Passeport pour l'informatique

EXERCICES D'ALGORITHMIQUE

Oraux d'ENS

Ouvrage collectif — Coordination Jean-Claude Bajard

INTERNATIONAL™ THOMSON PUBLISHING

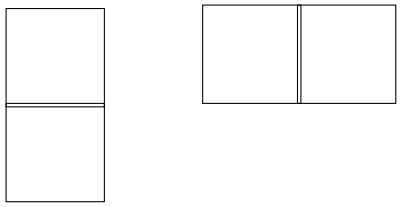
1

Géométrie et Images

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

1.1 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 :

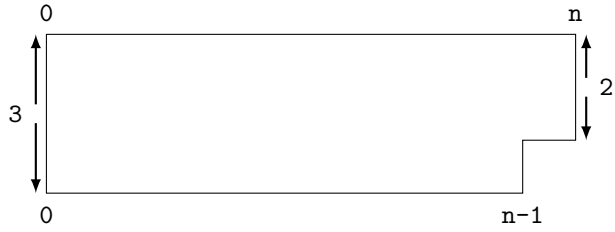
```
let largeur2 (n:int) : int =
  if n=1 then 1 else
  if n=2 then 2 else

  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 :

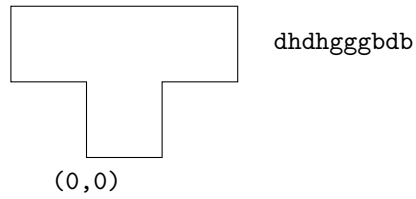
$$\frac{\log_2(u_n)}{n} \xrightarrow[n\to\infty]{} \sum_{r\geq 0} \frac{(-1)^r}{\pi(2r+1)^2} \approx 0.2916...$$

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

⋆⋆⋆⋆⋆⋆⋆⋆

1.2 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.

⋆⋆⋆⋆⋆⋆⋆⋆

1.3 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

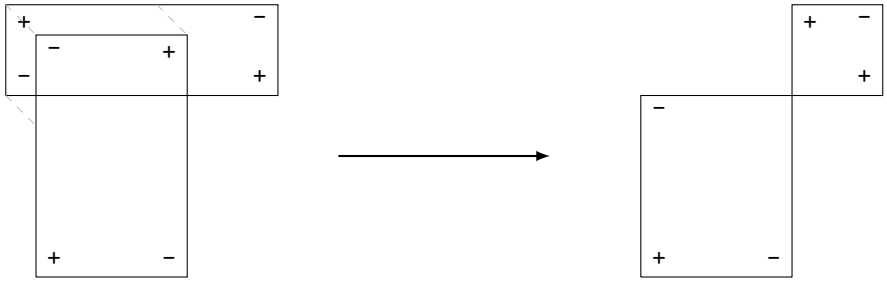
Montrer qu'un ensemble S avec poids peut être décrit comme somme d'ensembles si et seulement si la somme des poids de ses éléments est nulle.

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.

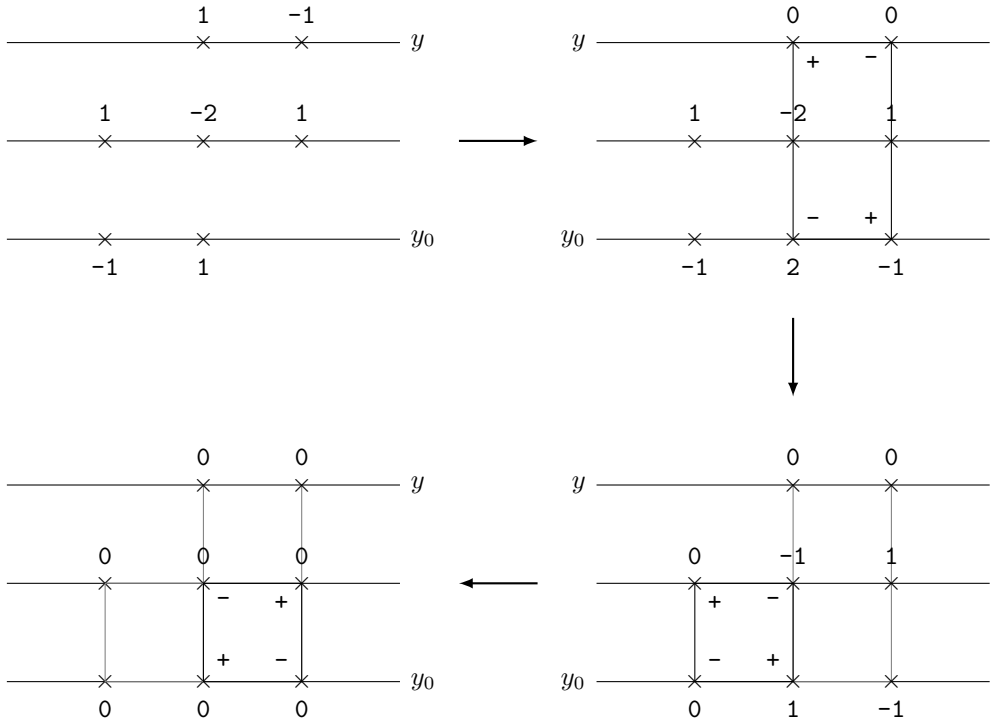
Question 4

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.

⋆⋆⋆⋆⋆⋆⋆⋆

1.4 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 : $\det(u, v) = ||u|| \cdot ||v|| \cdot \sin(u, v)$.

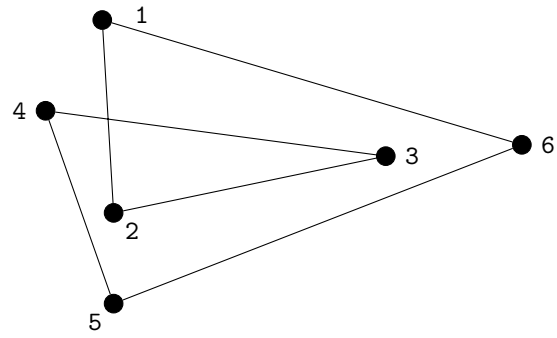
```
let saillant (a:point) (b:point) (c:point) : bool =
  let ax, ay = a and bx, by = b and cx, cy = c in
  let det = (ax - bx) * (cy - by) - (ay - by) * (cx - bx) in
  det > 0
```

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.

⋆⋆⋆⋆⋆⋆⋆⋆

1.5 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 donnés 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 ?

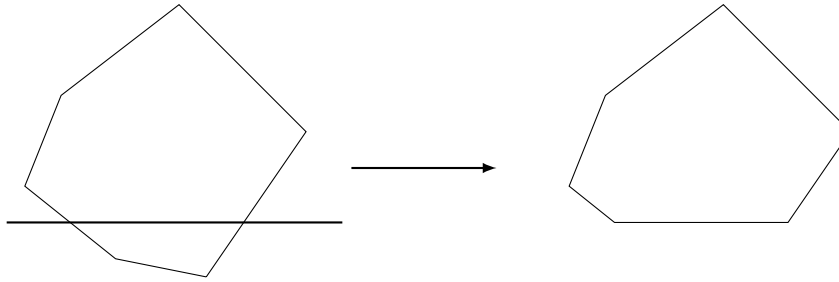
CORRIGÉ

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).

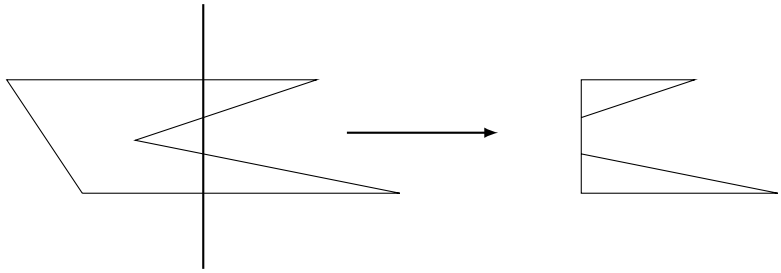
```
type fenetre = point * point
let clipping (w:fenetre) (p:polygone) =
  let (x1, y1), (x2, y2) = w in
  let p1 = clip_bord 0 y1 p in
  let p2 = clip_bord 1 y2 p1 in
  let p3 = clip_bord 2 x1 p2 in
  clip_bord 3 x2 p3
```

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.

⋆⋆⋆⋆⋆⋆⋆⋆

1.6 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 donnés 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 donnés 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 donnés 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`

possède 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 ?

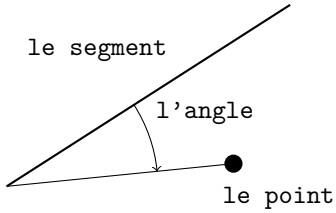
CORRIGÉ

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 $\leq_\alpha$).