noeud(Feuille 'c', Feuille 'd')
  )
)
```

Pour décoder le message, on parcourt l'arbre en lisant le message encodé caractère par caractère :

```
let decode (message:string) (a:arbre) : string =
  let n = String.length message in
  (* le message contient au plus n caractères *)
  let buf = Buffer.create n in

  let rec parcours i noeud =
    match noeud with
    | Feuille x -> Buffer.add_char buf x;
      if i < n then parcours i a else Buffer.contents buf
    | Noeud (fg, fd) -> if message.[i] = '0' then
      parcours (i+1) fg else parcours (i+1) fd
  in parcours 0 a
```

Question 3

Compte tenu du codage donné dans l'énoncé, le mot « difficile » devient :

d	i	f	f	i	c	i	l	e
111	000	0100	0100	000	110	000	0101	10

Une possibilité est de parcourir l'arbre et d'ajouter tous les codes à une table de hachage puis de parcourir le mot à encoder en cherchant les codes correspondants aux caractères dans la table :

```
let list_to_string (l:string list) : string =
  let n = List.length l in
  let buf = Buffer.create n in
  List.iter (Buffer.add_string buf) l;
  Buffer.contents buf

let encode (message:string) (a:arbre) : string =
  let h = Hashtbl.create 8 in
  let rec remplir a acc = match a with
    | Feuille x -> let code = list_to_string (List.rev acc) in
        Hashtbl.add h x code
    | Noeud (fg, fd) -> remplir fg ("0"::acc); remplir fd ("1"::acc)
  in remplir a [];

  let n = String.length message in
  let rec construire i acc =
    if i = n then list_to_string (List.rev acc)
    else let code = Hashtbl.find h message.[i] in
      construire (i+1) (code::acc)
  in construire 0 []
```

Ce type d'encodage est très intéressant pour la compression sans perte d'information. Dans la plupart des fichiers, le nombre d'occurrences de chaque caractère très variable : certains caractères sont plus fréquents que d'autres. Il est donc intéressant de donner aux plus fréquents les codes les plus courts afin de réduire la taille du fichier. Le code de Huffman (voir exercice sur le sujet) entre dans cette catégorie ; il est même optimal en ce qui concerne la compression par un code préfixé.

Pour stocker de la façon la plus compacte possible un message codé avec un code préfixé, nous pouvons découper le message en morceaux de 32 bits (en admettant que les entiers sont codés sur 32 bits) et stocker les entiers correspondants dans un tableau :

```
let string_to_int (message:string) : int =
  let k = ref (Int.shift_left 1 31) in  (* 2^31 *)
  let n= String.length message in

  let s = ref 0 in
  for i=0 to n-1 do
    if message.[i] = '1' then s := !s + !k;
    k := !k /2
  done;!s
```

```
let compact (message:string) : int list =
  let n = String.length message in

  let rec aux i acc =
    let len = if i=n/32 then n mod 32 else 32 in
    let s = String.sub message (32*i) len in
    let num = string_to_int s in
    if i=n/32 then num::acc else aux (i+1) (num::acc)
  in List.rev (aux 0 [])
```

⋆⋆⋆⋆⋆⋆⋆⋆

4.4 Génération d'un code de Huffman

En machine, à chaque caractère correspond un code binaire formé de 0 et de 1. Notre but est de construire une table de codage. Le codage de Huffman prend en compte les fréquences d'apparition des caractères. Nous allons étudier ici comment ce code est construit.

La structure de données utilisée pour la construction d'un tel code est celle d'arbre binaire : une feuille représente un caractère et à chaque sous-arbre est associé un poids correspondant à la somme des fréquences de ses feuilles.

Question 1

On désire fusionner deux arbres binaires en un seul de la manière suivante : on crée un nouvel arbre, de poids la somme des poids des deux arbres et dont la racine a pour fils gauche l'arbre de poids le plus faible et pour fils droit l'autre arbre.

Écrire une fonction de fusion des deux arbres binaires. Préciser la structure de données adoptée.

Question 2

Après l'analyse d'un texte écrit avec l'alphabet {a, b, c, d, e, f, g, h}, nous obtenons pour chaque lettre une fréquence d'apparition. À partir de ces fréquences, nous construisons 8 arbres correspondant aux 8 caractères de l'alphabet ; chaque arbre correspond à une feuille avec comme poids la fréquence de chaque caractère.

Nous fusionnons ensuite successivement les arbres de poids les plus faibles jusqu'à n'obtenir qu'un seul arbre.

Illustrer le fonctionnement de cet algorithme avec l'exemple suivant :

lettre	a	b	c	d	e	f	g	h
fréquence	25	12	10	8	27	10	5	3

Question 3

Nous déduisons un codage pour chaque caractère avec l'arbre obtenu : on marque les branches gauches avec des 0 et les branches droites avec des 1. Le chemin de la racine

à la feuille correspondant à un caractère nous donne une suite de 0 et de 1 qui définit le code du caractère.

Quelle est, dans le pire cas, la plus grande longueur possible pour le code d'un caractère ? Quelle structure de données peut-on utiliser pour stocker ces codes ?

Écrire une fonction OCaml permettant de construire le code correspondant pour chaque caractère.

CORRIGÉ

Question 1

On représente un arbre binaire avec le type :

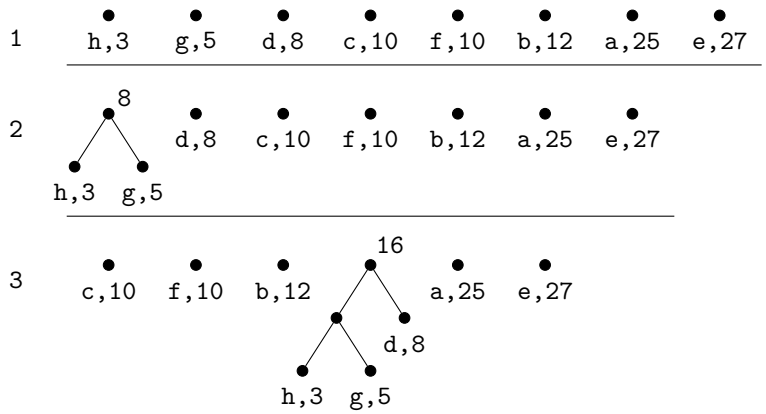
```
type arbre =
  | Noeud of arbre * int * arbre
  | Feuille of char
```

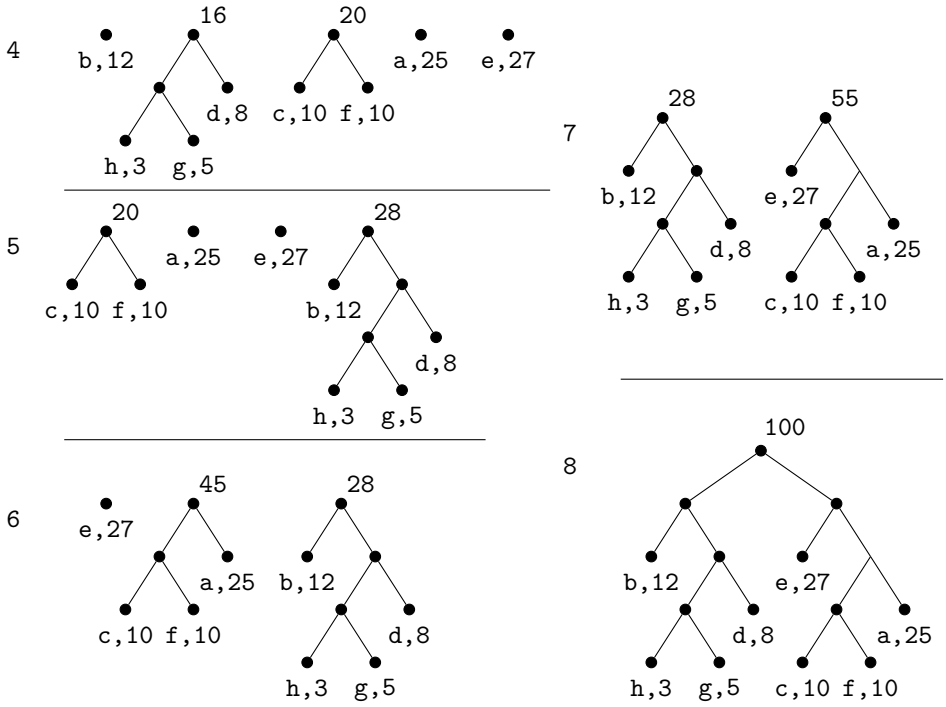
On propose la fonction de fusion suivante, où `occ` est un tableau des occurrences des lettres dans l'ordre et où on suppose que `a1` est de poids plus petit que `a2` :

```
let fusion (a1:arbre) (a2:arbre) (occ:int array) = match a1, a2 with
  | Noeud (_, p1, _), Noeud (_, p2, _) -> Noeud (a2, p1 + p2, a1)
  | Noeud (_, p1, _), Feuille c | Feuille c, Noeud (_, p1, _)
      -> let idx = int_of_char c - 97 in
      let p2 = occ.(idx) in Noeud (a1, p1 + p2, a2)
  | Feuille c1, Feuille c2 -> let i1 = int_of_char c1 - 97 in
    let i2 = int_of_char c2 - 97 in
    Noeud(a1, occ.(i1) + occ.(i2), a2)
```

Question 2

Notons qu'une manière efficace de construire un tel arbre à partir de l'ensemble de feuilles est d'utiliser un tas et de successivement fusionner les deux éléments les plus petits (situés à la racine) puis de réinsérer l'arbre obtenu dans le tas. On pourra se référer au premier exercice de ce chapitre pour l'implémentation en OCaml.





Question 3

Le pire cas se présente lorsque l'arborescence obtenue est composée d'une seule branche portant tous les caractères. La longueur du plus grand code possible pour un caractère est égale au cardinal de l'alphabet -1.

Une manière efficace de stocker les codes pour faciliter l'encodage de messages est d'utiliser une table de hachage.

De manière analogue à ce qui a été fait dans l'exercice précédent, on peut parcourir l'arbre et ajouter tous les codes à la table :

```
let list_to_string (l:string list) : string =
  let n = List.length l in
  let buf = Buffer.create n in
  List.iter (Buffer.add_string buf) l;
  Buffer.contents buf

let stocker_codes (a:arbre) : (char, string) Hashtbl.t =
  let h = Hashtbl.create 8 in
  let rec remplir a acc = match a with
    | Feuille x -> let code = list_to_string (List.rev acc) in
        Hashtbl.add h x code
    | Noeud (fg, _, fd) -> remplir fg ("0"::acc); remplir fd ("1"::acc)
  in remplir a []; h
```

⋆⋆⋆⋆⋆⋆⋆⋆

4.5 Tas binomial

Un arbre étiqueté est défini par une racine, des nœuds et des feuilles. Un nœud est défini par un père, un fils aîné, un frère, une étiquette (entier positif). La racine est un nœud sans père et une feuille un nœud sans fils.

Question 1

Expliquer comment représenter un arbre étiqueté en OCaml.

Question 2

Un arbre binomial est défini par récurrence : l'arbre binomial d'ordre 0 ne contient qu'un seul nœud, l'arbre binomial d'ordre k est construit à partir de deux arbres binomiaux d'ordre $k-1$ en plaçant la racine de l'un comme fils aîné de la racine de l'autre et l'ancien fils aîné devant le frère.

Montrer que la racine d'un arbre binomial est de degré k (nombre de fils).

Montrer que si les fils de la racine d'un arbre binomial sont numérotés de $k-1$ à 0 en partant du fils aîné alors le fils i est un arbre binomial d'ordre i.

Montrer enfin qu'un arbre binomial d'ordre k possède 2^k nœuds et que la plus longue branche contient k nœuds.

Question 3

Donner une fonction de fusion de deux arbres binomiaux d'ordre k en un seul d'ordre $k+1$.

Question 4

Un tas binomial est une liste d'arbres binomiaux telle que : il y a au plus un arbre binomial d'un degré donné et dans chaque arbre l'étiquette d'un nœud est supérieure ou égale aux étiquettes de ses parents.

Comment stocker un tas binomial ?

Combien d'arbres binomiaux composent un tas binomial de n étiquettes ?

Donner une fonction unifiant deux tas binomiaux en une liste ordonnée par ordre croissant (il peut y avoir deux arbres binomiaux de même ordre).

Question 5

Donner une fonction de fusion de deux tas binomiaux en un seul.

Donner une fonction permettant d'extraire la plus petite étiquette d'un tas binomial.

CORRIGÉ

Question 1

On représente chaque nœud de l'arbre par un tuple contenant l'ordre de l'arbre dont le nœud est racine, l'étiquette du nœud et une liste de tous ses fils directs :

```
type arbre = Noeud of int * int * arbre list
```

Question 2

Le fait que la racine d'un arbre binomial d'ordre k soit de degré k se montre par récurrence directement à partir de la définition donnée dans l'énoncé.

Il en est de même pour le fait que si les fils de la racine d'un arbre binomial sont numérotés de $k-1$ à 0 en partant du fils aîné alors le fils i est un arbre binomial d'ordre i. On ajoute un arbre d'ordre $k-1$ en fils aîné d'un arbre d'ordre $k-1$ pour construire un arbre d'ordre k.

Enfin, il est aisé de voir que si un arbre d'ordre $k-1$ possède 2^{k-1} nœuds alors un arbre d'ordre k en possède $2^{k-1}+2^{k-1}=2^k$.

Le fait que la plus longue branche contienne k nœuds se montre aussi par récurrence.

Question 3

On donne la fonction de fusion suivante :

```
let fusion (a:arbre) (b:arbre) : arbre =
  let Noeud(ordre_a, val_a, fils_a) = a in
  let Noeud(ordre_b, val_b, fils_b) = b in
  if val_a > val_b then Noeud(ordre_b+1, val_b, a::fils_b)
  else Noeud(ordre_a+1, val_a, b::fils_a)
```

Question 4

Il suffit de considérer une liste d'arbres binomiaux.

Un arbre binomial possède une puissance de deux comme nombre d'étiquettes. La décomposition de n sous la forme d'une somme de puissances de deux est unique, c'est son écriture binaire.

Le nombre d'arbres binomiaux d'un tas binomial de n étiquettes est donc égal au nombre de 1 dans l'écriture binaire de n.

La fonction demandée requiert simplement de parcourir les deux listes (supposées ordonnées) simultanément.

On commence par définir une fonction utilitaire :

```
let ordre (a:arbre) : int =
  let Noeud(ordre, _, _) = a in
  ordre
```

```
let union_tas (l1:arbre list) (l2:arbre list) : arbre list =
  let rec aux acc l1 l2 = match l1, l2 with
    | a1::q1, a2::q2 -> let Noeud(ordre1, _, _) = a1 in
      let Noeud(ordre2, _, _) = a2 in
      if ordre1 < ordre2 then aux (a1::acc) q1 l2
      else aux (a2::acc) l1 q2
    | x::q, [] | [], x::q -> aux (x::acc) q []
    | _ -> acc
  in List.rev (aux [] l1 l2)
```

Question 5

On utilise ici la fonction précédente pour obtenir une unique liste (contenant éventuellement des arbres de même ordre) et on transforme cette liste, union de deux tas en un tas (sans arbres de même ordre) :

```
let fusion_tas (t1:arbre list) (t2:arbre list) : arbre list =
  let tas = union_tas t1 t2 in

  (* on fusionne tous les arbres de même ordre *)
  let rec aux acc tas = match tas with
    | a1::a2::q -> if ordre a1 = ordre a2
      then let fus = fusion a1 a2 in
        aux acc (fus::q)
      else
        aux (a1::acc) (a2::q)
    | [x] -> x::acc
    | [] -> acc
  in List.rev (aux [] tas)
```

Pour extraire le plus petit, il suffit, vue la structure d'un tas binomial, de parcourir les racines en recherchant le minimum.

En supprimant le minimum, on transforme un tas binomial en deux tas binomiaux : l'ancien privé de l'arbre binomial de plus petite étiquette et du tas créé par la suppression de la racine de l'arbre cité que l'on ordonne suivant les ordres croissants. On reforme le tas en fusionnant ces deux tas.

La structure de tas binomial permet donc d'effectuer toutes les actions suivantes en $O(\log_2 n)$: l'insertion d'un nouvel élément, la recherche du plus petit élément, la suppression du plus petit élément, la fusion de deux tas en un seul.

Notons qu'il est également possible de décrémenter ou de supprimer un élément donné du tas en $O(\log_2 n)$ en utilisant les opérations précédentes.

⋆⋆⋆⋆⋆⋆⋆

5

Graphes

« *Des arêtes, des sommets, des poids.* »

5.1 Fête de Noël sans conflit

On considère une grande famille de n personnes avec beaucoup de gens qui ne s'entendent pas, représentée par une matrice $A = (a_{i,j}) \in \mathscr{M}_n$ telle que $a_{i,j} = a_{j,i} = 1$ si i et j ne peuvent pas se voir, 0 sinon. On a deux maisons de famille et on veut partager les n personnes entre ces deux maisons pour les fêtes de Noël, de façon que deux personnes qui ne s'entendent pas soient toujours dans des maisons différentes.

Question 1

Montrer par un exemple qu'il n'est pas toujours possible de répartir les membres de la famille entre les deux maisons pour éviter tout conflit.

Question 2

On va mettre le résultat dans un tableau `maison` de taille n tel que `maison.(i-1)=1` si la personne i est dans la première maison et `maison.(i-1)=-1` si la personne i est dans l'autre maison. Écrire une fonction `partage (a:int array array) : int array` qui teste s'il est possible de faire un partage sans conflit et propose un partage lorsque cela et possible.

Question 3

Deux personnes i et j sont dites en relation d'influence s'il existe une suite k_1, ..., k_l telle que $a_{i,k_1} = a_{k_1,k_2} = ... = a_{k_{l-1},k_l} = a_{k_l,j} = 1$. Montrer que cette relation est une relation d'équivalence. Soit N le nombre de classes d'équivalence, si l'on pose par convention $a_{i,i} = 1$ pour tout i. Montrer qu'un partage sans conflit est possible si et

seulement si il n'existe pas de suite i_1, ..., i_{2l+1} telle que $a_{i_1,i_2} = ... = a_{i_{2l+1},i_1} = 1$. Montrer que le nombre de partages sans conflits est soit 0, soit $2N$.

Question 4

On suppose maintenant qu'il y a trois maisons de famille. Proposer un algorithme qui fasse un partage sans conflits entre les trois maisons lorsque cela est possible. Qu'en pensez-vous ?

CORRIGÉ

Question 1

Il suffit d'une famille de trois personnes qui sont chacune en conflit avec les deux autres.

Question 2

On met la personne 1 dans la maison 1. On met toutes les personnes avec lesquelles elle est en conflit, s'il en existe, dans la maison -1. Puis on la marque comme « vue ». À chaque étape, on cherche une personne i qui a déjà été mise dans une maison mais n'est pas encore vue.

Premier cas de figure : il existe une telle personne i. On vérifie que i n'est pas en conflit avec des gens déjà placés dans la même maison, et on met tous les gens en conflit avec i encore non placés dans l'autre maison. Puis on marque la personne i comme « vue ».

Deuxième cas de figure : il n'existe pas de tel i. On prend alors une personne quelconque qui n'a pas encore été vue, et on la met dans la maison 1, puis on continue.

En termes plus techniques, on peut dire qu'on définit un graphe à partir de la matrice des conflits, avec un sommet pour chaque personne, une arête reliant deux sommets si les personnes correspondantes sont en conflit. L'algorithme consiste à traiter ce graphe composante connexe par composante connexe. Dans chaque composante, toutes les affectations des personnes dans les maisons sont déterminées dès qu'on a placé la première personne (qui par défaut est mise dans la maison 1).

On peut démontrer facilement l'invariant suivant :

Invariant : si i est vue, alors i est placée dans l'une des deux maisons, et toutes les personnes avec lesquelles i est en conflit se trouvent dans l'autre maison.

Ainsi, l'algorithme, lorsqu'il trouve une solution, trouve toujours une solution correcte.

Inversement, l'algorithme échoue lorsqu'il place i dans une maison où se trouve déjà une personne j en conflit avec i. Donc j est placée dans une maison mais n'a pas encore été vue, et l'algorithme n'a donc utilisé que le premier cas de figure entre le moment où j a été placée et celui où i a été placée. On a donc une chaîne de conflits telle qu'il ne peut exister de partage sans conflit (voir question 3).

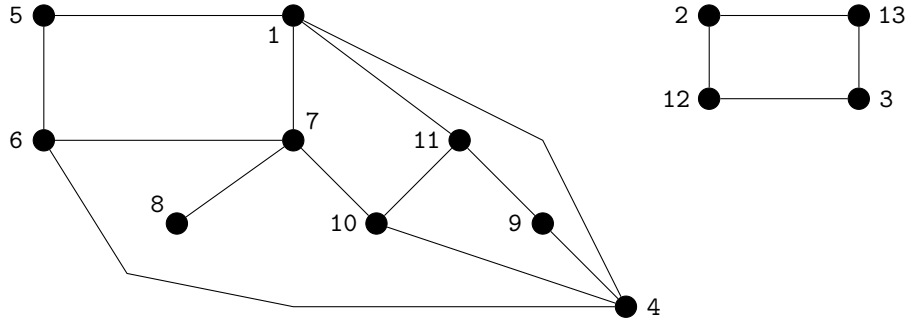
Ceci démontre que l'algorithme résout bien le problème posé.

```
let partage (a:int array array) : int array =
  let n = Array.length a in
```

```
let maison = Array.make n 0 in
let vu = Array.make n false in

for i=0 to n-1 do
  if not vu.(i) then ( (* nouvelle composante connexe *)
    maison.(i) <- 1;
    vu.(i) <- true;
    for j=i+1 to n-1 do (* ajout des personnes en conflit *)
      if a.(i).(j) = 1 then
        maison.(j) <- -1
    done;

    (* recherche d'un personne non vue dans une maison *)
    for j=i+1 to n-1 do
      if maison.(j) <> 0 && not vu.(j) then (
        (* on vérifie que j n'est pas en conflit *)
        for k=i+1 to n-1 do
          if k <> j && maison.(k) = maison.(j) && a.(k).(j) = 1 then
            failwith "partage impossible"
        done;

        (* on met les ennemis de j dans l'autre maison *)
        for k=i+1 to n-1 do
          if k <> j && a.(k).(j) = 1 then
            maison.(k) <- -maison.(j)
        done;

        vu.(j) <- true;
      )
    done
  )
done;
maison
```

Il est possible de regrouper certains tests, au détriment de la lisibilité du programme.

L'exemple suivant peut clarifier le fonctionnement de l'algorithme. Dans la figure suivante, chaque point représente une personne, et deux personnes sont reliées par un trait si elles sont en conflit.



Le tableau ci-dessous donne, pour chaque personne, l'étape à laquelle elle a été placée, ainsi que la maison dans laquelle elle a été placée.

personne	maison 1	maison -1
1	1	
2	7	
3	9	
4		4
5		2
6	3	
7		2
8	6	
9	5	
10	5	
11		2
12		8
13		8

Question 3

Réflexivité : $a_i, i = 1$ pour tout i par convention.
Symétrie : s'il existe une suite de personnes de i à j telle que chaque personne est en conflit avec la suivante et la précédente, la même suite dans l'ordre inverse prouve que $a_{i,j} = 1$.
Transitivité : s'il existe une suite de personnes en conflit de i à j et une autre suite de j à k, la concaténation des deux suites donne une suite de personnes en conflit de i à k.

On a donc une relation d'équivalence. Les classes d'équivalence correspondent à ce que nous avons appelé les « composantes connexes ».

Clairement, s'il existe une suite de i_1 à lui même telle que décrite dans l'énoncé et si i_1 est dans la maison 1, alors i_2 doit être placé dans la maison -1, i_3 dans la maison 1, ..., i_k dans la maison $(-1)^{k-1}$, donc i_{2l+1} dans la maison 1 et $i_1 = i_{2l+2}$ dans la maison -1 : contradiction.

Inversement, si la fonction de la question 2 aboutit à la réponse « partage impossible », c'est parce qu'une personne k dans la composante connexe de i se trouve contrainte d'être placée dans les deux maisons à la fois. Il existe donc deux suites de personnes en conflit allant de i à k, l'une de longueur paire et l'autre de longueur impaire. La concaténation de la première suite avec la deuxième suite en ordre inverse donne une suite de i à i de longueur impaire.

Les différentes classes d'équivalence étant indépendantes les unes des autres, le nombre total de partages sans conflits est le produit du nombre de partages sans conflit pour chaque classe. Ce nombre, pour la classe d'équivalence de i est soit 0 s'il existe une suite impaire de i à i, soit 2 sinon : en effet, le premier élément de la classe d'équivalence peut être placé au choix dans la maison 1 ou -1, et toutes les autres affectations sont ensuite déterminées par la parité de la longueur de la suite reliant une personne à i. Le produit est donc 0 ou 2^N.

Question 4

La première idée consiste à mettre la personne 1 dans la maison 1, son premier ennemi dans la maison 2, et ses ennemis successifs, soit dans la maison 2 s'ils ne sont pas en conflit avec quelqu'un se trouvant déjà dans la maison 2, soit dans la maison 3 sinon. Cela conduirait à une généralisation facile de l'algorithme de la question 2, mais malheureusement cet algorithme ne marche pas : il existe des conflits pour lesquels il ne trouve pas de solution alors qu'il en existe une. Par exemple :

$a_{1,2} = a_{1,3} = 1$;
$a_{2,1} = a_{2,3} = a_{2,4} = a_{2,5} = 1$,
$a_{3,1} = a_{3,2} = a_{3,5} = 1$,
$a_{4,2} = a_{4,5} = 1$,
$a_{5,2} = a_{5,3} = a_{5,4} = 1$.
(Les entrées non définies sont 0). Un algorithme similaire à celui de la question 2 ferait les affectations suivantes :
`maison.(0) <- 1`,
`maison.(1) <- 2`,
`maison.(2) <- 3`,
`maison.(3) <- 1`,
`maison.(4)` « impossible »,
alors que les affectations de 1, 2, 3, 4, 5 à 1, 2, 3, 2, 1 satisfont les contraintes.

De fait, il n'existe pas de modification astucieuse de l'algorithme qui le rende correct : ce problème, connu sous le nom de coloriage d'un graphe par trois couleurs, est bien connu en informatique théorique pour être difficile. Il n'existe actuellement pas d'algorithme efficace pour le résoudre, c'est à dire qui fasse beaucoup mieux que regarder les 3^n affectations possibles des personnes dans les maisons.

Plus généralement, le problème de coloriage d'un graphe avec le nombre minimum de couleurs, de sorte que deux sommets liés par une arête soient toujours de couleurs différentes, est le problème du nombre chromatique. Un problème célèbre, resté longtemps conjecture, est de démontrer que tout graphe planaire (représentable sur un plan sans que ses arêtes se coupent) a un nombre chromatique au plus égal à 4, ou encore que toute carte planaire peut être coloriée avec au plus 4 couleurs de façon que deux pays ayant une frontière commune soient toujours de couleurs différentes : ceci a finalement été démontré en réduisant l'ensemble de tous les graphes planaires possibles à un nombre fini mais élevé de cas de graphes (environ 1900), qui furent ensuite examinés un par un par ordinateur.

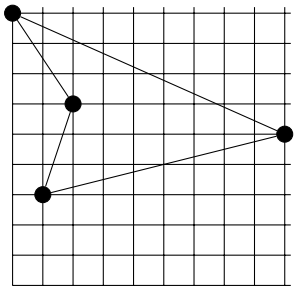
⋆⋆⋆⋆⋆⋆⋆

5.2 La tournée du facteur

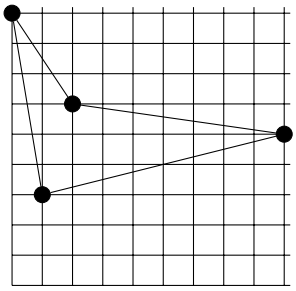
Un facteur doit distribuer du courrier dans n maisons différentes. Les coordonnées de ces maisons (qui sont supposées se trouver dans un village parfaitement plat) sont repérées par deux tableaux de flottants `x` et `y` de taille n.

La i-ième maison est de coordonnées `(x.(i), y.(i))`. On supposera que l'ordre dans lequel apparaissent les maisons est tel que `x.(0)` < `x.(1)` < ... < `x.(n)`. Le

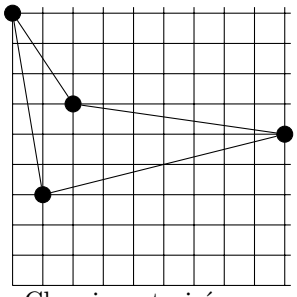
facteur cherche un chemin qui passe par toutes les maisons et qui minimise la distance totale à parcourir. Pour simplifier le problème, les seuls chemins que l'on autorisera seront des chemins qui partent du point d'abscisse minimale `(x.(0), y.(0))`, puis qui vont *toujours dans le sens des abscisses croissantes* vers le point d'abscisse maximale `x.(n-1), y.(n-1)` puis qui repartent *toujours dans le sens des abscisses décroissantes* vers le point de départ `(x.(0), x.(0))`. La figure suivante donne des exemples de chemins autorisés et de chemins interdits.



Chemin non autorisé



Chemin autorisé optimal



Chemin autorisé non optimal

Question 1

Proposer un algorithme qui donne le plus court des chemins autorisés. Justifier votre algorithme. Évaluer (en fonction de n) un ordre de grandeur du temps qu'il nécessite, en supposant qu'une addition ou une comparaison prennent une unité de temps.

Question 2

Si maintenant tous les chemins possibles sont autorisés, votre algorithme de la question précédente donne-t-il toujours un chemin de longueur minimale ?

CORRIGÉ

Question 1

Ce problème est un cas particulier du « problème du voyageur de commerce » dans lequel on recherche un circuit optimal dit « bitonique ». Un algorithme efficace utilise la programmation dynamique.

On note $(p_1, p_2, ..., p_n)$ l'ensemble des points du plan correspondant aux maisons ordonnées par abscisse croissante. On appelle « chemin bitonique » $P_{i,j}$ (où $i \leq j$), un chemin incluant les points p_1, p_2, ..., p_j qui commence à un point p_i, va jusqu'au point p_1 en allant toujours dans le sens des abscisses décroissantes puis va jusqu'au point p_j en allant toujours dans le sens des abscisses croissantes. Notons que p_j est le point le plus à droite dans $P_{i,j}$ et fait partie du sous-chemin de $P_{i,j}$ allant vers la droite. On cherche donc ici le chemin bitonique $P_{n,n}$ de taille minimale.

On note $|p_i, p_j|$ la distance euclidienne entre les points p_i et p_j et on note $b_{i,j}$ (où $i \leq j$) la longueur du chemin bitonique $P_{i,j}$ le plus court. La seule valeur $b_{i,i}$ dont nous avons besoin est $b_{n,n}$, correspondant à la longueur recherchée. On a la formule suivante pour $b_{i,j}$ avec $1 \leq i \leq j \leq n$:

$b_{1,2} = |p_1, p_2|$;
$b_{i,j} = b_{i,j-1} + |p_{j-1}, p_j|$ pour $i < j - 1$;
$b_{j-1,j} = \min_{1 \leq k < j-1} (b_{k,j-1} + |p_k, p_j|)$.

En effet, dans tout chemin bitonique se terminant par p_2, p_2 est le point d'abscisse maximale. Sa longueur est donc $|p_1, p_2|$.

On considère maintenant le chemin bitonique $P_{i,j}$ le plus court. Le point p_{j-1} se trouve quelque part sur ce chemin. S'il se trouve sur le sous-chemin allant vers la droite, alors il précède immédiatement p_j. Sinon, il se trouve sur le sous-chemin allant vers la gauche et il est le point le plus à droite donc $i = j - 1$.
Dans le premier cas, le sous-chemin de p_i à p_{j-1} est forcément le chemin bitonique $P_{i,j-1}$ le plus court, sans quoi on pourrait le remplacer par un chemin $\tilde{P}_{i,j-1}$ plus court et on obtiendrait un chemin plus court que $P_{i,j}$: absurde. La longueur de $P_{i,j}$ est donc donnée par $b_{i,j-1} + |p_{j-1}, p_j|$.
Dans le second cas, p_j a un prédécesseur immédiat p_k (où $k < j - 1$) sur le sous-chemin allant vers la droite. Par le même argument que précédemment, le sous-chemin de p_k à p_{j-1} est forcément $P_{k,j-1}$. La longueur de $P_{i,j}$ est donc donnée par $\min_{1 \leq k < j-1} (b_{k,j-1} + |p_k, p_j|)$.

Dans un chemin bitonique optimal $P_{n,n}$, l'un des points adjacents à p_n est nécessairement p_{n-1}. On a donc : $b_{n,n} = b_{n-1,n} + |p_{n-1}, p_n|$.

Pour reconstruire les points du chemin bitonique optimal $P_{n,n}$, on définit une matrice `b` de taille $(n \times n)$ pour stocker les valeurs $b_{i,i}$ et une matrice `r` telle que `r.(i-1).(j-1)` contient le prédécesseur immédiat de p_j dans le chemin bitonique $P_{i,j}$ le plus court.

L'algorithme comporte une première boucle qui itère sur tous les points puis la recherche du minimum a une complexité en $O(n)$. On a donc une complexité temporelle totale en $O(n^2)$.

Question 2

L'algorithme ne donne pas la longueur minimale dans le cas général (voyageur de commerce sans la contrainte des abscisses toujours croissantes puis toujours décroissantes). Il suffit de voir les exemples de l'énoncé, où le plus court chemin autorisé est de longueur supérieure à celle du chemin non autorisé.

⋆⋆⋆⋆⋆⋆⋆

5.3 Un mariage stable

n garçon et n filles se jugent mutuellement : chaque fille classe les garçons par ordre de préférence et chaque garçon classe les filles par ordre de préférence. Les classements sont représentés par deux matrices `f` et `g` de taille $n \times n$: `f.(i).(j)` est le classement que la fille i donne au garçon j et `g.(i).(j)` est le classement que donne le garçon i à la fille j. Par exemple, si `f.(i).(j)=1`, le garçon j est celui que préfère la fille i.

On appelle *mariage stable* une bijection M de l'ensemble des filles vers l'ensemble des garçons telle que pour tous i_1, i_2, j_1 et j_2 tels que $j_1 = M(i_1)$ et $j_2 = M(i_2)$, soit la fille i_1 préfère le garçon j_1 au garçon j_2, soit le garçon j_2 préfère la fille i_2 à la fille i_1.

Question 1

Montrer qu'un mariage stable est toujours possible et proposer un algorithme permettant de trouver un mariage stable.

CORRIGÉ

Question 1

On va construire petit à petit des couples de fiancés. On crée les deux tableaux suivants : `fiance` et `fiancee` tels que `fiance.(i)` est le numéro du fiancé de la fille i et `fiance.(j)` est le numéro de la fiancée du garçon j. Au début, tous les éléments de `fiance` et `fiancee` sont initialisés à -1. On crée également une fonction `preferee (i:int) (f:int array) : int` telle que `preferee i f` renvoie la fille préférée du garçon i parmi les choix possibles (l'algorithme supprimera des choix au fur et à mesure en plaçant la valeur $n+1$ dans le tableau f ou le tableau g). Il vient donc :

```
let preferee (i:int) (g:int array array) : int =
  let n = Array.length g in
  let ordre = ref (n+1) and p = ref (-1) in
  for j=0 to n-1 do
    if g.(i).(j) < !ordre then (
      ordre := g.(i).(j);
      p := j
    )
  done;
  !p
```

L'algorithme sera le suivant : tant qu'il existe un garçon g non fiancé [1], on cherche quelle est sa fille préférée f. Si cette fille n'est pas fiancée ou si elle ne préfère pas son actuel fiancé à g, on les fiance et on retire f des choix possible de l'ancien éventuel fiancé de f, sinon on retire f des choix de g. Donnons d'abord l'algorithme, on le discutera ensuite.

```
let mariages (f:int array array) (g:int array array) : int array =
  let n = Array.length f in
```

1. On peut bien entendu faire un algorithme équivalent en raisonnant sur les filles, pas de misogynie !

```
let fiance = Array.make n (-1) in
let fiancee = Array.make n (-1) in

let fini = ref false in
while not !fini do
  (* recherche d'un garçon non fiancé *)
  let garcon = ref 0 in
  while !garcon < n && fiancee.(!garcon) <> (-1) do
    incr garcon
  done;
  if !garcon = n then fini := true else (
    let fille = preferee !garcon g in
    let rival = fiance.(fille) in

    if rival <> (-1) then (
      if f.(fille).(!garcon) < f.(fille).(rival) then (
        (* la fille préfère le garçon à son actuel fiancé *)
        (* on rompt les fiançailles et on en célèbre de nouvelles *)
        fiancee.(!garcon) <- fille; fiancee.(fille) <- !garcon;
        fiancee.(rival) <- (-1); g.(rival).(fille) <- n+1
        (* on enlève la fille des choix possibles du rival *)
      ) else (
        (* elle préfère son actuel fiancé : on l'enlève des *)
        (* choix possibles du garçon *)
        g.(!garcon).(fille) <- n+1
      )
    ) else (    (* il n'y a pas de rival *)
      fiancee.(!garcon) <- fille; fiance.(fille) <- !garcon
    )
  )
done; (* à ce stade, tout le monde est fiancé,
         il ne reste plus qu'à célébrer les noces ! *)
fiance
```

Il nous faut maintenant prouver que ce programme termine et que le résultat est un mariage stable (ce qui prouvera l'existence d'un tel mariage).

Terminaison : Il suffit de constater qu'à chaque étape, soit le nombre de choix possibles pour un garçon (le garçon considéré ou son rival) diminue strictement, soit le nombre de garçons non fiancés décroît strictement. Le nombre $\sum$ card $\{$*choix possibles des garçons*$\}$ + card$\{$*garçons non fiancés*$\}$ est une suite d'entiers positifs strictement décroissants, elle est donc finie.

Correction : lors de l'exécution de l'algorithme, on n'enlève une fille des choix possibles d'un garçon que si cette fille préfère un autre garçon auquel elle est fiancée (ou devient fiancée). Par la suite, elle changera peut-être de fiancé, mais uniquement pour un garçon qu'elle préfère. On en déduit que si une fille est enlevée à un moment donné des choix possibles d'un garçon, elle sera finalement mariée à quelqu'un qu'elle préfère à ce garçon. Considérons deux couples (i_1, j_1) et (i_2, j_2) mariés par notre programme. Lorsque l'on a fiancé le garçon j_2 pour la dernière fois, soit la fille i_1 n'était plus dans

la liste des choix possibles de j_2 et dans ce cas, elle se retrouve mariée à un garçon qu'elle préfère à j_2, soit elle était dans la liste des choix possibles mais puisqu'alors on a fiancé j_2 à celle qu'il préférait parmi les choix restants (c'est-à-dire i_2), on en déduit que : soit la fille i_1 préfère le garçon j_1 au garçon j_2, soit le garçon j_2 préfère la fille i_2 à la fille i_1.

D'où le résultat.

⋆⋆⋆⋆⋆⋆⋆⋆

5.4 Coloriage d'un réseau de trains

On désire repeindre les gares d'un réseau de voies ferrées de sorte que deux gares reliées directement [2] ne soient jamais peintes de la même couleur. Dans un premier temps, on supposera que l'on ne dispose que de deux couleurs. Peindre des gares ainsi n'est pas toujours possible.

On suppose qu'il y a n gares. Le réseau est représenté par une matrice de booléens `r` tel que `r.(i).(j)` vaut `true` si la gare i est immédiatement reliée à la gare j et `false` sinon. On a toujours `r.(i).(j) = r.(j).(i)`.

Question 1

On dispose de seulement 2 couleurs, numérotées 1 et 2. Donner une fonction OCaml qui propose une couleur pour chaque gare de sorte que deux gares reliées directement ne soient jamais peintes de la même couleur si cela est possible, et qui indique si cela est impossible.

Question 2

Montrer que si chaque gare est reliée directement à au plus p gares, alors il existe un coloriage possible à l'aide d'au plus $p+1$ couleurs. Proposer un algorithme donnant un coloriage en $p+1$ couleurs dans un tel cas.

CORRIGÉ

Question 1

Le principe de l'algorithme est d'effectuer un parcours en profondeur du graphe des gares en faisant alterner la couleur de coloriage : on choisit une couleur arbitraire pour la première gare, puis on colore les gares voisines de l'autre couleur si elles n'ont pas encore été coloriées, ou on vérifie qu'elles sont de la bonne couleur sinon. Si pour la plupart des graphes, un parcours en largeur permettrait de trouver plus rapidement une éventuelle erreur, un parcours en profondeur est plus simple à mettre en place. On donne la fonction récursive suivante :

2. C'est à dire telles qu'on puisse aller directement en train de l'une à l'autre sans avoir à passer par une autre gare.

```
let coloriage (r:bool array array) : int array =
  let n = Array.length r in
  let couleur = Array.make n 0 in
  let vu = Array.make n false in

  let rec colorier (i:int) (col:int) : unit =
    vu.(i) <- true;
    couleur.(i) <- col;
    for j=0 to n-1 do
      if r.(i).(j) then (* i et j sont voisins *)
        if couleur.(i) = couleur.(j) then
          failwith "coloriage impossible"
        else if not vu.(j) then (
          vu.(j) <- true;
          couleur.(j) <- col;
          let new_col = if col = 1 then 2 else 1 in
          colorier j new_col
        )
    done in

  for i=0 to n-1 do
    if not vu.(i) then
      colorier i 1
  done;
  couleur
```

Question 2

La preuve se fait simplement par récurrence sur le nombre de gares. La propriété est élémentaire pour une gare. Si elle est vraie pour $n-1$ gares, alors considérons un réseau à n gares qui vérifie la propriété P : « chaque gare est connectée à au plus p gares ». Enlevons une gare à ce réseau. Le nouveau réseau vérifie trivialement la propriété P (on n'a fait qu'enlever des connexions). On peut donc le colorier à l'aide de $p+1$ couleurs. La gare qu'on avait enlevée est connectée à au plus p gares : on peut donc trouver pour cette gare au moins une couleur qui n'a pas été utilisée pour les gares adjacentes.

L'algorithme se déduit immédiatement de la récurrence.

⋆⋆⋆⋆⋆⋆⋆⋆

5.5 Graphes de dépendances

On considère l'ensemble $X = \{a, b, c, d\}$. Chaque élément de X représente une action possible d'un système et une relation réflexive et symétrique D incluse dans $X \times X$ décrit les actions ne pouvant pas être exécutées simultanément (D est dite relation de dépendance). Un graphe orienté étiqueté est un triplet $(S, A, etiq)$ où S est l'ensemble fini des sommets, A inclus dans $S \times S$ est l'ensemble fini des arêtes et $etiq$ une application

de S dans X est l'étiquetage du graphe. Un graphe fini orienté étiqueté $G = (S, A, etiq)$ est un graphe de dépendance s'il est sans cycle et si pour toute paire de sommets (s, s') :

$$(etiq(s), etiq(s')) \in D \text{ si et seulement si } [s = s' \text{ ou } (s, s') \in A \text{ ou } (s', s) \in A].$$

Ainsi„ il y a une arête entre s et s' distinct si et seulement si les actions correspondant aux étiquettes de s et s' ne peuvent être exécutées simultanément.

Question 1

Modéliser la notion de graphe orienté étiqueté décrite ci-dessus (pour un tel graphe $G = (S, A, etiq)$, on pourra supposer pour simplifier que les sommets de S sont des entiers et séparer les modélisations de (S, A) d'une part et $etiq$ d'autre part). Écrire un algorithme qui, étant donné un tel graphe, vérifie si c'est ou non un graphe de dépendance.

Question 2

Soit $G = (S, A, etiq)$ un graphe de dépendance. Une bijection C de l'ensemble $\{1, ..., |S|\}$ dans S peut être vue comme un choix successif de sommets distincts : si $C(i) = s$, on a choisi le sommet s au i-ième coup. La bijection C est fiable si, pour tous i, j : $(C(i), C(j)) \in A$ entraîne $i < j$ (i.e s'il y a une arête entre le sommet $C(i)$ et le sommet $C(j)$ alors il faut choisir $C(i)$ avant $C(j)$). Une linéarisation de $G = (S, A, etiq)$ est une suite $(u_1, u_2, ..., u_{|S|})$ d'éléments de X telle qu'il existe une bijection fiable C de $\{1, ..., |S|\}$ dans S vérifiant $u_i = etiq(C(i))$ pour tout i. Donner un algorithme qui, étant donné un graphe de dépendance G, calcule une linéarisation de G.

CORRIGÉ

Question 1

Comme le suggère l'énoncé, on va représenter l'ensemble S des sommets par l'ensemble des entiers de 1 à N pour une certaine constante N fixée. L'ensemble des arêtes sera décrit par un tableau de booléens. Enfin, la fonction d'étiquetage sera donnée par un tableau d'éléments de X.

En ce qui concerne la relation de dépendance D, il suffit de la représenter par une matrice carrée de booléens indexée par les éléments de X.

Pour vérifier qu'un graphe $(S, A, etiq)$ est bien un graphe de dépendance, la seule difficulté est de vérifier qu'il est bien acyclique. Pour cela, on effectue un parcours en profondeur du graphe en marquant en blanc les nœuds non visités, en gris les nœuds en cours de visite (i.e les nœuds se trouvant dans la pile d'appels récursifs) et en noir les nœuds visités. Si au cours du parcours on arrive sur un nœud colorié en gris alors le graphe contient un cycle, sinon il est bien acyclique.

On définit un type couleur :

```
type couleur = Blanc | Gris | Noir
```

```
let est_acyclique (g:bool array array) : bool =
  let n = Array.length g in
  let couleur = Array.make n Blanc in

  let rec parcours i =
    for j=0 to n-1 do
      if g.(i).(j) && couleur.(j) <> Noir then (
        if couleur.(j) = Gris then raise Exit;

        couleur.(j) <- Gris;
        parcours j;
        couleur.(j) <- Noir
      )
    done in

  try
    for i=0 to n-1 do
      if couleur.(i) <> Noir then
        parcours i
    done;
    true
  with Exit -> false
```

On a donc la fonction :

```
let est_graphe_dep (g:bool array array) (d:bool array array)
    (etiq:int array) : bool =
  let n = Array.length g in
  try
    for i=0 to n-1 do
      for j=0 to n-1 do
        if d.(etiq.(i)) = d.(etiq.(j)) then
          if not (i=j || g.(i).(j) || g.(j).(i)) then
            raise Exit;
        if (i=j || g.(i).(j) || g.(j).(i)) then
          if not (d.(etiq.(i)) = d.(etiq.(j))) then
            raise Exit
        done
      done;

      est_acyclique g
  with Exit -> false
```

Question 2

Par définition, une linéarisation commence par l'étiquette d'un sommet s sans prédécesseur, c'est à dire tel qu'il n'existe pas d'arête de la forme (s', s). Pour obtenir une bijection fiable, il faut donc trouver un sommet sans prédécesseur (il en existe puisque le graphe est acyclique), effacer toutes ses arêtes sortantes puis recommencer.

Notons que le graphe privé de s reste acyclique donc il existe toujours un sommet sans prédécesseur jusqu'à ce que tous les sommets aient été considérés.

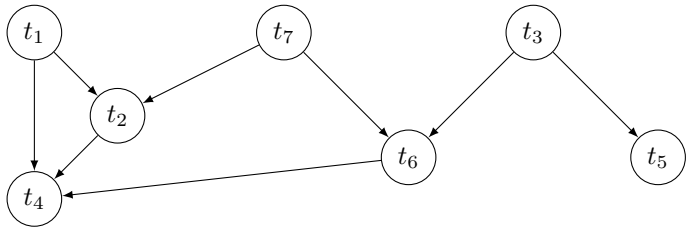
```
(* Retourne un noeud non vu et sans prédécesseur *)
let noeud_sans_pred (g:bool array array) (vus:bool array) : int =
  let n = Array.length g in
  let res = ref 0 in
  try
    for i=0 to n-1 do
      if not vus.(i) then
        let a_pred = ref false in
        for j=0 to n-1 do
          a_pred := !a_pred || g.(j).(i)
        done;
        if not !a_pred then (
          res := i;
          raise Exit
        )
    done; 0 (* impossible *)
  with Exit -> !res

let effacer_aretes_sortantes (g:bool array array)
    (x:int) : unit =
  let n = Array.length g in
  for i=0 to n-1 do
    g.(x).(i) <- false
  done

let lineariser (g:bool array array) (etiq:int array) : int array =
  let n = Array.length g in
  let suite = Array.make n 0 in
  let vus = Array.make n false in

  for i=0 to n-1 do
    let x = noeud_sans_pred g vus in
    effacer_aretes_sortantes g x;
    suite.(i) <- etiq.(x);
    vus.(x) <- true
  done;
  suite
```

⋆⋆⋆⋆⋆⋆⋆⋆

5.6 Ordonnancement

Un système de tâches $(T, <<)$ est un ensemble de n tâches $T = \{t_1, t_2, ..., t_n\}$. Ces tâches sont liées par des contraintes de précédence : on note $t_i << t_j$ pour dire que la tâche t_i doit être terminée avant que la tâche t_j ne puisse commence (on dit que t_j

est un successeur de t_i, et que t_i est un prédécesseur de t_j). Ci-dessous, on représente graphiquement un ensemble de 7 tâches, avec 8 contraintes de précédence ($t_1 << t_2$, $t_1 << t_4$, $t_2 << t_4$, $t_7 << t_2$, ...) matérialisées sous forme de flèches :



Chaque tâche a une durée entière positive $dur(t_i)$. On dispose d'une ou plusieurs machines identiques pour exécuter les tâches. Ordonnancer le système de tâches $(T, <<)$ revient à trouver un date de début d'exécution $deb(t_i)$ pour chaque tâche t_i. Le but est de minimiser le temps total d'exécution tout en respectant les contraintes de dépendance : si $t_i << t_j$, on doit avoir $deb(t_i) + dur(t_i) \leq deb(t_j)$.

Question 1

Proposer une structure de données pour un système de tâches et formaliser le problème. Écrire une fonction OCaml qui pour calcule pour chaque tâche le nombre et les numéros de ses successeurs et de ses prédécesseurs.

Question 2

Quelle condition doivent vérifier les contraintes de précédence pour qu'il soit possible d'ordonnancer le graphe ?

Question 3

On suppose la condition de la question 2 vérifiée. Pour chacun des cas suivants, proposer un algorithme pour ordonnancer le système de tâches :
(a) On dispose d'une seule machine ;
(b) On dispose de n machines ;
(c) On dispose de p machines, $1 \leq p \leq n$.

CORRIGÉ

Question 1

La structure de données la plus simple stocke les contraintes de précédence dans une matrice booléenne `c` telle que `c.(i).(j)` vaut `true` si $t_i << t_j$ et `false` sinon.

La matrice `c` donne les successeurs. Pour trouver les prédécesseurs, il suffit de parcourir la matrice `c` et stocker les résultats dans une matrice `pred` :

```
let pred_succ (c:bool array array) :
    (int array * int array * bool array array) =
  let n = Array.length c in
```

```
  let pred = Array.make_matrix n n false in
  let nb_pred = Array.make n 0 in
  let nb_succ = Array.make n 0 in

  for i=0 to n-1 do
    for j=0 to n-1 do
      if c.(i).(j) then (
        nb_succ.(i) <- nb_succ.(i) + 1;
        nb_pred.(j) <- nb_pred.(j) + 1;
        pred.(j).(i) <- true
      )
    done
  done;
  nb_pred, nb_succ, pred
```

Question 2

Il faut et il suffit que le graphe des tâches soit acyclique (sans cycle). C'est une condition nécessaire : on ne peut exécuter aucune tâche d'un cycle, et suffisante : on exécute d'abord les tâches sans prédécesseurs, puis on refait le graphe sans elles, et on recommence.

Question 3

(a) Avec une seule machine, le temps total d'exécution est la somme des n durées. On exécute successivement les tâches sans prédécesseurs parmi les tâches non traitées.

```
let une_machine (c:bool array array) (dur:int array) : int array =
  let n = Array.length c in
  let deb = Array.make n (-1) in
  let nb_pred, _, _ = pred_succ c in

  let taches_faites = ref 0 and top = ref 0 in
  while !taches_faites < n do
    for i=0 to n-1 do
      (* exécution des tâches sans prédécesseurs *)
      if nb_pred.(i) = 0 && deb.(i) = (-1) then (
        (* exécution de t_i *)
        deb.(i) <- !top;
        top := !top + dur.(i);
        incr taches_faites;

        for j=0 to n-1 do    (* mise à jour du graphe *)
          if c.(i).(j) then
            nb_pred.(j) <- nb_pred.(j) - 1
        done
      )
      done
  done; deb
```

(b) Avec n machines, il n'y a pas de problème de ressource. Par récurrence, on montre que l'optimal est d'exécuter chaque tâche au plus tôt, c'est-à-dire que la tâche i est exécutée lorsque tous ses prédécesseurs ont été exécutés, soit au temps $\max\{deb(j) + dur(j),\ j \text{ précécesseur de } i\}$ si elle a un prédécesseur j et au temps 0 sinon. La fonction ressemble un peu à la précédente : pour chaque tâche sans prédécesseur non déjà traité, on calcule le max, puis on met à jour le tableau des temps de départ.

(c) Divers algorithmes d'approximation sont possibles. Un algorithme de liste (« *list scheduling algorithm* ») est un algorithme glouton qui semble assez naturel. Quant à l'algorithme optimal, le problème problème est NP-difficile donc il n'existe pas d'algorithme polynomial (sauf si P=NP).

⋆⋆⋆⋆⋆⋆⋆⋆

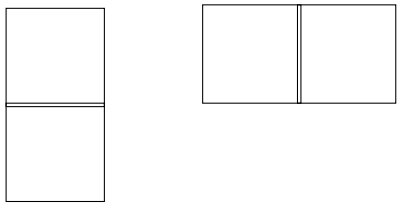
6

Géométrie et Images

« *Des représentations. Des figures. Des intersections. Des pavages.* »

6.1 Pavage d'un rectangle par des dominos

On appelle domino une pièce formée de deux carrés unité adjacents.



Un pavage d'un rectangle de largeur n et de hauteur k est un recouvrement des cases du rectangle par des dominos, chaque domino recouvrant deux cases et chaque case étant couverte par exactement un domino.

Question 1

Combien y a-t-il de pavages du rectangle $n \times 1$?

Question 2

Soit u_n le nombre de pavages du rectangle $n \times 2$. Écrire une fonction de calcul de u_n.

Question 3

Soit u_n le nombre de pavages du rectangle $n \times 3$. Proposer un algorithme de calcul de u_n. Écrire la fonction correspondante.

Question 4

Que pensez-vous du nombre de pavages du carré $n \times n$?

CORRIGÉ

Question 1

Le rectangle $n \times 1$ n'admet pas de pavage si n est impair, sa surface étant impaire. Si n est pair, il y a un pavage unique.

Question 2

Soit u_n le nombre de pavages du rectangle $n \times 2$. Considérons les deux cases les plus à droite du rectangle. Soit elles sont recouvertes par un même domino vertical, auquel cas le reste de la figure forme un rectangle $(n-1) \times 2$ qui doit également être pavé, soit elles sont recouvertes par deux dominos horizontaux l'un au dessus de l'autre, auquel cas le reste de la figure forme un rectangle $(n-1) \times 2$. On a donc la récurrence :

$$u_n = u_{n-1} + u_{n-2},$$

avec les conditions initiales $u_1 = 1$ et $u_2 = 2$. On reconnaît d'ailleurs là la suite de Fibonacci. Programmer la récurrence donne la fonction suivante :

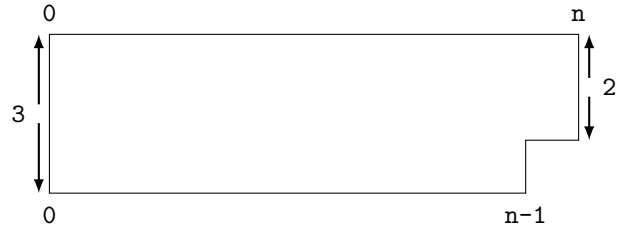
```
let largeur2 (n:int) : int =
  if n=1 then 1 else
  if n=2 then 2 else

  let a = ref 1 and b = ref 2 in
  for i=3 to n do
    (* Calcul de u_i *)
    let tmp = !a + !b in
    a := !b;
    b := tmp
  done;
  !b
```

Cette solution, très simple, permet de calculer u_n en $O(n)$ opérations. Il est cependant possible de faire mieux : en effet, il est bien connu qu'on peut calculer u_n explicitement, et on obtient $u_n = (\alpha^{n+1} - \beta^{n+1})/\sqrt{5}$, où α et β sont les deux racines de l'équation $x^2 = x + 1$. Ainsi, on a simplement besoin d'une fonction calculant la n-ième puissance d'un nombre réel, ce qui peut se faire en $O(\log_2 n)$ multiplications en utilisant la décomposition binaire de n. Voir ci-dessous une fonction dite « d'exponentiation rapide » pour calculer a^n.

```
let rec puiss (x:int) (n:int) : int = match n with
  | 0 -> 1
  | n when (n mod 2) = 0 -> puiss (x * x) (n / 2)
  | n -> (puiss (x * x) ((n-1)/2)) * x
```

Question 3

On a maintenant un rectangle de largeur 3. Soit w_n le nombre de pavages de la figure suivante, obtenue à partir du rectangle en enlevant le coin en bas à droite :



Pour calculer u_n, on regarde les dominos recouvrant les trois cases les plus à droite du rectangle. Soit ce sont trois dominos horizontaux, auquel cas il reste un pavage du rectangle de taille $n-2$, soit ce sont un domino horizontal et un domino vertical, disposés de deux façons possibles, auquel cas il reste une region du type ci-dessus, c'est à dire un rectangle de taille $n-1$ auquel il manque un coin (par symétrie, que le coin soit en haut à gauche ou en haut à droite, le nombre de pavages est le même). On a donc la récurrence :

$$u_n = 2w_{n-1} + u_{n-2},$$

De même, on observe facilement que :

$$w_n = u_{n-1} + w_{n-2}.$$

De plus, $u_n = 0$ si n est impair et $w_n = 0$ si n est pair (car la surface doit être paire). Les conditions initiales sont : $u_2 = 3$ et $w_1 = 1$. D'où la fonction suivante, qui à chaque étape calcule w_{2i-1} et u_{2i} (les autres valeurs sont 0 par parité de la surface) :

```
let largeur3 (n:int) : int =
  if n mod 2 = 1 then 0 else
  let wn = ref 1 in (* valeur de w1 *)
  let un = ref 2 in (* valeur de u2 *)
  for i=2 to (n/2) do
    (* Calcul de w2i-1 et de u2i *)
    wn := !wn + !un;
    un := 2 * !wn + !un
  done;
  !un
```

Ici encore, on peut résoudre explicitement le système et se ramener à une formule qui peut être évaluée en $O(\log_2 n)$ multiplication de réels.

Question 4

Tout d'abord, par parité, le nombre de pavages est 0 si n est impair. Si n est pair, soit u_n le nombre de pavages. On sait que chaque carré 2×2 peut être pavé de deux manières par des dominos. Donc en partitionnant un carré $n \times n$ en $n/2 \times n/2$ petits carrés 2×2, on obtient la borne inférieure :

$$u_n \geq 2^{n^2/4}.$$

Par ailleurs, tout pavage peut être décrit par un mot de $n^2/2$ bits comme suit : si le carré en haut à gauche est recouvert par un domino horizontal, le premier bit est 0, sinon il est 1. Supposons que les k premiers bits décrivent la position de k dominos. Si la case la plus en haut à gauche de la région restante est recouverte par un domino horizontal, le $(k+1)$-ième bit est 0, sinon il est 1. Ceci décrit une application injective de l'ensemble des pavages vers l'ensemble des mots de longueur $n^2/2$, et donc :

$$u_n \leq 2^{n^2/2}.$$

En fait, ce problème est bien connu en physique statistique, où le nombre de pavages correspond au nombre de configurations de molécules diatomiques sur une surface. En 1961, Kasteleyn propose une formule compliquée donnant la valeur exacte de u_n et calcule sa valeur asymptotique :

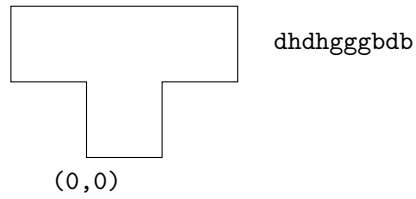
$$\frac{\log_2(u_n)}{n} \xrightarrow[n\to\infty]{} \sum_{r\geq 0} \frac{(-1)^r}{\pi(2r+1)^2} \approx 0.2916...$$

Cependant, le nombre de pavages du cube $n \times n \times n$ par des dominos est toujours un problème ouvert.

⋆⋆⋆⋆⋆⋆⋆⋆

6.2 Surface d'un polygone dessiné sur un réseau

Soit P un polygone dont toutes les arêtes sont horizontales ou verticales et tous les sommets à coordonnées entières. On le suppose donné par un mot décrivant son contour, de la façon suivante :
- on part de $(0, 0)$
- on lit les lettres du mot une à une : « d » signifie qu'on fait un pas d'une unité vers la droite, « g » vers la gauche, « h » vers le haut et « b » vers le bas. Par exemple :



Question 1

Écrire une fonction qui test si le polygone est fermé, c'est à dire si le dernier point du contour coïncide avec le premier.

Question 2

Écrire une fonction qui détermine les abscisses minimale et maximale $xmin$ et $xmax$ des points du polygone.

Question 3

Soient $ymin$ et $ymax$ définis de façon similaire. On se donne une matrice d'entiers `t` de taille $n \times m$ initialement remplie par des 0. On veut « dessiner » le polygone dans `t`, c'est à dire mettre des 0 et des 1 dans les entrées `t.(i).(j)` de façon que l'ensemble des entrées égales à 1 décrive le contour du polygone. Autrement dit, la matrice doit être comme une « fenêtre » par laquelle on voit une portion de $\mathbb{Z}^2$. Écrire une fonction à cet effet.

Question 4

Écrire une fonction pour calculer la surface du polygone.

Question 5

Proposer une solution si le polygone est dessiné sur le réseau triangulaire (pavage du plan par des triangles équilatéraux) au lieu du réseau carré.

CORRIGÉ

Question 1

Il suffit de faire le tour du polygone à partir de $(0, 0)$ en calculant à chaque pas la variation de l'abscisse et de l'ordonnée. On suppose le polygone donné par une chaîne de caractères.

```
let est_ferme (poly:string) : bool =
  let n = String.length poly in
  let x = ref 0 and y = ref 0 in
  for i=0 to n-1 do
    match poly.[i] with
      | 'h' -> incr y
      | 'b' -> decr y
      | 'd' -> incr x
      | _ -> decr x
  done;
  (!x, !y) = (0,0)
```

Question 2

Là encore, il suffit de faire un parcours du contour en regardant à chaque pas la variation de l'abscisse et en mettant à jour `xmin` et `xmax` lorsque c'est nécessaire.

```
let abscisses (poly:string) : (int * int) =
  let n = String.length poly in
  let x = ref 0  in
  let xmax = ref 0 and xmin = ref 0 in
  for i=0 to n-1 do
    match poly.[i] with
      | 'd' -> incr x; if !x > !xmax then xmax := !x
      | 'g' -> decr x; if !x < !xmin then xmin := !x
      | _ -> ()
  done;
  (!xmin, !xmax)
```

On supposera dans la suite disposer dans la suite d'une fonction `ordonnees` analogue.

Question 3

Le polygone va pouvoir se tenir dans le tableau si et seulement si $xmax - xmin + 1$ et $ymax - ymin + 1$ sont tous deux inférieurs ou égaux à n. Attention à bien dessiner le contour tel qu'il apparaît dans $\mathbb{Z}^2$, et non une image de ce contour par rotation ou symétrie. En particulier, faire varier l'indice des lignes de la matrice de 0 à $n-1$ revient à faire varier l'ordonnée du point de $\mathbb{Z}^2$ de $ymax$ à $ymax - n + 1$, et faire varier l'indice des colonnes de 0 à $n-1$ revient à faire varier l'abscisse du point de $\mathbb{Z}^2$ de $xmin$ à $xmin + n - 1$. L'application qui à un point de $\mathbb{Z}^2$ associe une entrée de la matrice doit donc être :

$$\sigma : (x, y) \mapsto \texttt{t.(ymax-y).(x-xmin)}.$$

Ainsi, $(0, 0)$ a pour image `t.(ymax).(-xmin)`, et à chaque lettre lue on a :
« d » : `j` augmente de 1
« g » : `j` diminue de 1
« h » : `i` diminue de 1
« b » : `i` augmente de 1.
D'où la fonction :

```
let dessin (poly:string) : int array array =
  let xmin, xmax = abscisses poly in
  let ymin, ymax = ordonnees poly in
  let n = ymax - ymin + 1 and m = xmax - xmin + 1 in
  let t = Array.make_matrix n m 0 in

  let l = String.length poly in
  let i = ref ymax and j = ref (-xmin) in
  t.(!i).(!j) <- 1;
  for k=0 to l-1 do
    (match poly.[k] with
      | 'd' -> incr j
      | 'g' -> decr j
      | 'h' -> decr i
      | _ -> incr i);
    t.(!i).(!j) <- 1
  done; t
```

```
let airesimple (poly:string) : int =
  let n = String.length poly in
  let aire = ref 0 in
```

Question 4

Le calcul de la surface A peut sembler difficile, mais en fait il suffit d'utiliser la formule de Green-Riemann (cas particulier de la formule de Stokes) pour que la programmation devienne très simple. On a dans notre cas :

$$A = \iint_{\text{surface}} \mathrm{d}x\mathrm{d}y = \left| \int_{\text{contour}} x\mathrm{d}y \right|$$

Le lecteur pourra vérifier sur un schéma que la justification est évidente dans le cas présent. On a donc la fonction :

```
let airesimple (poly:string) : int =
  let n = String.length poly in
  let aire = ref 0 in
  let x = ref 0 in
  for i=0 to n-1 do
    let dy = ref 0 in
    (match poly.[i] with
      | 'h' -> dy := 1
      | 'b' -> dy := -1
      | 'd' -> incr x
      | _ -> decr x);
    aire := !aire + !x * !dy
  done;
  abs !aire
```

Question 5

Si le polygone est dessiné sur le réseau triangulaire, son contour est décrit par un mot où chaque lettre décrit l'une des six directions possibles ; on peut considérer que le réseau est formé par $\mathbb{Z}^2$ plus des arêtes diagonales montantes, ce qui permet d'appliquer encore une fois Green-Riemann avec de simples calculs en entiers ; la surface obtenue par une fonction analogue à celle de la question 4 est égale à deux fois le nombre de triangles.

Si le polygone est dessiné sur le réseau régulier hexagonal, son contour est décrit par un mot où chaque lettre donne l'une des trois directions possibles, le sens étant déterminé par le sommet où on se trouve ; on peut encore une fois supposer que les sommets sont dans $\mathbb{Z}^2$, par exemple en prenant dans une direction des arêtes horizontales de longueur 1, dans la deuxième des arêtes diagonales de longueur $\sqrt{2}$, et dans la troisième, des arêtes de l'autre diagonale de longueur $\sqrt{2}$. Une fonction analogue à celle de la question 4 donne comme aire quatre fois le nombre d'hexagones dans la région considérée.

Le lecteur pourra réfléchir à un algorithme qui teste si le polygone est simple, c'est-à-dire si le contour du polygone ne s'intersecte pas lui même.

⋆⋆⋆⋆⋆⋆⋆⋆

6.3 Sommes de rectangles

On prend un ensemble de n points de $\mathbb{R}$, où chaque point a un poids élément de $\mathbb{Z}^*$. On appelle intervalle un ensemble de deux points, l'un de poids 1 et l'autre de poids -1. L'addition de deux ensembles avec poids est définie en prenant l'union des deux ensembles et en prenant comme poids la somme des deux poids (un élément qui n'apparaît pas dans un ensemble est considéré comme ayant un poids de 0 dans cet ensemble).

Question 1

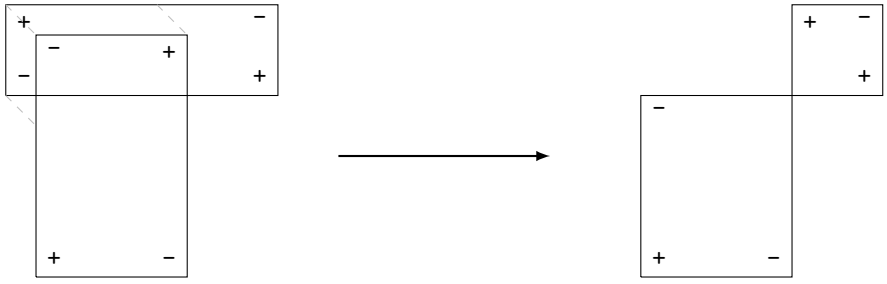
Montrer qu'un ensemble S avec poids peut être décrit comme somme d'ensembles si et seulement si la somme des poids de ses éléments est nulle.

Question 2

Écrire une fonction qui teste si un ensemble est somme d'intervalles et trouve une telle décomposition lorsque cela est possible.

On prend maintenant un ensemble de n points de $\mathbb{R}^2$, où chaque point a un poids élément de $\mathbb{Z}^*$. On appelle rectangle un ensemble de 4 points de la forme (x_1, y_1), (x_1, y_2), (x_2, y_1), (x_2, y_2), où (x_1, y_1) et (x_2, y_2) ont un poids de 1 et (x_1, y_2) et (x_2, y_1) ont un poids de -1. Ceci est représenté graphiquement de la façon suivante :

De plus, lors de l'addition de deux ensembles avec poids, les poids du même point s'additionnent. Par exemple, on voit sur la figure suivante, le résultat de la somme de deux rectangles particuliers. (On suppose ici leurs coins supérieurs gauches confondus).



On souhaite décider si un ensemble de points donné peut s'écrire comme somme de rectangles.

Question 3

Trouver une condition nécessaire et suffisante généralisant celle de la question 1.

Question 4

On suppose que les points de S sont donnés triés par ordonnée décroissante et, à ordonnée égale, par abscisse croissante. Écrire une fonction qui écrit S comme somme de rectangles lorsque cela est possible. Combien de rectangles apparaissent dans la somme dans le pire cas ?

Question 5

On s'intéresse maintenant au problème analogue en trois dimensions : on a un ensemble de points de $\mathbb{R}^3$ avec des poids entiers et on désire l'écrire comme somme de parallélépipèdes, où les poids du parallélépipède sont 1 et -1 alternativement. Proposer un algorithme.

CORRIGÉ

Question 1

Si S peut être écrit comme somme d'intervalles, chaque intervalle ayant un poids total nul (somme de ses poids), le poids total de S est nul.

Inversement, si la somme des poids des éléments de S est nulle, il existe au moins un élément x de poids strictement positif et un y de poids strictement négatif (si S est non vide). On ajoute à S l'intervalle $(x : -1, y : 1)$, et la somme des valeurs absolues des poids des éléments de S décroît strictement. Lorsque cette somme est 0, S devient vide : par récurrence, S est donc somme d'intervalles.

Question 2

On suppose S donné par un tableau de 2-tuples tel que le premier élément de chaque tuple donne l'abscisse du point et le second son poids. Il suffit de sommer les poids de tous les points pour voir si une décomposition est possible. La sortie est donnée sous forme d'une liste d'intervalles représentés par des tuples (x, y) avec x de poids 1 et y de poids -1.

```
let somme (points:(float * int) array) : (float * float) list =
  let n = Array.length points in
  let intervalles = ref [] in
  let i = ref 0 and j = ref 0 in
  let plusdepos = ref false in
  let plusdeneg = ref false in
```

```
while not (!plusdepos && !plusdeneg) do
  (* positionner i sur le premier élément de poids positif *)
  while !i < n && snd points.(!i) <= 0 do incr i done;
  if !i = n then plusdepos := true;

  (* positionner j sur le premier élément de poids négatif *)
  while !j < n && snd points.(!j) >= 0 do incr j done;
  if !j = n then plusdeneg := true;

  (* étude des cas particuliers *)
  if !plusdepos && !plusdeneg then (* fini *) () else
  if !plusdepos || !plusdeneg then
    failwith "décomposition impossible"
  else ( (* on va retrancher un intervalle à S *)
    let x, xp = points.(!i) in
    let y, yp = points.(!j) in
    points.(!i) <- (x, xp-1);
    points.(!j) <- (y, yp+1);
    intervalles := (x,y)::!intervalles
  )
done;
List.rev !intervalles
```

Question 3

La condition est : pour tout x de $\mathbb{R}$, la somme des poids des points d'abscisse x est 0, et pour tout y de $\mathbb{R}$, la somme des poids des points d'ordonnée y est 0. Cette condition est nécessaire : en effet, tout rectangle satisfait cette condition, et donc toute somme de rectangles également. Cette condition est suffisante : en effet, prenons le y maximal tel que S ait des points de poids non nul et d'ordonnée y. Comme la somme des points d'ordonnée y est nulle, il en existe au moins un, soit (x_1, y), de poids strictement positif, et au moins un, soit (x_2, y), de poids strictement négatif. Soit y_0 l'ordonnée minimale telle que S ait des points de poids non nul et d'ordonnée y_0. Si $y \neq y_0$, on retranche à S le rectangle $((x_1, y) : 1, (x_2, y) : -1, (x_1, y_0) : -1, (x_2, y_0) : 1)$, et la somme des valeurs absolues des poids des points d'ordonnée maximale a diminué. On se ramène ainsi au cas où tous les points de S ont la même ordonnée y_0. Soit (x, y_0) l'un de ces points. Comme la somme des poids des points d'abscisse x doit être nulle, (x, y_0) doit avoir un poids de 0. Par conséquent, S est réduit à l'ensemble vide.

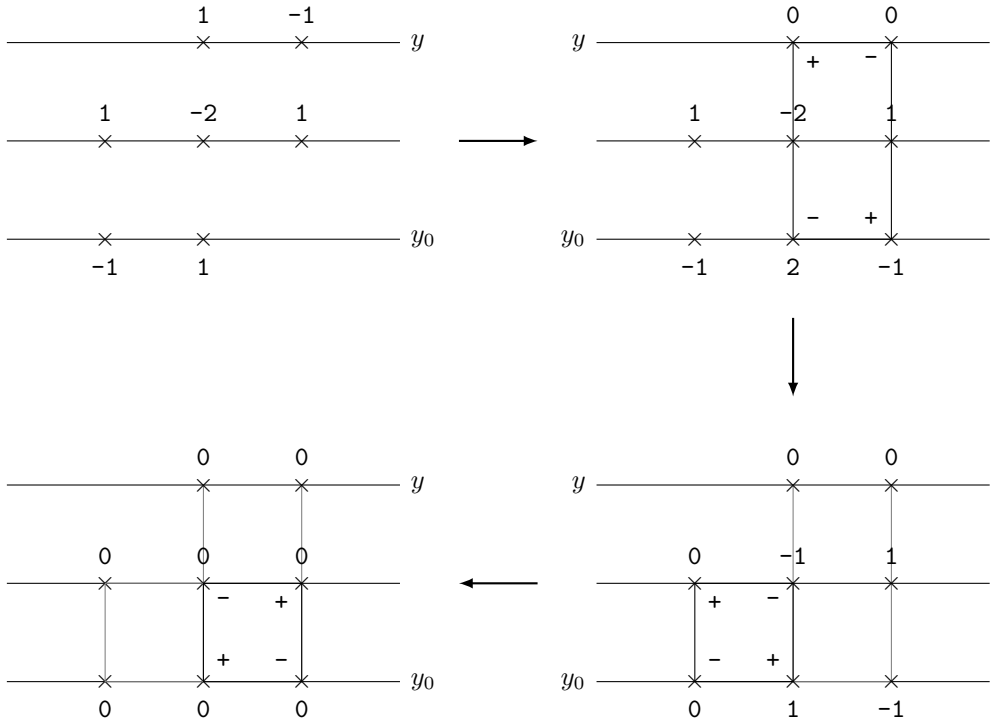
Question 4

On va utiliser l'algorithme esquissé dans la question 3. On représente les points à l'aide d'un tableau de 2-tuples tel que le premier élément de chaque tuple donne les coordonnées (x, y) du point et le second son poids. Chaque rectangle doit être donné par deux abscisses et deux ordonnées : on représentera donc un rectangle par un tuple de deux points correspondant aux points de poids 1 du rectangle. Chaque rectangle posé ajoute deux nouveaux points sur la ligne d'ordonnée y_0, et on doit vérifier à la fin que tous ces points s'annulent mutuellement : les coordonnées des points étant des

flottants, on va utiliser une table de hachage afin d'associer à chaque ordonnée x la somme des poids des points à cette ordonnée.

```
let incr_valeur_hashtable (ht:(float, int)Hashtbl.t) (k:float) (v:int)
    : unit =
  match Hashtbl.find_opt ht k with
  | Some x -> Hashtbl.replace ht k (x+v)
  | None -> Hashtbl.add ht k v

let rectangles (points:((float * float) * int) array)
    : ((float * float) * (float * float)) list =
  let n = Array.length points in
  let rectangles = ref [] in
  let poids = Hashtbl.create 8 in
  let i = ref 0 and j = ref 0 in
  let plusdepos = ref false in
  let plusdeneg = ref false in

  let y0 = snd (fst points.(n-1)) in
  while not (!plusdepos && !plusdeneg) do
    (* positionner i sur le premier élément de poids positif *)
    while !i < n && snd points.(!i) <= 0 do incr i done;
    if !i = n then plusdepos := true;

    (* positionner j sur le premier élément de poids négatif *)
    while !j < n && snd points.(!j) >= 0 do incr j done;
    if !j = n then plusdeneg := true;

    (* étude des cas particuliers *)
    if !plusdepos && !plusdeneg then (
      (* vérifier que les points de même ordonnée se compensent bien *)
      Hashtbl.iter (fun _ v ->
        if v <> 0 then failwith "décomposition impossible") poids;
      (* fini *)
    ) else if !plusdepos || !plusdeneg then
      failwith "décomposition impossible"
    else ( (* on va retrancher un rectangle à S *)
      let (x1, y1), p1 = points.(!i) in
      let (x2, y2), p2 = points.(!j) in
      if y1 <> y2 then failwith "décomposition impossible";

      points.(!i) <- (x1, y1), p1-1;
      points.(!j) <- (x2, y2), p2+1;
      incr_valeur_hashtable poids x1 1;
      incr_valeur_hashtable poids x2 (-1);
      if y1 <> y0 then rectangles := ((x1,y1), (x2,y0))::!rectangles
    ) (* aux dernières itérations, on décrémente les points sur y0 *)
  done;
  List.rev !rectangles
```

On donne ici une représentation graphique de l'algorithme :



Soient deux points (x_1, y_1) et (x_2, y_2) considérés par l'algorithme à une certaine étape. L'algorithme ajoute un rectangle à la liste si et seulement si $y_1 = y_2$ et $y_1 > y_0$ c'est-à-dire si les deux points sont sur une même ligne d'ordonnée strictement supérieure à y_0. Supposons l'ensemble de points décomposable en somme de rectangles.

Par récurrence sur le nombre de points sur une ligne $y > y_0$ donnée, montrons que l'algorithme ajoute un nombre de rectangles égal à la demi-somme des valeurs absolues des poids des points de cette ligne lorsqu'il considère tous les points de cette ligne.

Initialisation : Soit donc une ligne d'ordonnée y contenant deux uniques points (x_1, y) et (x_2, y). Tant que les deux points ont un poids non nul, l'algorithme fait strictement décroître leurs poids en valeur absolue puis ajoute un rectangle. Or, par hypothèse, on a $x_1 + x_2 = 0 \implies |x_1| = |x_2|$ donc par le variant de boucle, l'algorithme ajoute exactement $|x_1|$ rectangles soit $\frac{1}{2}(|x_1| + |x_2|)$.

Hérédité : Soit une ligne d'ordonnée y contenant $n+1$ points d'abscisses $x_1 < x_2 < ... < x_{n+1}$ et de poids respectifs $p_1, p_2, ..., p_{n+1}$. Les points de même ordonnée étant triés par abscisse croissante, l'algorithme considère les points dans l'ordre. Par hypothèse de récurrence, il ajoute $\frac{1}{2}\sum_{i=1}^{n} |p_i|$ rectangles lorsqu'il considère les points 1 à n. Par l'existence d'une décomposition en rectangles, le poids du point n après traitement des points 1 à n, noté $\tilde{p_n}$, vérifie nécessairement $|\tilde{p_n}| = |p_{n+1}|$. Par ce qui précède, l'algorithme ajoute donc $|p_{n+1}|$ rectangles à la dernière étape. D'où la propriété au rang $n + 1$.

Donc en sommant sur toutes les lignes : le nombre de rectangles renvoyés correspond donc à la demi-somme des valeurs absolues des poids de tous les points d'ordonnée strictement supérieure à y_0 :

$$\frac{1}{2}\sum_{y>y_0}\sum_{x}|\text{poids}(x,y)| = \frac{1}{2}\sum_{\substack{(x,y)\\ y>y_0}}|\text{poids}(x,y)|$$

Question 5

Les conditions des questions précédentes se généralisent naturellement : S est somme de parallélépipèdes si et seulement si pour tout choix de deux coordonnées (x et y, y et z ou z et x), la somme des poids des points qui ont ces deux coordonnées, la troisième étant libre, est égale à 0. On peut adapter l'algorithme précédent en conséquence.

⋆⋆⋆⋆⋆⋆⋆⋆

6.4 Polygones convexes

Nous définissons un point de l'écran (que l'on considère muni d'un repère orthonormé direct) comme étant un couple d'entiers (x, y), et un polygone comme étant une suite de finie de points telle que tout couple formé de deux points consécutifs ou du dernier et du premier point de cette suite représente une arête du polygone.

Nous supposons qu'il n'y a **pas de points consécutifs alignés**.

Attention : nous ne disposons pas de fonctions trigonométriques.

Question 1

Écrire la fonction OCaml `saillant` qui prend trois points A, B et C comme arguments et qui renvoie `true` si la mesure de l'angle (BA, BC), suivant le sens trigonométrique, est inférieure à 180° et `false` sinon.

Question 2

Pour tester si un polygone est convexe, on vous propose la méthode suivante : soient `p` un polygone et n le nombre de sommets `p.(i)` (pour `i` entre 0 et $n-1$) de celui-ci. Si pour tout triplet (`p.(i)`, `p.(i+1 mod n)`, `p.(i+2 mod n)`) on tourne dans un même sens (pour `i` entre 0 et $n-1$) alors le polygone est convexe, sinon il ne l'est pas. Implémenter cette méthode par une fonction OCaml `tester` qui utilise la fonction `saillant`. Évaluer le nombre d'appels à la fonction `saillant` lors de l'exécution de la fonction `tester`. La réponse est-elle toujours correcte ?

Question 3

Modifier la fonction `tester` sans en augmenter le nombre d'appels à `saillant`, de façon qu'elle renvoie toujours une réponse correcte.

Question 4

Considérons un ensemble E de points. Écrire une fonction `enveloppe` qui renvoie l'enveloppe convexe de E. Évaluer le nombre d'appels à la fonction `saillant`.

Question 5

Supposons l'ensemble E tel que son premier point est celui d'ordonnée minimale et les points suivants sont ordonnés suivant les angles polaires croissants avec `e.(0)` comme origine. Écrire une fonction de balayage telle que le nombre d'appels à la fonction `saillant` est de l'ordre de grandeur du nombre de points de l'ensemble E.

CORRIGÉ

Question 1

Nous utilisons les structures de données suivantes :

```
type point = int * int
type polygone = point array
```

La fonction `saillant` s'écrit simplement.

On utilise le fait que : $\det(u, v) = ||u|| \cdot ||v|| \cdot \sin(u, v)$.

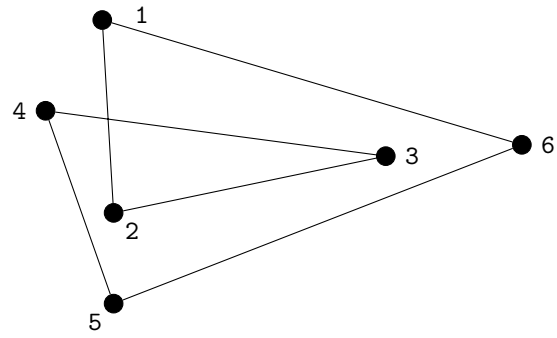
```
let saillant (a:point) (b:point) (c:point) : bool =
  let ax, ay = a and bx, by = b and cx, cy = c in
  let det = (ax - bx) * (cy - by) - (ay - by) * (cx - bx) in
  det > 0
```

Question 2

Nous appliquons l'algorithme proposé dans l'énoncé.

```
let tester (p:polygone) : bool =
  let n = Array.length p in
  let test = saillant p.(0) p.(1) p.(2) in

  try
    for i=1 to n-1 do
      let cour = saillant p.(i) p.((i+1) mod n) p.((i+2) mod n) in
      if cour <> test then
        raise Exit
    done;
    true
  with Exit -> false
```

La fonction `saillant` est appelée au plus $n + 1$ fois. Cette fonction ne tient compte que du sens de rotation lorsque l'on passe d'un sommet à son suivant, or il peut y avoir une intersection qui rompt la convexité (voir dessin).



Question 3

Afin de tester s'il y a une intersection, on compte le nombre de changements de direction suivant les abscisses lorsque l'on parcourt les sommets consécutifs du polygone considéré. Compte tenu du fait que l'on tourne toujours dans le même sens, si ce nombre est strictement inférieur à 3 alors il n'y a pas d'intersection. Dans ce cas, pour chaque arête considérée, tous les sommets sont d'un même côté, ce qui définit bien un polygone convexe.

```
let n = Array.length p in
let test = saillant p.(0) p.(1) p.(2) in
let croit = ref (fst p.(0) < fst p.(1)) in
let sens = ref 0 in

try
  for i=1 to n-1 do
    let cour = saillant p.(i) p.((i+1) mod n) p.((i+2) mod n) in
    if !croit <> (fst p.(i) <= fst p.((i+1) mod n)) then (
      incr sens;
      croit := not !croit
    );
    if !sens >= 3 || cour <> test then
      raise Exit
  done;
  true
with Exit -> false
```

Question 4

On part du point de l'ensemble E d'ordonnée la plus basse, ce point est le premier sommet de l'enveloppe convexe P. Puis on cherche le point M de l'ensemble E tel que si on passe en argument de la fonction `saillant` le segment d'extrémités le dernier sommet de P et M, et un point quelconque de E alors `saillant` renvoie `true`. On ajoute ce point à l'enveloppe convexe P puis on recommence en partant du point suivant dans E. L'algorithme s'arrête dès que l'on revient sur le premier point de P.

```
let enveloppe (e:point array) : polygone =
  let n = Array.length e in
  let env = Array.make n (0,0) in

  let min = ref 0 in
  Array.iteri (fun i _ -> if snd e.(i) < snd e.(!min) then min := i) e;

  let suiv = ref !min in
  let np = ref 0 in
  while !suiv <> !min || !np = 0 do
    let cour = !suiv in
    env.(!np) <- e.(cour);
    incr np;
    suiv := (cour + 1) mod n;

    for i=2 to n-1 do
      if not (saillant e.(!suiv) e.(cour) e.((cour + i) mod n)) then
        suiv := (cour + i) mod n
    done
  done;
  Array.sub env 0 !np
```

Question 5

Les points de E sont classés par ordre croissant d'angle polaire avec comme origine le point le plus bas de E. On place d'office les trois premiers points de E dans P. Puis, on considère le point suivant : si ce point et les deux derniers de P forment un angle saillant, on ajoute ce point à P, sinon, on retire le dernier point de P et on recommence ce test. On parcourt ainsi tous les points de E.

```
let balayage (e:point array) : polygone =
  let n = Array.length e in
  let p = Array.make n (0,0) in
  let np = ref 0 in

  for i=0 to 2 do
    p.(!np) <- e.(i);
    incr np
  done;
  for i=3 to n-1 do
    while not (saillant e.(!np-2) e.(!np-1) e.(i)) do decr np done;
    p.(!np) <- e.(i);
    incr np
  done;
  p
```

La fonction `saillant` est appelée au plus $2n - 3$ fois.

⋆⋆⋆⋆⋆⋆⋆⋆

6.5 Intersection d'un polygone et d'une fenêtre rectangulaire

Nous définissons un point de l'écran (que l'on considère muni d'un repère orthonormé direct) comme étant un couple d'entiers (x, y), et un polygone comme étant une suite de finie de points telle que tout couple formé de deux points consécutifs ou du dernier et du premier point de cette suite représente une arête du polygone.

Dans tout le problème, les polygones considérés sont par défaut convexes.

Question 1

Écrire une fonction OCaml `clip_bord` qui, étant donné un polygone et une droite parallèle à l'axe des abscisses, renvoie l'intersection du polygone avec le demi-plan supérieur à la droite donnée.

Question 2

Modifier la fonction précédente pour que l'on puisse choisir une droite qui soit parallèle à l'un ou l'autre des deux axes, et que le demi-plan déterminé par cette droite et utilisé pour l'intersection puisse lui aussi être choisi.

Question 3

Écrire une fonction OCaml `clipping` qui détermine l'intersection d'un polygone avec une fenêtre rectangulaire dont les bords sont parallèles aux deux axes.

Question 4

Que se passe-t-il si le polygone considéré n'est pas convexe ? Quelles solutions proposez-vous ?

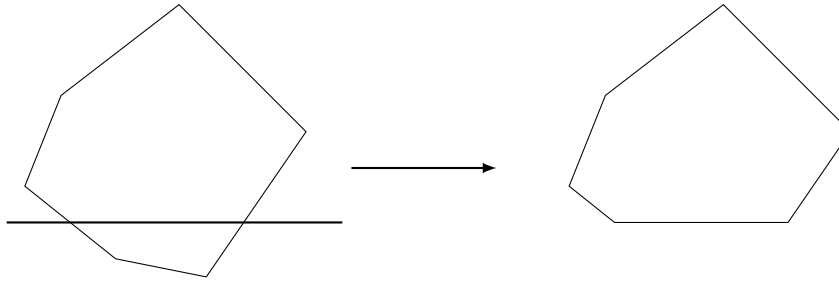
CORRIGÉ

Question 1

On parcourt le polygone et, pour chaque sommet, on regarde si le segment d'extrémités le nouveau sommet et le sommet précédent coupe cette droite ou non. Si oui, on stocke l'intersection dans le polygone intersection, puis on regarde si le sommet courant est dans ou hors de la fenêtre : s'il est dedans, on le place lui aussi dans le polygone de l'intersection.

Afin de déterminer l'abscisse x du point d'intersection entre la droite passant par (x_1, y_1) et (x_2, y_2), d'équation $ax + b$ et la droite horizontale d'ordonnée y, on résout simplement l'équation $ax + b = y$ où $a = (y_2 - y_1)/(x_2 - x_1)$. Ce qui donne la formule :

$$x = (y - y_1)\frac{(x_2 - x_1)}{(y_2 - y_1)} + x_1$$



```
type point = int * int
type polygone = point array

let clip_bord (axe:int) (p:polygone) : polygone =
  let n = Array.length p in
  (* dans le pire cas, le nouveau polygone a n+1 sommets (convexité) *)
  let clipped = Array.make (n+1) (0,0) in
  let nc = ref 0 in
  let cour = ref 0 in

  let first = ref true in
  while !first || !cour <> 0 do
    first := false;
    let pred = ref !cour in incr cour;
    if !cour = n then cour := 0;
    let x1, y1 = p.(!cour) and x2, y2 = p.(!pred) in
    if (y1 > axe) <> (y2 > axe) then (
      let x = (axe - y1) * (x2 - x1) / (y2 - y1) + x1 in
      clipped.(!nc) <- (x, axe);
      incr nc
    );
    if y1 > axe then (
      clipped.(!nc) <- (x1, y1);
      incr nc
    )
  done;
  Array.sub clipped 0 !nc
```

Question 2

La fonction est identique, on ajoute simplement un argument `t` égal à :
0 pour horizontal supérieur
1 pour horizontal-inférieur
2 pour vertical-droit
et 3 pour vertical-gauche.

De plus, on utilise une variable `sg` pour changer le sens des inégalités.

```
let clip_bord (t:int) (axe:int) (p:polygone) : polygone =
  let n = Array.length p in
  let clipped = Array.make (n+1) (0,0) in
  let nc = ref 0 in
  let cour = ref 0 in

  let sg = if t mod 2 = 0 then 1 else -1 in
  let first = ref true in
  while !first || !cour <> 0 do
    first := false;
    let pred = ref !cour in incr cour;
    if !cour = n then cour := 0;

    let (x1, y1), (x2, y2) = if t < 2 then p.(!cour), p.(!pred)
      else let (y1, x1), (y2, x2) = p.(!cour), p.(!pred)
      in (x1, y1), (x2, y2) in (* on inverse x et y *)

    if (sg * y1 > sg * axe) <> (sg * y2 > sg * axe) then (
      let x = (axe - y1) * (x2 - x1) / (y2 - y1) + x1 in
      clipped.(!nc) <- if t < 2 then (x, axe) else (axe, x);
      incr nc
    );
    if sg * y1 > sg * axe then (
      clipped.(!nc) <- if t < 2 then (x1, y1) else (y1, x1);
      incr nc
    )
  done;
  Array.sub clipped 0 !nc
```

Question 3

La fonction est très simple : on utilise la fonction `clip_bord` pour chaque axe de la fenêtre (représentée par ses coins inférieur gauche et supérieur droit).

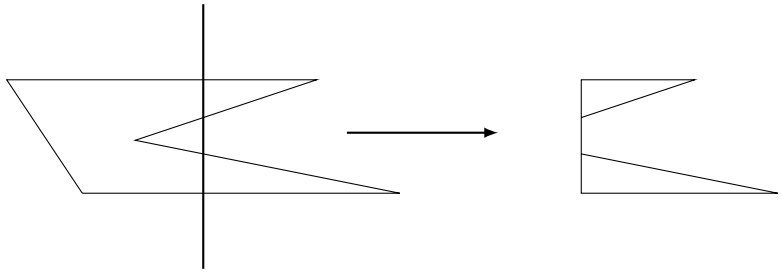
```
type fenetre = point * point
let clipping (w:fenetre) (p:polygone) =
  let (x1, y1), (x2, y2) = w in
  let p1 = clip_bord 0 y1 p in
  let p2 = clip_bord 1 y2 p1 in
  let p3 = clip_bord 2 x1 p2 in
  clip_bord 3 x2 p3
```

Nous remarquons que chaque sommet est visité un nombre fini de fois qui ne dépend pas du nombre de sommets n.

Question 4

La fonction précédente fonctionne avec des polygones non convexes à la condition que l'intersection se réduise à au plus un polygone. Sinon, les polygones résultant de

l'intersection seront réunis par des segments communs aux bords de la fenêtre.



Une solution est de considérer, lors de la construction de l'intersection avec un demi-plan, les points d'intersection avec la droite frontière et, en partant d'une extrémité, d'associer un polygone à chaque paire. Ceci nous obligerait donc à gérer plusieurs polygones.

⋆⋆⋆⋆⋆⋆⋆⋆

6.6 Intersection de segments

Nous définissons un point de l'écran (que l'on considère muni d'un repère orthonormé direct) comme étant un couple d'entiers (x, y), et un segment comme un couple de points. Dans un premier temps, nous désirons déterminer si l'intersection de deux segments donnés est vide ou non. Attention : nous ne disposons pas de fonctions trigonométriques.

Question 1

Écrire une fonction OCaml `trigo` qui, étant donné un segment et un point renvoie 1 si le point est à gauche du segment (sens trigonométrique), 0 s'il est sur la droite support du segment, et -1 sinon.

Question 2

Écrire une fonction OCaml `coupe` qui, étant donné deux segments, renvoie 1 si le deuxième segment coupe le support du premier en un point, 0 s'il se trouve sur ce support, et -1 sinon.

Question 3

Écrire une fonction OCaml `intersecte` qui, étant donné deux segments, renvoie `true` si l'intersection des deux segments passés en argument est non-vide, et `false` sinon.

Question 4

Considérons un ensemble de n segments stocké dans une variable `s`. Écrire la fonction OCaml `intersection` qui renvoie `true` si au moins deux segments de l'ensemble `s`

possèdent une intersection non-vide, et `false` sinon (notre but ici n'est pas de déterminer toutes les intersections). Évaluez le nombre d'appels à la fonction `intersecte` dans le pire des cas.

Question 5

Considérons que tout segment est donné avec des extrémités ordonnées suivant les abscisses croissantes et qu'aucun segment n'est vertical (même abscisse pour les deux extrémités).

Peut-on envisager une solution effectuant un balayage, suivant les abscisses croissantes, des sommets des segments de `s` où pour chaque sommet rencontré on utilise au plus deux fois la fonction `intersecte` ? Dans quel cas cette solution peut-elle s'avérer meilleure que la précédente ?

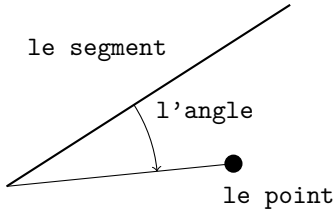
CORRIGÉ

Dans tout l'exercice, nous définissons les types suivants :

```
(* un point est un couple d'entiers représentant
  ses coordonnées à l'écran *)
type point = int * int

(* un segment est un couple de points représentant
  ses extrémités *)
type segment = point * point
```

Question 1

La connaissance du signe du sinus d'un angle défini par le segment et le point nous permet de déterminer de quel côté se trouve le point par rapport au segment.



Pour simplifier, considérons les extrémités des segments ordonnés suivant les abscisses croissantes (en cas d'égalité, suivant les ordonnées croissantes). Ainsi, la fonction `trigo` qui utilise le produit vectoriel, renvoie le signe de la mesure de l'angle défini par le segment et le point. On utilise la propriété $\det(u, v) = ||u|| \cdot ||v|| \cdot \sin(u, v)$.

```
let trigo (s:segment) (pt:point) : int =
  let (x1, y1), (x2, y2) = s and x, y = pt in
  let det = (x2 - x1) * (y - y1) - (y2 - y1) * (x - x1) in
  if det < 0 then -1
  else if det = 0 then 0
  else 1
```

Question 2

Nous vérifions si les extrémités du deuxième segment sont de part et d'autre du support du premier segment.

```
let coupe (s1:segment) (s2:segment) : int =
  let x, y = s2 in
  let trig = trigo s1 x in
  if trig = trigo s1 y then
    if trig = 0 then 0 else 1
  else 1
```

Question 3

Mis à part le cas où les deux segments ont le même support, nous vérifions si chaque segment coupe le support de l'autre.

```
let intersecte (s1:segment) (s2:segment) : bool =
  let (x1, y1), (x2, y2) = s1 in
  let (x1', y1'), (x2', y2') = s2 in
  let intsup1 = coupe s1 s2 in
  let intsup2 = coupe s2 s1 in
  if intsup1 = -1 || intsup2 = -1 then false
  else if intsup1 = 1 || intsup2 = 1 then true
  else if x1 < x1' && x1 < x2' && x2 < x1' && x2 < x2' then false
  else if y1 < y1' && y1 < y2' && y2 < y1' && y2 < y2' then false
  else true
```

Question 4

Nous effectuons un parcours exhaustif de l'ensemble des segments,

```
let intersection (s:segment array) : bool =
  let n = Array.length s in
  try
    for i=0 to n-1 do
      for j=i+1 to n-1 do
        if intersecte s.(i) s.(j) then
          raise Exit
      done;
    done;
    false
  with Exit -> true
```

Le pire cas se présente lorsqu'il n'y a pas d'intersection. Dans ce cas, pour chaque segment `e.(i)` de l'ensemble `e`, on vérifie s'il intersecte `e.(j)` pour tout $j > i$. Donc le nombre d'appels à `intersecte` vaut :

$$\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2}$$

Question 5

Considérons la droite verticale Δ_α correspondant à l'abscisse α. Cette droite va balayer l'écran en partant de $\alpha = 0$. Supposons que l'ont ait une une structure de données `t` permettant de stocker la liste des segments interceptés par la droite de balayage Δ_α (les segments interceptés sont ainsi ordonnés suivant un ordre total $\leq_\alpha$).

On se propose ici d'implémenter une version simplifiée l'algorithme de Bentley-Ottmann (permettant de trouver toutes les intersections d'un ensemble de segments), utilisant un arbre binaire de recherche équilibré et une file de priorité.

On utilise ici un arbre bicolore pour stocker les segments rencontrés en les ordonnant selon l'ordonnée de leur point d'abscisse minimale. On suppose disposer des fonctions suivantes :

- `insere_abr (t:arbre) (s:segment) : arbre` qui permet d'insérer le segment `s` dans l'arbre bicolore `t` ;
- `supprime_abr (t:arbre) (s:segment) : arbre` qui permet de supprimer le segment `s` de l'arbre bicolore `t` ;
- `en_dessous (t:arbre) (s:segment) : segment option` qui renvoie `Some(s')` s'il existe un segment `s'` immédiatement en dessous de `s` et `None` sinon ;
- `au_dessus (t:arbre) (s:segment) : segment option` qui renvoie `Some(s')` s'il existe un segment `s'` immédiatement au dessus de `s` et `None` sinon.

Certains détails d'implémentation sont donnés dans l'exercice 2 du chapitre 4.

On se munit également d'un tas-min permettant de stocker les points du plan et le segment auquel ils sont associés, ordonnés par abscisse croissante. On suppose disposer des fonctions suivantes :

- `init_tas (n:int) : tas` qui permet de créer un tas-min de taille n ;
- `insere_tas (t:tas) (e:point * segment) : unit` qui permet d'insérer l'élément `e` dans le tas `t` ;
- `retire_tas (t:tas) : (point * segment)` qui permet de retirer l'élément à la racine (donc d'abscisse minimale) du tas `t` et de le renvoyer.

Certains détails d'implémentation sont donnés dans l'exercice 1 du chapitre 4.

L'algorithme parcourt tous les segments et ajoute leurs extrémités au tas-min qui permet de les ordonner selon leur abscisse. Il balaye ensuite les points (c'est-à-dire les retire du tas) tant qu'aucune intersection n'a été trouvée et qu'il reste des points dans le tas. Lorsqu'un nouveau point est rencontré, deux cas de figure sont possibles :

- Si le point correspond à l'extrémité gauche d'un segment `s` alors ce segment n'a jamais été vu et il est alors inséré dans l'arbre `t`. Puis, on vérifie si `s` intersecte le segment immédiatement au dessus de lui ou le segment immédiatement au dessus de lui (en parcourant l'arbre).
- Si le point correspond à l'extrémité droite d'un segment `s` alors ce segment se trouve déjà dans `t`. On vérifie si `s` intersecte le segment immédiatement au dessus de lui ou le segment immédiatement au dessus de lui, puis, on retire `s` de `t`.

On donne la fonction suivante :

```
let test_intersection (t:arbre) (s:segment) : unit =
  match au_dessus t s with
  | Some x when intersecte x s -> raise Exit
  | _ -> ();
match en_dessous t s with
  | Some x when intersecte x s -> raise Exit
  | _ -> ()

let intersection (s:segment array) : bool =
  let n = Array.length s in
  let tas = init_tas (2*n) in
  let arbre = ref Nil in

  for i=0 to n-1 do
    let p1, p2 = s.(i) in
    insere_tas tas (p1, s.(i));
    insere_tas tas (p2, s.(i))
  done;

  try
    for i=0 to 2*n-1 do
      let p, s = retire_tas tas in
      let p1, p2 = s in
      if p = p1 then (    (* Cas 1 *)
        arbre := insere_abr !arbre s;
        test_intersection !arbre s
      ) else (    (* Cas 2 *)
        test_intersection !arbre s;
        arbre := supprime_abr !arbre s
      )
    done;
    false
  with Exit -> true
```

L'arbre `t` étant équilibré, toutes les opérations ont une complexité en $O(\log_2 n)$ avec n le nombre de segments. De même, les opérations effectuées dans le tas-min sont en $O(\log_2 n)$.

Dans le pire cas (pas d'intersection), chaque segment est ajouté puis retiré de l'arbre. On a donc une complexité en $O(n \log_2 n)$ et le nombre d'appels à `intersecte` est de $2n$. Notons que la complexité dépend de la structure de données choisie et qu'un choix moins judicieux pourrait conduire à une solution moins efficace.

⋆⋆⋆⋆⋆⋆⋆⋆

7

Arithmétique et calculs numériques

« *Écrire. Calculer. Résoudre.* »

7.1 Parties d'un ensemble

On considère un ensemble $E = \{0, ..., N-1\}$. On représente une partie de E par un tableau `p` d'entiers de $\{0, 1\}$ de taille N tel que `p.(i)=1` si et seulement si l'élément i est dans la partie représentée par `p`. On associe à chaque partie `p` son numéro défini par numéro(`p`) $= \sum_{i=0}^{N-1}$ `p.(i)`2^i.

Question 1

Écrire un algorithme qui, étant donnée une variable `p` représentant une partie, calcule numero(`p`). Quel est le nombre de multiplications par 2 effectuées par votre algorithme ? Pouvez-vous diminuer ce nombre ?

Question 2

Étant donné un entier positif ou nul k, montrer qu'il existe au plus une partie de `p` telle que numero(`p`) $= k$. Écrire un algorithme qui donne cette partie `p` lorsqu'elle existe. On pourra utiliser la fonction OCaml `mod` (si `a` et `b` sont deux entiers positifs, `a mod b` est le reste de la division euclidienne de `a` par `b`).

Question 3

Écrire un algorithme qui, étant donnée une partie `p`, calcule la partie `q` (lorsqu'elle existe) telle que numero(`q`) = numero(`p`) + 1.

Question 4

Écrire un algorithme qui énumère toutes les parties de E.

CORRIGÉ

Question 1

On donne la fonction :

```
let numero (p:int array) : int =
  let n = Array.length p in
  let s = ref 0 in
  let pow = ref 1 in
  for i=0 to n-1 do
    s := !s + p.(i) * !pow;
    pow := 2 * !pow
  done;
  !s
```

Le nombre de multiplications effectuées par cet algorithme est de N. La puissance maximale de 2 à calculer étant 2^{N-1}, $N-1$ multiplications sont nécessaires. Il est facile d'écrire un algorithme, peut-être un tout petit peu moins naturel que celui présenté, effectuant $N-1$ multiplications.

Question 2

Soit $k \in \mathbb{N}$. S'il existe une partie `p` de E telle que numero(`q`) $= k$, la suite `p.(n-1)`, `p.(n-2)`, ..., `p.(0)` est exactement l'écriture en binaire du nombre k. Ainsi, si la partie `p` existe, elle est unique. De plus, `p` existe si et seulement si $0 \leq k \leq \sum_{i=0}^{N-1} 2^i = 2^N - 1$.

Dans ce cas, les éléments `p.(i)` sont obtenus en calculant successivement `k mod 2`, `(k/2) mod 2`, ...

```
let partie (k:int) (n:int) : int array =
  let p = Array.make n 0 in

  let k = ref k in
  for i = 0 to n-1 do
    p.(i) <- !k mod 2;
    k := !k/2;
  done;
  if !k <> 0 then failwith "k est trop grand";
  p
```

Cette fonction a l'inconvénient de faire des calculs inutiles si k est petit par rapport à 2^N. La fonction suivante est, de ce point de vue, plus efficace.

```
let partie2 (k:int) (n:int) : int array =
  let p = Array.make n 0 in
```

```
  let rec aux k i =
    if i > n then failwith "k est trop grand";
    if k > 0 then (
      p.(i) <- k mod 2;
      aux (k/2) (i+1)
    )
  in aux k 0;
  p
```

Question 3

Cette question revient à déterminer le successeur d'un entier écrit en binaire. Il suffit de parcourir les chiffres de cet entier de droite à gauche et de transformer les 1 en 0 jusqu'au moment où on trouve un 0 que l'on transforme en 1.

Exemple : $\begin{array}{r} 101011 \\ +1 \\ \hline 101100 \end{array}$

```
  let n = Array.length p in
  let q = Array.copy p in

  let rec aux i =
    if i > 0 then (
      if p.(i) = 1 then (
        q.(i) <- 0;
        aux (i-1)
      ) else q.(i) <- 1
    ) else if p.(i) = 1 then
      failwith "pas de successeur"
    else q.(i) <- 1
  in aux (n-1);
  q
```

Question 4

Il suffit d'appliquer l'algorithme de la question précédente de manière itérative à partir de la partie vide.

```
let affiche_partie (p:int array) =
  let n = Array.length p in
  Printf.printf "{";
  let first = ref true in
  for i=0 to n-1 do
    if p.(i) = 1 && !first then
      (Printf.printf "%d" i; first := false)
    else if p.(i) = 1 then
      Printf.printf ", %d" i
  done; Printf.printf "}\n"
```

```
let enumere (n:int) : unit =
  (* initialisation de la partie vide *)
  let p = ref (Array.make n 0) in
  try
    while true do
      affiche_partie !p;
      p := plus_un !p
    done;
  with _ -> ()
```

Notons qu'il serait plus efficace de définir une fonction `plus_un_en_place (p:int array) : int` qui modifie `p` en place plutôt que créer un nouveau tableau à chaque itération.

⋆⋆⋆⋆⋆⋆⋆⋆

7.2 Conversion d'écriture romaine-décimale

On rappelle le système d'écriture des nombres utilisés par les romains. Il utilise les symboles I=1, V=5, X=10, L=50, C=100, D=500, M=1000. On rappelle quelques exemples dont le candidat pourra s'inspirer pour déduire les règles d'écriture des nombres :

(I, II, III, IV, V, VI, VII, VIII, IX, X) = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10).

XXXIX = 39, XL = 40.

(XLI, XLII, XLIII, XLIV, XLV, XLVI, XLVII, XLVIII, XLIX, L) = (41, 42, 43, 44, 45, 46, 47, 48 , 49, 50).

En particulier, on ne peut jamais avoir quatre symboles identiques consécutifs. Un nombre n est supposé codé dans un tableau d'entiers s'il est écrit dans le système décimal, le chiffre le plus significatif étant en tête du tableau, et dans un tableau de caractères s'il est écrit dans le système romain.

Question 1

Quel est le plus grand nombre qui puisse être écrit ? Quel est le nombre dont l'écriture est la plus longue ?

Question 2

Donner un algorithme qui prend en entrée un nombre écrit dans le système romain et donne en sortie son écriture dans le système décimal usuel. Écrire la fonction correspondante.

Question 3

Donner un algorithme de conversion de décimal en romain. Écrire la fonction correspondante.

Question 4

Proposer un algorithme d'addition de deux nombres dans le système romain.

Question 5

Que pensez-vous du nombre moyen de caractères nécessaires pour écrire un nombre en romain ?

CORRIGÉ

Question 1

Le plus grand nombre ne peut avoir plus de trois M pour les milliers. Les centaines, dizaines et unités peuvent toutes valoir 9. La réponse est donc 3999, c'est-à-dire MMMCMXXCIX.

Le nombre le plus long doit aussi avoir trois M correspondant au chiffre des milliers, mais ensuite chaque chiffre 8 correspond à quatre caractères, le maximum possible. La réponse est donc 3888, c'est-à-dire MMMDCCCLXXXVIII qui comporte 15 caractères.

Question 2

Notons que chaque symbole correspond à une quantité précise qui doit être soit ajoutée, soit retranchée du total, suivant le symbole qui suit. On peut par exemple commencer par convertir chaque symbole en la quantité correspondante, puis calculer l'entier représenté en faisant une suite d'additions et de soustractions, puis convertir cet entier en tableau de chiffres.

```
let rom_to_dec (r:char array) : int array =
  let n = Array.length r in

  (* Création du tableau des valeurs *)
  let valeurs = Array.make n 0 in
  for i=0 to n-1 do
    valeurs.(i) <- match r.(i) with
      | 'I' -> 1
      | 'V' -> 5
      | 'X' -> 10
      | 'L' -> 50
      | 'C' -> 100
      | 'D' -> 500
      | _   -> 1000
  done;

  (* Conversion en entier *)
  let entier = ref valeurs.(n-1) in
  for i=0 to n-2 do
    if valeurs.(i) >= valeurs.(i+1)
```

```
    then entier := !entier + valeurs.(i)
  else entier := !entier - valeurs.(i)
done;

(* Conversion de l'entier en tableau de chiffres,
    poids faible en tête *)
let d = Array.make 4 0 in
let rec aux entier i =
  if entier <> 0 then (
    d.(i) <- entier mod 10;
    aux (entier/10) (i+1)
  )
in aux !entier 0;

(* Inversion du tableau pour mettre les poids forts en tête *)
let tmp = d.(0) in
d.(0) <- d.(3); d.(3) <- tmp;
let tmp = d.(1) in
d.(1) <- d.(2); d.(2) <- tmp;
d
```

Notons que les instructions de conversion d'entier en tableau de chiffres et d'inversion de l'ordre d'un tableau sont des exercices standards qui doivent être maîtrisés par les candidats. Une autre solution aurait consisté à lire le tableau de caractères romains, en séparant les parties utilisées pour le chiffre des milliers, celui des centaines, celui des dizaines et celui des unités. Les milliers sont faciles à convertir (il suffit de compter le nombre de M). Pour les autres chiffres, une même fonction peut être appliquée pour convertir les centaines, les dizaines ou les unités, en remplaçant M, D et C respectivement par C, L et X pour les dizaines et par X, V et I pour les unités. On obtiendrait ainsi une fonction de traduction directe sans passer par l'évaluation du nombre.

Question 3

L'algorithme naturel consiste à lire chaque décimale et à écrire la suite de caractères correspondante. On peut appeler une fonction auxiliaire `chiffre` qui fera le même travail pour les centaines, les dizaines et les unités, en remplaçant les symboles adéquatement. Soit k le chiffre à convertir, et j la première case libre du tableau.

On suppose que le nombre ici que le nombre à coder est inférieur à 3999 et donc qu'on peut prendre $|\mathtt{d}| = 4$.

```
let dec_to_rom (d:int array) : char array =
  let r = Array.make 15 '0' in
  let j = ref 0 in

  let chiffre (x:char) (v:char) (i:char) (k:int) : unit =
    match k with
      | 1 -> r.(!j) <- i; j := !j+1
      | 2 -> r.(!j) <- i; r.(!j+1) <- i; j := !j+2
```

```
      | 3 -> r.(!j) <- i; r.(!j+1) <- i; r.(!j+2) <- i; j := !j+3
      | 4 -> r.(!j) <- i; r.(!j+1) <- v; j := !j+2
      | 5 -> r.(!j) <- v; j := !j+1
      | 6 -> r.(!j) <- v; r.(!j+1) <- i; j := !j+2
      | 7 -> r.(!j) <- v; r.(!j+1) <- i; r.(!j+2) <- i; j := !j+3
      | 8 -> r.(!j) <- v; r.(!j+1) <- i; r.(!j+2) <- i; r.(!j+3) <- i;
             j := !j+4
      | 9 -> r.(!j) <- i; r.(!j+1) <- x; j := !j+2
      | _ -> ()
  in
  (* On traduit les chiffres un à un *)
  chiffre 'M' 'M' 'M' d.(0);
  chiffre 'M' 'D' 'C' d.(1);
  chiffre 'C' 'L' 'X' d.(2);
  chiffre 'X' 'V' 'I' d.(3);
  Array.sub r 0 !j
```

Question 4

Il n'existe en fait pas de méthode plus facile que de d'abord faire la conversion en entier. On va donc simplement utiliser les fonctions précédentes. On commence par définir une fonction `tab_to_int` qui permet de calculer la valeur d'un entier écrit en décimal :

```
let tab_to_int (d:int array) : int =
  let n = Array.length d in
  let entier = ref d.(0) in
  for i=1 to n-1 do
    entier := 10 * !entier + d.(i)
  done;
  !entier
```

Puis on obtient :

```
let add (r1:char array) (r2:char array) =
  (* Conversion en tableaux de chiffres *)
  let d1 = rom_to_dec r1 in
  let d2 = rom_to_dec r2 in
  (* Conversion en entier *)
  let n1 = tab_to_int d1 in
  let n2 = tab_to_int d2 in
  let somme = n1 + n2 in
  (* Conversion de l'entier somme en tableau de chiffres,
    poids faible en tête *)
  let d = Array.make 4 0 in
  let rec aux entier i =
    if entier <> 0 then (
      d.(i) <- entier mod 10;
      aux (entier/10) (i+1)
    )
```

```
in aux somme 0;

(* Inversion du tableau pour mettre les poids forts en tête *)
let tmp = d.(0) in
d.(0) <- d.(3); d.(3) <- tmp;
let tmp = d.(1) in
d.(1) <- d.(2); d.(2) <- tmp;

(* Conversion en caractères romains *)
dec_to_rom d
```

Question 5

Calculons la longueur moyenne d'un nombre compris entre 0 et 3999 quand il est écrit en romain.

Le chiffre des milliers prend 0, 1, 2 ou 3 caractères donc la longueur de sa traduction est : 0 pour les 1000 premiers nombres, 1 pour les 1000 suivants, 2 pour les 1000 suivants et 3 pour les 1000 suivants, soit en moyenne 2,5.

Le chiffre des centaines a une longueur moyenne de $(0+1+2+3+2+1+2+3+4+2)/10$, soit 2 caractères.

Le calcul est le même pour les dizaines et pour les unités. Un nombre s'écrit donc en moyenne avec $1,5 + 3 \times 2 = 7,5$ caractères. En revanche, un nombre écrit en notation décimale et compris entre 0 et 3999 utilise utilise 1 chiffre pour les 10 premiers nombres, 2 pour les 90 suivants, 3 pour les 900 suivants et 4 pour les 3000 suivants, soit en moyenne : $(10 \times 1 + 90 \times 2 + 900 \times 3 + 3000 \times 4)/4000 = 3,7225$.

Notons qu'on aurait pu ajouter une question subsidiaire plus difficile : proposer un algorithme qui teste si un tableau de caractères est bien une représentation valide d'un nombre en écriture romaine.

⋆⋆⋆⋆⋆⋆⋆⋆

7.3 Polynômes à trois variables

On veut représenter des polynômes à coefficients entiers sur trois variables X, Y et Z. Un monôme est un élément de la forme $aX^bY^cZ^d$ où a est un entier et b, c, d des entiers positifs ou nuls. L'entier a est le coefficient du monôme et le triplet (b, c, d) le degré du monôme. Un polynôme P est un ensemble non vide de monômes. Dans un polynôme, on peut regrouper les monômes de même degré en faisant la somme de leurs coefficients (bien sûr, si cette somme est nulle, il est sans intérêt de faire apparaître le monôme correspondant dans l'écriture du polynôme). Un polynôme est dit réduit s'il contient au plus un monôme de degré fixé.

Question 1

Modéliser la situation décrite et écrire un algorithme qui, étant donné un polynôme, calcule une représentation réduite de ce polynôme. Quel est le nombre de comparaisons entre monômes effectuées par votre algorithme ?

Question 2

Donner une manière de ranger les polynômes dans la modélisation d'un polynôme afin qu'il existe un algorithme résolvant la question 1 en effectuant au plus N comparaisons entre monômes, où N est le nombre de monômes. Donner cet algorithme.

Question 3

Un polynôme est dit symétrique si le monôme $aX^bY^cZ^d$ apparaît dans P si et seulement si les monômes $aX^bY^dZ^c$, $aX^cY^bZ^d$, $aX^cY^dZ^b$, $aX^dY^bZ^c$ et $aX^dY^cZ^b$ apparaissent. Écrire un algorithme qui, étant donné un polynôme P, indique si ce polynôme est symétrique ou non.

Question 4

Un monôme est dit unitaire si son coefficient est égal à 1. Donner un algorithme qui énumère tous les monômes unitaires de degré (b, c, d) tels que $b + c + d = k$ fixé.

CORRIGÉ

Question 1

Une modélisation simple consiste à représenter un monôme par un 4-tuple d'entiers et un polynôme par un tableau de monômes (l'utilisation de types structurés permettrait une modélisation plus élégante mais ces types ne sont pas au programme). Nous sommes ainsi amenés à faire les déclarations suivantes :

```
type monome = int * int * int * int
type polynome = monome array
```

Le principe de l'algorithme consiste à prendre un monôme dans le polynôme et à chercher tous les monômes de même degré. Il ne faut pas oublier « d'effacer » les monômes traités en mettant leurs coefficients à 0.

```
let reduit (p:polynome) : polynome =
  let n = Array.length p in
  let q = Array.copy p in

  let k = ref 0 in (* prochaine case de q à remplir *)
  for i=0 to n-1 do
    let a, b, c, d = q.(i) in
    if a <> 0 then
      let coef = ref a in
      (* Parcours des monômes p.(i+1)... *)
```

```
      (* pour chercher ceux de même degré que p.(i) *)
      for j=i+1 to n-1 do
        let a', b', c', d' = q.(j) in
        if (b, c, d) = (b', c', d') then (
          coef := !coef + a';
          q.(j) <- (0,b,c,d); (* on efface le monôme *)
        )
      done;
      if !coef <> 0 then (
        q.(!k) <- (!coef, b, c, d);
        incr k
      )
    done;
```

Dans le pire cas, tous les monômes du polynôme sont de degrés distincts et le nombre de comparaisons entre monômes est $\frac{N(N-1)}{2}$.

Question 2

Pour n'avoir qu'au plus N comparaisons à effectuer, il suffit que les monômes de même degré soient rangés consécutivement dans le polynôme. C'est le cas par exemple dès qu'on définit un ordre sur le degré des monômes et que l'on range les monômes selon cet ordre. Sous cette hypothèse, l'algorithme devient alors :

```
let reduit_quand_trie (p:polynome) : polynome =
  let n = Array.length p in
  let q = Array.copy p in

  let k = ref 0 in (* prochaine case de q à remplir *)
  let coef = ref 0 in (* coefficient du monôme en cours de calcul *)
  let idx = ref 0 in (* p.(!idx) est le monôme en cours de calcul *)
  for i=0 to n-1 do
    let a, b, c, d = q.(!idx) in (* monôme courant *)
    let a', b', c', d' = q.(i) in
    if a <> 0 then
      let meme_deg = (b, c, d) = (b', c', d') in
      if meme_deg then
        coef := !coef + a'
      else (    (* On a trouvé un monôme de degré différent *)
        if !coef <> 0 then (
          q.(!k) <- (!coef, b, c, d);
          incr k
        );
        (* On met à jour les variables relatives au monôme *)
        idx := i;
        let c, _, _, _ = q.(i) in
        coef := c
      );
      (* On ajoute les monômes du degré restant *)
```

```
      if i=n-1 && !coef <> 0 then (
        q.(!k) <- (!coef, b', c', d');
        incr k
      )
  done;
  Array.sub q 0 !k
```

Question 3

On va supposer que le polynôme considéré est réduit. On va prendre successivement les monômes et aller vérifier que tous les monômes obtenus par permutation du degré sont présents. On mettra à 0 les coefficients de tous les monômes traités.

```
let meme_degree_permutation (p:monome) (q:monome) : bool =
  let _, b, c, d = p in
  let _, b', c', d' = q in
  (b=b' && c=c' && d=d') || (b=b' && c=d' && d=c')
  || (b=c' && c=b' && d=d') || (b=c' && c=d' && d=b')
  || (b=d' && c=b' && d=c') || (b=d' && c=c' && d=b')

let est_symetrique (p:polynome) : bool =
  let n = Array.length p in
  let p = Array.copy p in
  try
    for i=0 to n-1 do
      let a, b, c, d = p.(i) in
      if a <> 0 then
        let nb_sym  = ref 0 in
        (* Parcours des monômes p.(i+1)... pour chercher ceux *)
        (* de degré égal à p.(i) à une permutation près *)
        for j=i+1 to n-1 do
          let a', b', c', d' = p.(j) in
          if a' <> 0 && meme_degree_permutation p.(i) p.(j) then (
            if a' = a then (
              incr nb_sym;
              p.(j) <- (0, b', c', d')
            ) else raise Exit
          )
        done;
        if !nb_sym <> 5 then raise Exit
    done;
    true
  with Exit -> false
```

Question 4

Cette question ne présente aucune difficulté particulière. Il suffit d'énumérer tous les monômes de coefficient 1 à l'aide de deux boucles « for » convenablement imbriquées.

```
let enumere (k:int) : unit =
  for i=0 to k do
    for j=0 to k-i do
      Printf.printf "X^%d Y^%d Z^%d\n" i j (k-i-j)
    done
  done
```

⋆⋆⋆⋆⋆⋆⋆⋆

7.4 JPEG : Transformée en cosinus discrète

Nous considérons des images en niveaux de gris données sous la forme de matrices d'entiers compris entre 0 et 255. Le format JPEG permet de stocker ces images en occupant moins de place. Ce format est basé sur la transformée en cosinus discrète qui transforme un bloc image en un bloc de fréquences que l'on filtre afin d'obtenir une matrice creuse (beaucoup de coefficients nuls).

Chaque image est découpée en blocs 8×8 (ce qui sous-entend que le nombre de colonnes et que le nombres de lignes sont des multiples de 8) et on applique une transformée en cosinus discrète sur chacun de ces blocs 8×8. Un bloc 8×8 dont les pixels sont repérés par $\text{pix}(x, y)$ avec x et y deux entiers compris entre 0 et 7 (inclus) est transformé en un bloc de fréquences $\text{dct}(i, j)$ avec i et j deux entiers compris entre 0 et 7 (inclus). Nous avons la relation suivante :

$$\text{dct}(i,j) = \frac{1}{4} C(i) C(j) \sum_{x=0}^{7} \sum_{y=0}^{7} \text{pix}(x,y) \cos \frac{(2x+1)i\pi}{16} \cos \frac{(2y+1)j\pi}{16}$$

avec $C(k) = \begin{cases} \frac{1}{\sqrt{2}} & \text{si } k = 0 \\ 1 & \text{sinon} \end{cases}$

Question 1

1. Écrire une fonction OCaml qui prend en entrée la matrice 8×8 d'un bloc image et qui renvoie en sortie la matrice 8×8 du bloc de fréquences de la transformée en cosinus discrète correspondante.
2. Évaluer le nombre total d'additions et de multiplications effectuées lors de l'exécution de votre fonction.

Question 2

En fait, l'évaluation de la transformée en cosinus discrète peut se décomposer comme un produit de matrices.

$$\text{DCT} = \frac{1}{4} D \times \text{PIX} \times D^{\top}$$

où PIX est la matrice 8×8 représentant le bloc image, DCT la matrice représentant le bloc issu de la transformée et D une matrice 8×8.

1. Déterminer les coefficients de la matrice D.
2. Écrire une nouvelle fonction OCaml qui renvoie le bloc 8×8 des fréquences de la transformée en cosinus discrète d'un bloc image donné en entrée.
3. Évaluer le nombre total d'additions et de multiplications effectuées lors de l'exécution de votre fonction.

Question 3

1. Montrer que la matrice D peut se décomposer en $D = B \times A$ avec

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

2. Que peut-on dire des multiplications par 1 ou -1 ? En déduire une fonction OCaml évaluant la transformée en cosinus discrète.
3. Évaluer le nombre d'additions et de multiplications effectuées lors de l'exécution de votre fonction.
4. Pouvez-vous améliorer votre fonction ?

CORRIGÉ

Question 1

On suppose que la variable `cos` est une matrice contenant les valeurs pré-calculées de $\cos \frac{(2x_1)i\pi}{16}$.

On donne la fonction de conversion suivante.

```
let mat_dct (pix:int array array) : float array array =
  let dct = Array.make_matrix 8 8 0. in
  for i=0 to 7 do
    for j=0 to 7 do
      for x=0 to 7 do
        for y=0 to 7 do
          dct.(i).(j) <- dct.(i).(j) +. float_of_int pix.(x).(y)
            *. cos.(x).(i) *. cos.(y).(j);
        done
      done
    done
  done;
  dct
```

Le nombre d'additions et de multiplications effectuées lors de l'exécution de cette fonction sont : 8^4 additions et $2 \times 8^4 + 8^2$ multiplications.

Question 2

Les coefficients de la matrice D sont :

$$D = \begin{pmatrix}
\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\
\cos\frac{\pi}{16} & \cos\frac{3\pi}{16} & \cos\frac{5\pi}{16} & \cos\frac{7\pi}{16} & \cos\frac{9\pi}{16} & \cos\frac{11\pi}{16} & \cos\frac{13\pi}{16} & \cos\frac{15\pi}{16} \\
\cos\frac{\pi}{8} & \cos\frac{3\pi}{8} & \cos\frac{5\pi}{8} & \cos\frac{7\pi}{8} & \cos\frac{9\pi}{8} & \cos\frac{11\pi}{8} & \cos\frac{13\pi}{8} & \cos\frac{15\pi}{8} \\
\cos\frac{3\pi}{16} & \cos\frac{9\pi}{16} & \cos\frac{15\pi}{16} & \cos\frac{21\pi}{16} & \cos\frac{27\pi}{16} & \cos\frac{33\pi}{16} & \cos\frac{39\pi}{16} & \cos\frac{45\pi}{16} \\
\cos\frac{\pi}{4} & \cos\frac{3\pi}{4} & \cos\frac{5\pi}{4} & \cos\frac{7\pi}{4} & \cos\frac{9\pi}{4} & \cos\frac{11\pi}{4} & \cos\frac{13\pi}{4} & \cos\frac{15\pi}{4} \\
\cos\frac{5\pi}{16} & \cos\frac{15\pi}{16} & \cos\frac{25\pi}{16} & \cos\frac{35\pi}{16} & \cos\frac{45\pi}{16} & \cos\frac{55\pi}{16} & \cos\frac{65\pi}{16} & \cos\frac{75\pi}{16} \\
\cos\frac{3\pi}{8} & \cos\frac{9\pi}{8} & \cos\frac{15\pi}{8} & \cos\frac{21\pi}{8} & \cos\frac{27\pi}{8} & \cos\frac{33\pi}{8} & \cos\frac{39\pi}{8} & \cos\frac{45\pi}{8} \\
\cos\frac{7\pi}{16} & \cos\frac{21\pi}{16} & \cos\frac{35\pi}{16} & \cos\frac{49\pi}{16} & \cos\frac{63\pi}{16} & \cos\frac{77\pi}{16} & \cos\frac{91\pi}{16} & \cos\frac{105\pi}{16}
\end{pmatrix}$$

En simplifiant, D devient :

$$\begin{pmatrix}
\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\
\cos\frac{\pi}{16} & \cos\frac{3\pi}{16} & \cos\frac{5\pi}{16} & \cos\frac{7\pi}{16} & -\cos\frac{7\pi}{16} & -\cos\frac{5\pi}{16} & -\cos\frac{3\pi}{16} & -\cos\frac{\pi}{16} \\
\cos\frac{\pi}{8} & \cos\frac{3\pi}{8} & -\cos\frac{3\pi}{8} & -\cos\frac{\pi}{8} & -\cos\frac{\pi}{8} & -\cos\frac{3\pi}{8} & \cos\frac{3\pi}{8} & \cos\frac{\pi}{8} \\
\cos\frac{3\pi}{16} & -\cos\frac{7\pi}{16} & -\cos\frac{\pi}{16} & -\cos\frac{5\pi}{16} & \cos\frac{5\pi}{16} & \cos\frac{\pi}{16} & \cos\frac{7\pi}{16} & -\cos\frac{3\pi}{16} \\
\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\
\cos\frac{5\pi}{16} & -\cos\frac{\pi}{16} & \cos\frac{7\pi}{16} & \cos\frac{\pi}{16} & -\cos\frac{\pi}{16} & -\cos\frac{7\pi}{16} & \cos\frac{\pi}{16} & -\cos\frac{5\pi}{16} \\
\cos\frac{3\pi}{8} & -\cos\frac{\pi}{8} & \cos\frac{\pi}{8} & -\cos\frac{3\pi}{8} & -\cos\frac{3\pi}{8} & \cos\frac{\pi}{8} & -\cos\frac{\pi}{8} & \cos\frac{3\pi}{8} \\
\cos\frac{7\pi}{16} & -\cos\frac{5\pi}{16} & \cos\frac{3\pi}{16} & -\cos\frac{\pi}{16} & \cos\frac{\pi}{16} & -\cos\frac{3\pi}{16} & \cos\frac{5\pi}{16} & -\cos\frac{7\pi}{16}
\end{pmatrix}$$

Nous en déduisons une nouvelle fonction OCaml qui renvoie le bloc 8×8 des fréquences de la transformée en cosinus discrète d'un bloc image donné en entrée.

La matrice D est pré-calculée.

```
let mat_dct (pix:int array array) : float array array =
  let dct = Array.make_matrix 8 8 0. in
  let int = Array.make_matrix 8 8 0. in
  for i=0 to 7 do
    for j=0 to 7 do
      for k=0 to 7 do
        int.(i).(j) <- int.(i).(j) +. float_of_int pix.(i).(k)
          *. d.(j).(k)
      done
    done
  done;
```

```
for i=0 to 7 do
  for j=0 to 7 do
    for k=0 to 7 do
      dct.(i).(j) <- dct.(i).(j) +. d.(i).(k) *. int.(k).(j)
    done
  done
done;
for i=0 to 7 do
  for j=0 to 7 do
    dct.(i).(j) <- dct.(i).(j) *. 0.25
  done
done;
dct
```

Les nombres d'addition et de multiplications effectuées lors de l'exécution de cette fonction sont : 2×8^3 additions et $2 \times 8^3 + 8^2$ multiplications.

Question 3

Certaines symétries apparaissent au sein des lignes de la matrice D. Les colonnes de la matrice A représentent ces symétries. La matrice A est inversible.

On en déduit la matrice B suivante (on peut vérifier en effectuant le produit) :

$$B = \begin{pmatrix}
0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\
\cos\frac{\pi}{16} & \cos\frac{3\pi}{16} & \cos\frac{5\pi}{16} & \cos\frac{7\pi}{16} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -\cos\frac{\pi}{8} & -\cos\frac{3\pi}{8} & \cos\frac{3\pi}{8} & \cos\frac{\pi}{8} \\
\cos\frac{3\pi}{16} & -\cos\frac{7\pi}{16} & -\cos\frac{\pi}{16} & -\cos\frac{5\pi}{16} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\
\cos\frac{5\pi}{16} & -\cos\frac{\pi}{16} & \cos\frac{7\pi}{16} & \cos\frac{\pi}{16} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -\cos\frac{3\pi}{8} & \cos\frac{\pi}{8} & -\cos\frac{\pi}{8} & \cos\frac{3\pi}{8} \\
\cos\frac{7\pi}{16} & -\cos\frac{5\pi}{16} & \cos\frac{3\pi}{16} & -\cos\frac{\pi}{16} & 0 & 0 & 0 & 0
\end{pmatrix}$$

Il est bon de remarquer que multiplier par 1 ou -1 revient simplement à prendre en compte un facteur ou son inverse. De plus, les matrices A et B sont creuses, il n'est donc pas intéressant de programmer un produit de matrices.

Nous en déduisons la fonction OCaml suivante :

```
let mat_dct (pix:int array array) : float array array =
  let dct = Array.make_matrix 8 8 0. in
  let pax = Array.make_matrix 8 8 0 in
  (* multiplication par A *)
  for i=0 to 3 do
    for j=0 to 7 do
      pax.(i).(j) <- pix.(i).(j) - pix.(7-i).(j);
      pax.(7-i).(j) <- pix.(i).(j) + pix.(7-i).(j)
    done
```

```
done;
for i=0 to 7 do
  for j=0 to 3 do
    pax.(i).(j) <- pix.(i).(j) - pix.(i).(7-j);
    pax.(i).(7-j) <- pix.(i).(j) + pix.(i).(7-j)
  done
done;
(* multiplication par B *)
let int = Array.make_matrix 8 8 0. in
for i=0 to 7 do
  for j=0 to 7 do
    let t = if j mod 2 = 0 then 4 else 0 in
    for k=0 to 3 do
      int.(i).(j) <- int.(i).(j)
        +. float_of_int pax.(i).(k+t) *. b.(j).(k+t)
    done
  done
done;
for i=0 to 7 do
  for j=0 to 7 do
    let t = if j mod 2 = 0 then 4 else 0 in
    for k=0 to 3 do
      dct.(i).(j) <- dct.(i).(j) +. b.(i).(t+k) *. int.(k+t).(j)
    done
  done
done;
dct
```

Les nombres d'additions et de multiplications effectuées lors de l'exécution de cette fonction sont : $2 \times 8^2 + 8^3$ additions et $8^3 + 8^2$ multiplications.

On peut réduire ce nombre de multiplications : en multipliant B par $\sqrt{2}$, on obtient deux lignes de 1.

⋆⋆⋆⋆⋆⋆⋆⋆

8

Vers la récursivité

« *Diviser pour régner. Faire moins, pour faire plus.* »

8.1 L'angoisse de la panne sèche

Une route comporte n stations-service. La première est à une distance d_1 du départ, la la deuxième est à une distance d_2 de la première, la troisième à une distance d_3 de la deuxième, etc. La fin de la route est à une distance d_{n+1} de la n-ième station-service. Les distances d_i sont représentées par un tableau de flottants.

Un automobiliste prend le départ de la route avec une voiture dont le réservoir d'essence est plein. Sa voiture est capable de parcourir une distance r (mais pas plus !) avec un plein. On suppose que r est supérieur ou égal à chacun des d_i et inférieur à leur somme, sinon le problème n'a pas de sens.

Question 1

L'automobiliste désire faire le plein le moins souvent possible. Écrire une fonction OCaml qui détermine à quelles stations-service il doit s'arrêter.

Question 2

Maintenant, l'automobiliste part avec un réservoir vide. Il doit au départ acheter de l'essence (autant qu'il veut dans la contenance de son réservoir) au prix de E_0 euros par litre. Par la suite, à la station numéro i, l'essence coûte E_i euros par litre. L'automobiliste consomme kt litres pour parcourir une distance t, et le réservoir peut contenir L litres d'essence. Écrire une fonction OCaml qui indique à quelles stations-service l'automobiliste doit s'arrêter, et combien de litres il doit prendre à chaque fois, pour que son trajet lui coûte le moins cher possible.

CORRIGÉ

Question 1

Un algorithme glouton est optimal : on regarde jusqu'où on peut aller et on s'arrête à la première station avant ce point, puis on recommence.

```
let rapide (d:int array) (r:int) : int list =
  let n = Array.length d in
  let rec aux acc i reste =
    if i=n then acc else
    if d.(i) > reste then
      aux (i::acc) (i+1) (r-d.(i))
    else
      aux acc (i+1) (reste-d.(i))
  in List.rev (aux [] 0 r)
```

Cet algorithme est bien optimal. En effet, soit une autre solution. Son premier plein est forcément au même endroit ou avant le premier plein de la solution de l'algorithme. Par récurrence, son i-ième plein se produit au même endroit ou avant le i-ième plein de l'algorithme.

Question 2

Repérons tout d'abord la station où l'essence est la moins chère, et faisons deux remarques :

- L'automobiliste doit faire en sorte que son réservoir soit vide au moment où il arrive à cette station (risqué en pratique !). En effet, s'il lui reste x litres d'essence à ce moment là, il les a payés plus cher auparavant pour rien.
- Toujours à cette station, il doit prendre juste ce qu'il faut d'essence pour aller jusqu'au bout si la capacité de son réservoir le permet, soit remplir complètement son réservoir.

On recommence pour les deux sous-problèmes consistant à aller du début à cette station-service et de cette station service à la fin. Pas besoin de récursivité, la stratégie qui s'en déduit est la suivante. À chaque fois que l'on passe devant une station service :

- Soit il existe dans le rayon d'action que peut atteindre la voiture avec un plein des stations moins chères, dans ce cas, on prend le minimum d'essence (éventuellement pas du tout) permettant d'aller à la première station moins chère que celle où l'on se trouve (attention : la première, c'est-à-dire pas nécessairement la moins chère des stations accessibles).
- Soit toutes les stations que l'on peut atteindre sont plus chères, dans ce cas on remplit complètement le réservoir, sauf si on peut atteindre l'arrivée, dans quel cas on prend juste ce qu'il faut pour y aller.

L'écriture de la fonction OCaml ne représente plus aucune difficulté, elle est laissée au lecteur.

⋆⋆⋆⋆⋆⋆⋆⋆

8.2 Produit de plusieurs matrices

On cherche à effectuer un produit de matrices

$$M_1 \times M_2 \times ... \times M_n$$

où la matrice i comporte r_{i-1} lignes et r_i colonnes, en effectuant la moins possible de multiplications de réels (la multiplication de deux matrices se fera par la méthode usuelle).

Question 1

Écrire une fonction qui multiplie une matrice A de taille $p \times q$ par une matrice B de taille $q \times r$. Combien de multiplications de réels nécessite-t-elle ?

Question 2

On effectue le produit $M_1 \times M_2 \times M_3 \times M_4$, où M_1 est de taille 10×20, M_2 de taille 20×50, M_3 de taille 50×1 et M_4 de taille 1×100. Combien de multiplications de réels fera-t-on si on calcule le produit dans les deux ordres (indiqués par les parenthèses) suivants :

$$M_1 \times (M_2 \times (M_3 \times M_4))$$

$$(M_1 \times (M_2 \times M_3)) \times M_4$$

Question 3

Écrire une fonction pour trouver le plus petit nombre de multiplications de réels nécessaires pour calculer $M_0 \times M_1 \times ... \times M_{n-1}$, où la matrice i comporte r_{i-1} lignes et r_i colonnes. Aide : définir une valeur `m.(i).(k)` égale au plus petit nombre de multiplications pour calculer $M_i \times M_{i+1} \times ... \times M_k$.

CORRIGÉ

Question 1

On a classiquement :

```
type matrix = float array array
let produit (a:matrix) (b:matrix) : matrix =
  let p = Array.length a in
  let q = Array.length b in
  let r = Array.length b.(0) in
  let c = Array.make_matrix p r 0. in
  for i=0 to p-1 do
    for j=0 to r-1 do
      for k=0 to q-1 do
        c.(i).(j) <- c.(i).(j) +. a.(i).(k) *. b.(k).(j)
```

```
      done
    done
  done;
  c
```

Cette fonction utilise $p \times q \times r$ multiplications.

Question 2

Il faut $50 \times 1 \times 100 + 20 \times 50 \times 100 + 10 \times 20 \times 100 = 125\,000$ multiplications pour la première méthode et $20 \times 50 \times 1 + 10 \times 20 \times 1 + 10 \times 1 \times 100 = 2200$ pour la seconde méthode.

Question 3

Il suffit de voir comment s'obtient `m.(i).(j)`. On trouve aisément :

$$\texttt{m.(i).(j)} = \begin{cases} 0 & \text{si } i = j \\ \min_{i \leq k < j} \{\texttt{m.(i).(k)} + \texttt{m.(k+1).(j)} + r_{i-1} r_k r_j\} & \text{sinon} \end{cases}$$

Il n'est pas nécessaire d'écrire une fonction récursive, on itère sur la longueur des chaînes de multiplication.

```
let nb_min_multiplications (r:int array) : int =
  let n = Array.length r - 1 in
  let m = Array.make_matrix n n 0 in

  for l = 2 to n do
    for i = 0 to n - l do
      let j = i + l - 1 in
      m.(i).(j) <- max_int;
      for k = i to j - 1 do
        let q = m.(i).(k) + m.(k+1).(j) + r.(i) * r.(k+1) * r.(j+1) in
        if q < m.(i).(j) then
          m.(i).(j) <- q
      done
    done
  done;
  m.(0).(n-1)
```

⋆⋆⋆⋆⋆⋆⋆⋆

8.3 Les deux points les plus proches

Le but de ce problème est d'identifier, dans un ensemble de points, quel est le couple de points les plus proches au sens de la distance euclidienne. Ce type d'algorithme voit son utilité dans les transports aériens ou maritimes.

L'ensemble de points est vu comme un tableau de 2-tuples de coordonnées (x, y).

On dispose de la fonction `sqrt (x:float) : float` qui renvoie la racine carrée de `x`.

Question 1

Donner un algorithme itératif qui détermine le couple de points les plus proches de l'ensemble.

Montrer qu'à chaque itération, votre algorithme répond aux conditions que vous vous êtes fixées (c'est-à-dire vérifie un invariant de boucle).

Évaluer le nombre d'additions, de multiplications et d'appels à la fonction `sqrt`.

L'ensemble de points est stocké dans deux tableaux `x` où les points sont ordonnés suivant les abscisses croissantes, et `y` où les points sont ordonnés suivant les ordonnées croissantes.

Question 2

Écrire une fonction OCaml `decoupe (x:point array) (y:point array) (y':point array) (deb:int) (fin:int) : unit` qui place les points du sous-tableau `x`[`deb`..`med`] classés par ordonnée croissante dans le sous-tableau `y'`[`deb`..`med`] et les éléments du sous-tableau `x`[`med+1`..`fin`] classés par ordonnée croissante dans le sous-tableau `y'`[`med+1`..`fin`], où `med` $= \left\lfloor \frac{\texttt{deb}+\texttt{fin}}{2} \right\rfloor$.

Évaluer le nombre de transferts effectués lors de l'exécution de cette fonction.

Question 3

Nous supposons que A et B sont les points les plus proches du sous-tableau `x`[`deb`..`med`] et, C et D sont les points les plus proches du sous-tableau `x`[`med+1`..`fin`].

Écrire un algorithme linéaire qui détermine les deux points les plus proches de l'ensemble `x`[`deb`..`fin`], connaissant A, B, C, D et `y`[`deb`..`fin`].

En déduire une fonction de recherche du couple de points les plus proches (où A, B, C et D sont déterminés de façon classique).

Déterminer la complexité de votre fonction.

Que peut-on conclure ?

CORRIGÉ

Question 1

Nous proposons un algorithme exhaustif où tous les couples de points sont examinés.

```
type point = int * int

let dist_sq (p1:point) (p2:point) : int =
  let x1, y1 = p1 and x2, y2 = p2 in
  (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1)
let couple (tab:point array) : (point * point) =
  let n = Array.length tab in
```

```
  let min_dist = ref max_int in
  let p1 = ref (0,0) and p2 = ref (0,0) in

  for i=0 to n-2 do
    for j=i+1 to n-1 do
      let dist = dist_sq tab.(i) tab.(j) in
      if dist < !min_dist then (
        min_dist := dist;
        p1 := tab.(i);
        p2 := tab.(j)
      )
    done
  done;
  (!p1, !p2)
```

Dans la boucle i : `p1`, `p2` est le couple de points les plus proches avec `p1` pris parmi les i premiers points de `tab`.
Dans la boucle j : `p1`, `p2` est le couple de points les plus proches avec `p1` pris parmi les $i-1$ premiers points de `tab` avec `p1=tab.(i)` et `p2` pris parmi les j premiers points de `tab`.

Le nombre d'additions est $3 \times (n-1)n/2$, de multiplications $(n-1)n$, où n est le nombre de points. Il n'y a pas besoin d'appeler la fonction `sqrt`.

Question 2

Il s'agit simplement de transcrire ce qui est proposé dans l'énoncé :

```
let decoupe (x:point array) (y:point array) (y':point array)
    (deb:int) (fin:int) : unit =
  let med = (deb + fin) / 2 in
  let i = ref deb in
  let j = ref (med + 1) in
  for k=deb to fin do
    if fst y.(k) > fst x.(med) then (
      y'.(!j) <- y.(k);
      incr j
    )
    else (
      y'.(!i) <- y.(k);
      incr i
    )
  done
```

Le nombre de transferts est de l'ordre du nombre d'éléments de `x`.

Question 3

Suivant les instructions de l'énoncé, le tableau `x` est découpé en deux parties. Dans chacune d'entre elles, nous avons recherché le couple de points les plus proches.

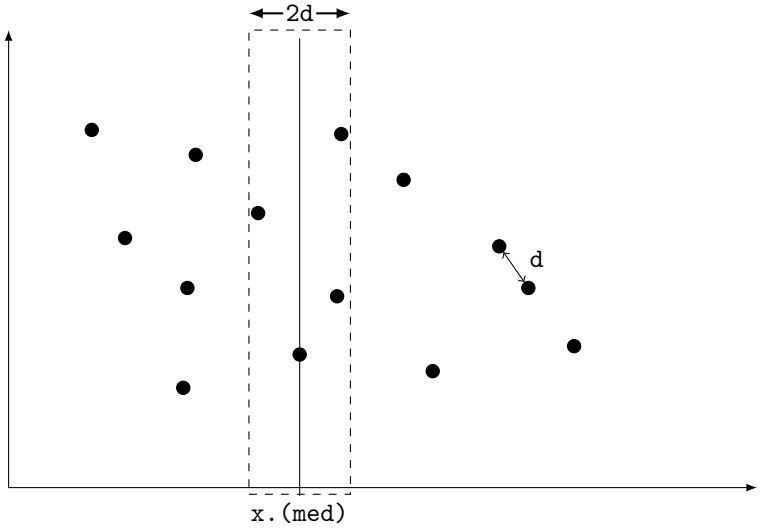
Mais la distance la plus courte peut être donnée par un couple de points à cheval sur les deux parties.

Supposons que nous ayons comparé les deux distances et gardé le couple donnant la plus petite distance `dist`, alors via la fonction `termine`, nous allons rechercher s'il existe un couple de points pris dans chacune des deux parties, dont la distance est plus petite.

Nous reprenons la même terminologie pour les variables.

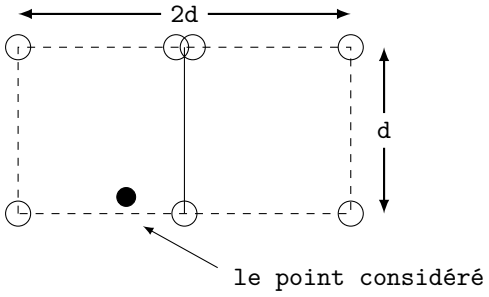
```
let termine (x:point array) (y:point array) (deb:int) (fin:int)
    (p1:point) (p2:point) : (point * point) * int =
  let dist = ref (dist_sq p1 p2) in
  let n = Array.length x in
  let z = Array.make n (0,0) in
  let t = ref 0 in

  let med = (deb + fin) / 2 in
  let a = fst x.(med) - !dist in
  let b = fst x.(med) + !dist in
  (* On ajoute tous les candidats potentiels *)
  for k=deb to fin do
    if a < fst y.(k) && fst y.(k) < b then (
      z.(!t) <- y.(k);
      incr t
    )
  done;
  (* On cherche une plus petite distance *)
  let p1 = ref p1 and p2 = ref p2 in
  for k=0 to !t-2 do
    let i = ref 1 in
    while !i + k < !t && (snd z.(k + !i)) - (snd z.(k)) < !dist do
      let d = dist_sq z.(k) z.(k + !i) in
      if d < !dist then (
        p1 := z.(k);
        p2 := z.(k+1);
        dist := d
      );
      incr i
    done
  done;
  (!p1, !p2), !dist
```

On ne considère que les points contenus dans le rectangle en pointillés, en partant du point le plus bas.



Si `d` est la plus petite distance pour chacune des deux moitiés, nous avons au plus 4 points dans un carré de côté `d`.

Ainsi, dans la boucle `while`, il y a au plus 7 itérations. On représente ici le rectangle examiné dans la boucle `while` :



Nous pouvons ainsi envisager la fonction récursive suivante :

```
let cherche (x:point array) (y:point array) : point * point =
  let n = Array.length x in
  let y' = Array.copy y in

  let rec cherche_aux deb fin =
    if fin = deb then
      ((0,0), (0,0)), max_int
    else if fin = deb + 1 then
      (x.(deb), x.(fin)), dist_sq x.(deb) x.(fin)
    else (    (* fin > deb+1 *)
      let med = (deb+fin)/2 in
```

```
      decoupe x y y' deb fin;
      let (a, b), d1 = cherche_aux deb med in
      let (c, d), d2 = cherche_aux (med+1) fin in
      let p1, p2 = if d1 < d2 then  a, b else c, d in
      termine x y deb fin p1 p2
    )
  in let (p1, p2), d = cherche_aux 0 (n-1) in
  p1, p2
```

Les fonctions `decoupe` et `termine` ont une complexité linéaire (c'est à dire en $O(n)$). La complexité de la fonction `cherche` lorsque `deb=0` et `fin=n-1` est donnée par la formule de récurrence suivante : $C(n) = 2 \cdot C(n/2) + O(n)$.

La complexité de `cherche` est donc en $O(n \log_2 n)$.

⋆⋆⋆⋆⋆⋆⋆⋆

8.4 Hanoï

Les tours de Hanoï est un exemple très connu pour sa programmation récursive. Nous nous intéressons ici à sa version itérative.

Le jeu en lui même est très simple. Nous disposons de trois colonnes sur lesquelles sont empilés des disques de taille différentes.

Les règles sont les suivantes : sur chaque colonne, un disque plus gros ne peut pas être posé sur un disque plus petit, on ne peut déplacer qu'un seul disque à la fois.

Au départ, tous les disques sont empilés sur la première colonne. Le joueur doit en un minimum de coups déplacer tous les disques de la première colonne à la dernière colonne.

Question 1

Donner les déplacements à faire avec trois disques. Puis avec quatre disques.

Question 2

Montrer que le plus petit disque se déplace de la même façon un coup sur deux. Montrer que pour chaque autre déplacement, un seul choix est possible.

Question 3

Numérotons les coups en partant de 1.

Déterminer la relation qu'il y a entre le numéro du disque à déplacer et le numéro du coup.

Pour n disques, combien de déplacements effectuez-vous ? Est-ce optimal ?

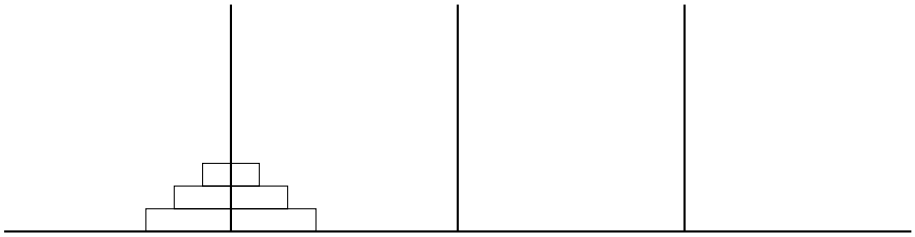
Écrire le programme affichant tous les coups effectués.

CORRIGÉ

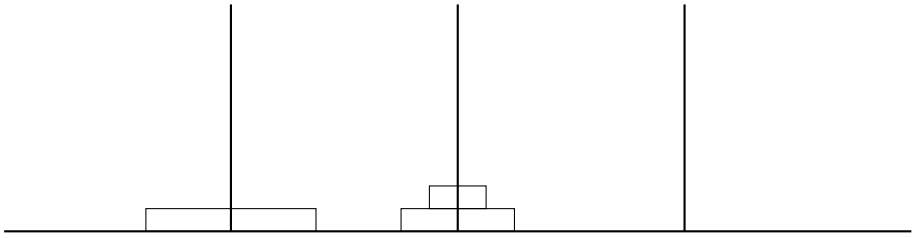
Question 1

Nous donnons dans les tableaux suivants les positions de chaque disque au cours du temps. Les colonnes sont numérotées 0, 1 et 2. Les disques sont nommés du plus petit au plus grand d_0, d_1, ..., d_n. Tous les disques sont sur la colonne 0 au départ.

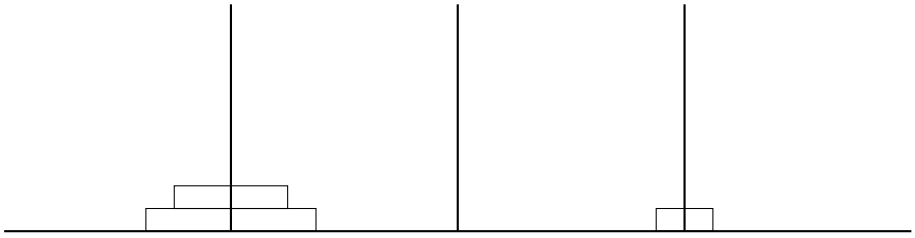
Pour trois disques nous avons :



Comme nous devons déplacer le plus grand disque de la colonne 0 à la colonne 2, la configuration suivante est un passage obligé :



Pour obtenir cette configuration, le disque immédiatement inférieur au plus grand a dû être déplacé de la colonne 0 sur la colonne 1 d'où un autre passage obligé :



Nous en déduisons le tableau suivant :

coups	0	1	2	3	4	5	6	7
d_0	0	2	2	1	1	0	0	2
d_1	0	0	1	1	1	1	2	2
d_2	0	0	0	0	2	2	2	2

Pour quatre disques, nous avons, en suivant le même raisonnement :

coups	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
d_0	0	1	1	2	2	0	0	1	1	2	2	0	0	1	1	2
d_1	0	0	2	2	2	2	1	1	1	1	0	0	0	0	2	2
d_2	0	0	0	0	1	1	1	1	1	1	1	1	2	2	2	2
d_3	0	0	0	0	0	0	0	0	2	2	2	2	2	2	2	2

Question 2

Les deux exemples précédents mettent en évidence une dépendance entre la parité du nombre de disques et le sens de déplacement de ceux-ci.

Tous les coups impairs, le disque d_0 se déplace de la colonne x vers la colonne $x+(-1)^n$ mod 3, où n représente le nombre de disques.

Comme on ne peut rien mettre sur le disque d_0, le seul choix possible est de déplacer le plus petit des disques situés sur les deux autres colonnes. Maintenant, seul le disque d_0 est déplaçable sinon il y a retour à la situation précédente. Nous vérifions ainsi que d_0 est à déplacer un coup sur deux.

Question 3

En fait, le disque d_1 se déplace comme s'il était le disque 0 des $n-1$ disques restants en ne prenant en compte que les coups pairs. Ce qui revient à un déplacement tous les coups pairs (multiple de deux) non multiples de quatre.

Ainsi, le disque d_i se déplace de $(-1)^{n-1}$ tous les coups multiples de 2^i non multiples de 2^{i+1}.

Ainsi, le disque d_{n-1} se déplace au coup 2^{n-1} lorsque tous les autres disques sont sur la colonne 1. Le temps de les replacer sur la colonne 2 est le même que celui pour les placer sur 1, c'est-à-dire $2^{n-1}-1$.

D'où le temps total pour n disques : 2^n-1. On peut maintenant le montrer par récurrence.

Est-il optimal ? Oui, car pour déplacer le plus grand des disques, il faut avoir déplacé tous les autres sur l'autre colonne, ce qui donne la formule de récurrence suivante, où $T(n)$ représente le nombre de coups pour n disques :

$$T(n) = 2 \cdot T(n-1) + 1$$

Nous obtenons $T(n) = 2^n - 1$.

Voici un exemple de programme où :

```
let deplacements (n:int) : unit =
  let p = Int.shift_left 1 n in (* 2^n *)
  for i=1 to p-1 do
    let t = ref i and k = ref 0 in
    while !t mod 2 = 0 do
      t := !t / 2;
      incr k
    done;
    let num_disque = !k in
    let deplacement = -2*((n - !k) mod 2) + 1 in
    (* Coup à jouer *)
    let dir = if deplacement = -1 then "gauche" else "droite" in
    Printf.printf "Déplacer le disque %d vers la %s\n" num_disque dir
  done
```

La complexité en termes déplacement est optimale. On peut calculer le nombre de comparaisons de la façon suivante (les seules comparaisons effectuées sont dans la condition de la boucle `while`) :

- La moitié des nombres binaires se terminent par un 1. Dans ce cas, on effectue une comparaison.
- Parmi ceux se terminant par un 0, la moitié possède un 1 en deuxième position. Dans ce cas, on effectue 2 comparaisons.
- Parmi ceux se terminant par deux zéros, la moitié possède un 1 en troisième position. Dans ce cas, on effectue 3 comparaisons.
- De manière récurrente, on a donc une proportion $\frac{1}{2^n}$ des nombres qui nécessitent n comparaisons.

Donc en sommant tous les cas possibles, le nombre de comparaisons N est donné par la formule :

$$N = 2^n \cdot \sum_{k=1}^{n} k \frac{1}{2^k} = 2^{n-1} \cdot \sum_{k=1}^{n} k \left(\frac{1}{2}\right)^{k-1}$$

En remarquant que :

$$\sum_{k=1}^{n} k x^{k-1} = \frac{\mathrm{d}}{\mathrm{d}x} \sum_{k=1}^{n} x^k = \frac{\mathrm{d}}{\mathrm{d}x} \frac{x - x^{n+1}}{1 - x}$$

on obtient $N = 2^{n+1} - (n + 2)$.

On peut si on le désire également donner la version récursive suivante où A, B et C représentent les trois colonnes :

`Hanoï`(n, A, B, C)
    si $n \geq 1$ faire
        `Hanoï`($n - 1$, A, C, B)
        Déplacer le disque de A vers C
        `Hanoï`($n - 1$, B, A, C)

⋆⋆⋆⋆⋆⋆⋆⋆