Concours Centrale 2005

Corrigé de l'épreuve d'informatique

Partie I : Polygones simples

Remarque : l'introduction demande de manipuler des fonctions curryfiées, alors que ces mêmes fonctions sont non curryfiées dans les énoncés des questions.

A.1)

```
let determinant u v = u.xv*v.yv-u.yv*v.xv;;

let produit_scalaire u v = u.xv*v.xv+u.yv*v.yv;;
```

A.2) Notons tout d'abord que les points b et c doivent être distincts de a pour que l'angle $(\overrightarrow{ab}, \overrightarrow{ac})$ soit défini. En notant alors θ une mesure de cet angle, nous avons $\sin(\theta) = \dfrac{\det(\overrightarrow{ab}, \overrightarrow{ac})}{||\overrightarrow{ab}|| \times ||\overrightarrow{ac}||} > 0 \iff \det(\overrightarrow{ab}, \overrightarrow{ac}) > 0$ et donc (a, b, c) est direct si et seulement si $\det(\overrightarrow{ab}, \overrightarrow{ac}) > 0$.

A.3)

```
let direct a b c = let d= determinant (CreerVecteur a b) (CreerVecteur a c) in
      if d>0 then
         1
      else if d<0 then
         -1
      else
         0;;
```

A.4) Cette question n'a de sens que si l'on impose aux segments d'être définis par deux points distincts. En notant $a =$ `I.a` et $b =$ `I.b`, nous avons trois cas possibles :

- p et q sont du même côté (strict) de la droite D, et (a, b, p) et (a, b, q) sont tous les deux directs, ou tous les deux indirects : `direct(a,b,p)` `direct(a,b,q)` vaut alors 1 ;
- p ou q est sur le droite D et `direct(a,b,p)` `direct(a,b,q)` vaut 0 ;
- p et q sont chacun d'un côté différent de D et `direct(a,b,p)` `direct(a,b,q)` vaut alors -1.

On obtient donc la fonction :

```
let meme_cote I p q = (direct I.a I.b p)*(direct I.a I.b q);;
```

A.5) p est sur le segment I si et seulement si p est sur la droite D engendrée par I et si, une droite D_1 passant par p distincte de D étant choisie arbitrairement, les extrémités de I ne sont pas d'un même côté de D_1. Nous utiliserons donc la démarche suivante, où a et b désigne les extrémités de I :

- si `direct(a,b,p)` est non nul, on renvoie la valeur -1 : p n'est pas sur la droite D ;
- sinon, on distingue deux cas :

- si $\overrightarrow{ab}$ a une première coordonnée non nulle, on choisit pour D_1 une droite verticale : on définit le segment J d'extrémités p et $p+(0,1)$;
- sinon, on choisit pour D_1 une droite horizontale : on définit le segment J d'extrémités p et $p+(1,0)$.

- si `meme_cote J a b` prend la valeur 1, on renvoie -1 ;
- sinon, on renvoie 1.

```
let appartient I p = match direct I.a I.b p with
     0 -> if I.a.x<>I.b.x then
             let J={a=p;b={x=p.x;y=p.y+1}} in
                if meme_cote J I.a I.b=1 then -1 else 1
          else
             let J={a=p;b={x=p.x+1;y=p.y}} in
                if meme_cote J I.a I.b=1 then -1 else 1
     |_ -> -1 ;;
```

A.6) Notons a_1 et b_1 (resp. a_2 et b_2) les extrémités de I et de J et D_1 (resp. D_2) la droite portée par I (resp. par J). Notons également ε_1 (resp. ε_2) la valeur renvoyée par l'instruction `meme_cote` J a_1 b_1 (resp. `meme_cote` I a_2 b_2). Plusieurs cas peuvent être distingués :

- Si $\varepsilon_1 = 1$, le segment J est contenu un demi-plan ouvert délimité par D_1 et donc $I \cap J = \emptyset$. Symétriquement, $I \cap J = \emptyset$ quand $\varepsilon_2 = 1$;
- Si $\varepsilon_1 = \varepsilon_2 = -1$, les droites D_1 et D_2 sont sécantes en un point p ; comme a_1 et b_1 (resp. a_2 et b_2) ne sont pas du même côté de D_2 (resp. de D_1), $p \in J$ (resp. $p \in J$) : $I \cap J \neq \emptyset$.
- Si $\varepsilon_1 = 0$ et $\varepsilon_2 = -1$, un seul des points a_1 ou b_1 appartient à D_2 (car $\varepsilon_2 \neq 0$). Par symétrie, on peut supposer que $a_1 \in D_2$ et $b_1 \notin D_2$. Les droites D_1 et D_2 sont alors sécantes en a_1 et a_2 et b_2 ne sont pas du même côté de D_1 : a_1 est donc élément de J et $I \cap J \neq \emptyset$.
- Si $\varepsilon_1 = -1$ et $\varepsilon_2 = 0$, on a comme ci-dessus $I \cap J \neq \emptyset$.
- Si $\varepsilon_1 = \varepsilon_2 = 0$, les quatre points sont alignés et $D_1 = D_2$. On montre facilement que $I \cap J$ est non vide si et seulement si $a_1 \in J_2$ ou $a_2 \in J$ ou $b_2 \in J$.

Cette analyse permet d'écrire la fonction suivante :

```
let intersecte I J = match (meme_cote I J.a J.b , meme_cote J I.a I.b) with
   (1,_) -> -1
   | (_,1) -> -1
   | (-1,_) -> 1
   | (_,-1) -> 1
   | (_,_) -> if appartient J I.a=1 or appartient I J.a=1 or appartient I J.b=1 then
                 1
              else
                 -1 ;;
```

B.1) L'énoncé n'est pas assez précis, le domaine de variation de l'indice i n'étant pas défini. En notant $[p_1; p_2; \ldots; p_n]$ la liste définissant le polygone, deux problèmes se posent :

- peut-on prendre $i = n$, et doit-on supprimer p_n quand p_{n-1}, p_n et p_1 sont alignés ?
- peut-on prendre $i = 1$, et doit-on supprimer p_1 quand p_n, p_1 et p_2 sont alignés ?

Je ne suis pas certain que l'auteur du sujet ait pensé aux difficultés posées par le second problème. Je donne ici une solution supprimant tous les sommets inutiles, y compris (éventuellement) le premier et le dernier.

Il faut parcourir la liste mais en gardant en mémoire le point de départ, qui sert à effectuer le dernier test d'alignement, et en calculant le dernier point de la liste, pour supprimer, au besoin, le point p_1. Nous utilisons donc une procédure récursive auxiliaire `simpl` qui, appliquée à un point p et à une liste de points $L = [p_1; p_2; \ldots; p_n]$, supprime tout point aligné avec son prédécesseur et son successeur (le successeur de p_n étant égal à p), et renvoie le couple formé par la nouvelle liste et le dernier point non supprimé. La méthode est élémentaire :

- si $n = 2$ et si (p_1, p_2, p) ne sont pas alignés, on renvoie $[p_1; p_2], p_2$;
- si $n = 2$ et si (p_1, p_2, p) sont alignés, on renvoie $[p_1], p_1$;
- si $n \geq 3$ et si (p_1, p_2, p_3) ne sont pas alignés, on calcule le couple $(L1, q) =$ `supp` $p\,[p2; p3; \ldots; p_k]$ et on renvoie $(p_1 :: L1, q)$;
- si $n \geq 3$ et si (p_1, p_2, p_3) sont alignés, on renvoie `supp` $p\,[p1; p3; \ldots; p_n]$.

La fonction `supprimer` s'obtient ensuite en appliquant `supp` à p_1 et $[p_1; p_2; \ldots; p_n]$: on obtient un couple (L, q) ; si q est aligné avec p_1 et p_2, on renvoie L privé de son premier élément ; sinon, on renvoie L. Cela donne :

```
let rec supp p = function
   [p1;p2] -> if direct p1 p2 p =0 then
              [p1],p1
           else
              [p1;p2],p2
   | p1::p2::p3::L -> if direct p1 p2 p3 = 0 then
              supp p (p1::p3::L)
           else
              let couple = supp p (p2::p3::L) in p1::fst(couple),snd(couple);;

let supprime = function
   p1::L1 -> let (L,q)=supp p1 (p1::L1) in
              if direct p1 hd(L1) q = 0 then
                 tl L
              else
                 L ;;
```

Le temps de calcul est linéaire en le nombre de points du polygone.

B.2) Un point p sera en dehors de `P` dès que son abscisse sera plus grande que le maximum des abscisses des points de `P`. Il suffit donc de calculer récursivement l'abscisse maximale M des sommets du polygone, puis de renvoyer (par exemple) le point $\{x = M + 1; y = 0\}$: le temps de calcul sera bien linéaire en le nombre de points de `P`. La procédure `maximum` fonctionne avec des listes non vides.

```
let rec maximum = function
   [p] -> p.x
   | p::L -> max (maximum(L)) p.x ;;

let en_dehors P = {x=maximum(P)+1;y=0};;
```

B.3) L'énoncé est encore ambigu : pour parler de la composante connexe contenant q, il ne faut pas que q soit élément de P, ce que nous supposerons donc. Comme $[s,q]$ ne contient aucun sommet de P, la droite D portée par $[s,q]$ ne contient aucun des segments de P et le nombre de points d'intersection de $[s,q]$ avec P est fini. D'autre part, à chaque point d'intersection de $[s,q]$ et de P, le segment $[s,q]$ "traverse" la frontière du polygone et passe d'une composante connexe à l'autre. Comme s est à l'extérieur de P, s est à l'intérieur de P si le nombre de points d'intersection est impair, et à l'extérieur de P sinon.

B.4) On a une nouvelle fois besoin d'une procédure récursive auxiliaire `calcul` qui, appliquée à une liste $[p_1, p_2, \ldots, p_n]$ de points, à un point p, à un point s et à un points q, renvoie la valeur :

- 0 si $q \in [p_1,p_2] \cup \ldots [p_{n-1},p_n] \cup [p_n,p]$;
- 1 si $q \notin [p_1,p_2] \cup \ldots [p_{n-1},p_n] \cup [p_n,p]$ et $[s,q]$ intersecte un nombre impair des n segments $[p_1,p_2]$, …, $[p_{n-1},p_n]$, $[p_n,p]$;
- -1 sinon.

La méthode est la suivante :

- si L est réduit à $[p_1]$: si q appartient à $[p,p_1]$, on renvoie 0 ; sinon, on renvoie le résultat de la fonction `intersecte` appliquée aux segments $[s,q]$ et $[p,p_1]$;
- sinon, on considère les deux premiers points de L : si $q \in [p_1,p_2]$, on renvoie 0 ; sinon, on calcule le résultat ε_1 de la fonction `intersecte` appliquée aux segments $[s,q]$ et $[p_1,p_2]$ ainsi que le résultat ε_2 de la fonction `epsilon` appliquée à $[p_2,p_3,\ldots,p_n]$, p, s et q, et on renvoie le produit $\varepsilon_1\varepsilon_2$.

```
let rec epsilon L p s q = match L with
   [p1] -> if appartient {a=p;b=p1} q = 1 then
              0
           else
              intersecte {a=s;b=q} {a=p;b=p1}
   |p1::p2::L1 -> if appartient {a=p1;b=p2} q = 1 then
                     0
                  else
                     intersecte {a=s;b=q} {a=p1;b=p2}*(epsilon (p2::L1) p s q) ;;
```

La fonction `intérieur` se contente ensuite de calculer un point s situé en dehors du polygone et de calculer l'opposé du résultat de la fonction `epsilon`, puisque le point s est à l'extérieur du polygone.

```
let interieur P q = - epsilon P (hd P) (en_dehors P) q  ;;
```

Comme dit dans l'énoncé, cette procédure peut donner un résultat faux si le point q est tel que $[q,s]$ contient un des sommets de P.

B.5) Il faut donc modifier le choix de s pour que $p_i \notin [s,q]$ pour tout sommet p_i du polygone. Il y a peut-être une astuce que je ne vois pas … mais il est possible d'améliorer ainsi la fonction `en_dehors` :

notons (x_i, y_i) les coordonnées des points p_i et (x, y) celles de q. On commence par calculer $\alpha = 1 + \max(x_1, \ldots, x_n)$, comme dans la procédure `en_dehors`. On calcule ensuite un β tel que les droites $[p_i, q]$ ne passent pas par le point de coordonnées (α, β). Pour cela, on initialise β à une valeur quelconque (par exemple 0), puis on parcourt la liste P en augmentant β (si besoin est) pour que le point (α, β) soit toujours strictement au dessus de la droite $[p_i, q]$ (quand cette droite existe et quand elle n'est pas verticale). Il suffit pour cela d'avoir $\beta > y_i + \dfrac{(\alpha - x_i)(y_i - y)}{x_i - x} = \beta_i$. Cela s'écrit assez facilement, la fonction `beta` calculant récursivement la valeur de β en fonction de α, q et $[p_1, p_2, \ldots, p_n]$:

```
let rec beta alpha q = function
   []->0
   | p::L -> let N=beta alpha q L in
               if p.x=q.x then
                  N
               else
                  max N 1+(p.y+((alpha-p.x)*(p.y-q.y))/(p.x-q.x));;


let en_dehors_bis P q = let alpha=maximum(P)+1 in {x=alpha;y=beta alpha q P};;

let interieur_bis P q = -epsilon P (hd P) (en_dehors_bis P q) q ;;
```

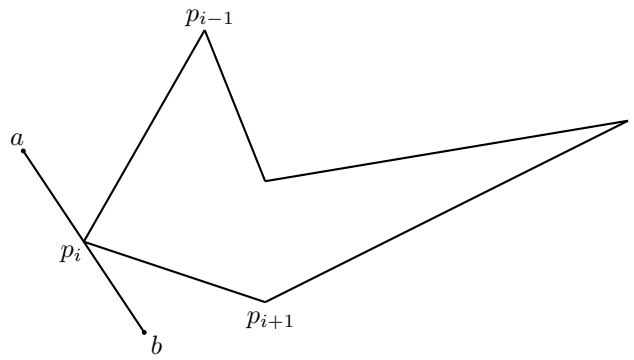
Cette méthode pose toutefois un problème : même si les x_i, y_i sont de taille raisonnable, les valeurs β_i peuvent être très grandes et dépasser `max_int`, le pire étant le cas où l'on calcule un β_i égal à `max_int` : en ajoutant 1, on obtient une valeur aberrante pour `1+(p.y+((alpha-p.x)*(p.y-q.y))/(p.x-q.x))`.

D'autre part, cette méthode a un temps de calcul linéaire par rapport à n, alors que les questions 7 et 8 sont censées améliorer ce temps de calcul pour arriver ... à un temps linéaire par rapport à n. Il reste une solution qui est peut-être celle attendue par le concepteur du sujet : on commence avec $\beta = 0$, puis on regarde si un des p_i est dans $[s, q]$ (en traitant à part le cas où q est un sommet de P). Si c'est le cas, on incrémente β d'une unité et on recommence à tester, et ainsi de suite jusqu'à ne plus avoir de problème. Le temps mis pour tester une valeur de β est de l'ordre de n dans le pire des cas (on parcourt toute la liste), et on devra incrémenter β au maximum n fois (pour chaque i, si on a $p_i \in [s, p]$ à un certain instant du calcul, on n'aura plus jamais ensuite de problème avec le point p_i puisque l'ordonnée de s augmente). Le temps de calcul est donc quadratique en n. Pour effectuer ce calcul de β, nous aurons besoin de la procédure récursive `probleme` qui, appliquée à s, q et $[p_1, \ldots, p_n]$ renvoie le booléen `false` si q est égal à l'un des points p_i ou si $[s, q]$ ne contient aucun des points p_i, et le booléen `true` sinon. En effet, si q est l'un des sommets du polygone, le point extérieur n'a aucune importance puisque la procédure `interieur` va détecter l'appartenance de q à P.

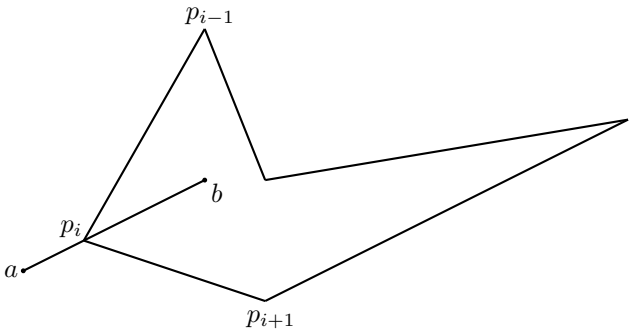
```
let rec probleme s q = function
   []-> false
   | p1::L -> if (p1.x=q.x & p1.y=q.y) then
                 false
              else if (appartient {a=s;b=q} p1=1) then
                 true
              else
                 probleme s q L;;


let en_dehors_ter P q = let alpha=maximum(P)+1 in let beta=ref(0) in
                        while probleme {x=alpha;y=(!beta)} q P do beta := !beta+1 done;
                        {x=alpha;y=(!beta)};;


let interieur_ter P q = -epsilon P (hd P) (en_dehors_ter P q) q ;;
```

B.6) Comme le polygone est sous forme simplifié, nous avons deux situations possibles :

- p_{i-1} et p_{i+1} sont du même côté de $[a, b]$ et a et b sont dans la même composante connexe :



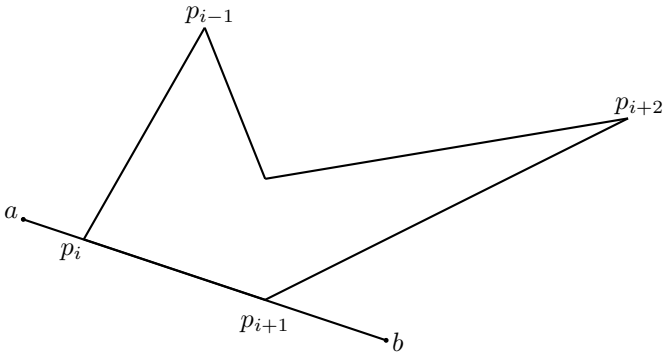
- p_{i-1} et p_{i+1} ne sont pas du même côté de $[a, b]$ et a et b ne sont pas dans la même composante connexe :



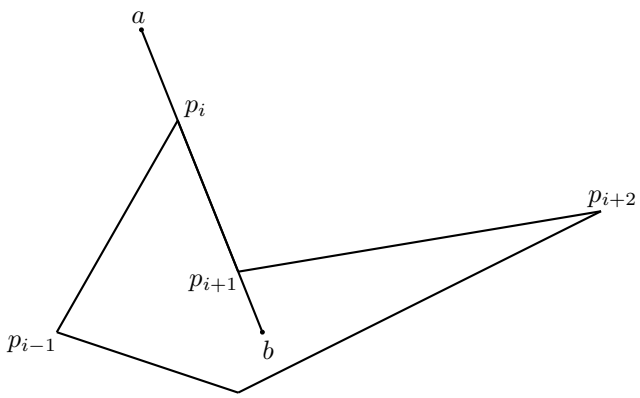
en remarquant que p_{i-1} et p_i ne sont pas sur la droite portée par $[a, b]$.

B.7) La situation est assez semblable au cas précédent :

- p_{i-1} et p_{i+2} sont du même côté de $[a, b]$ et a et b sont dans la même composante connexe :

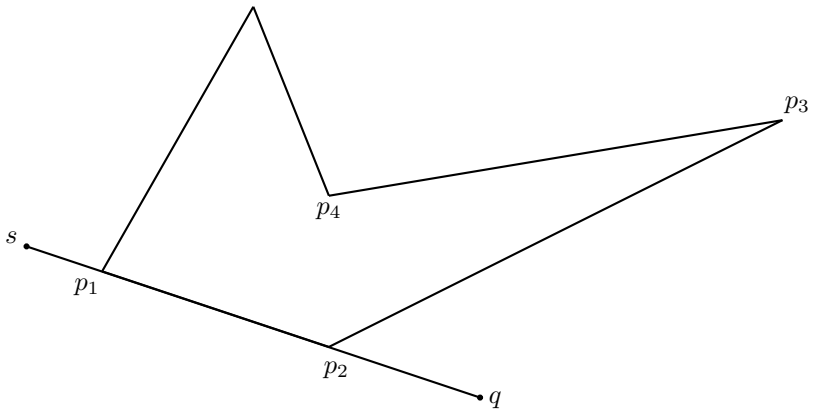


- p_{i-1} et p_{i+2} ne sont pas du même côté de $[a, b]$ et a et b ne sont pas dans la même composante connexe :



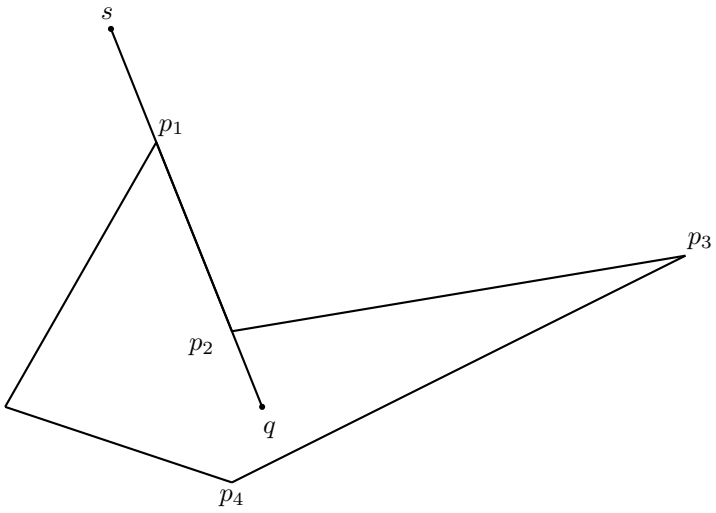
B.8) Les résultats précédents permettent d'écrire une fonction qui teste si un segment $[a, b]$ "traverse" une arête : si p_1, p_2, p_3 et p_4 sont quatre points consécutifs du polygone et si s et q sont deux points tel que s soit à l'extérieur du polygone, la fonction `traverse` appliquée à p_1, p_2, p_3, p_4, s et q renvoie 0 si $q \in [p_2, p_3]$, -1 si $[s, q]$ traverse $[p_2, p_3]$ en un point appartenant à $[p_2, p_3[$ [1] et 1 sinon. Le calcul de cette valeur ε se fait de la façon suivante :

- si $q \in [p_2, p_3]$, $\varepsilon = 0$;
- si $q \notin [p_2, p_3]$ et si $[s, q] \cap [p_2, p_3] = \emptyset$, $\varepsilon = 1$;
- si $q \notin [p_2, p_3]$, si $p_2 \in [s, q]$, si $p_3 \notin [s, q]$, on calcule $\varepsilon_1 =$ `meme_cote` $[s, q]$ p_1 p_3. Il y a encore deux cas à considérer :
  - si $\varepsilon_1 = 0$, il n'y a pas lieu de considérer que l'arête $[p_2, p_3]$ est traversée. En effet, ou bien $[s, q]$ ne traverse le polygone, comme dans le schéma :



[1] Quand $[p_2, p_3] \cap [s, q] = \{p_3\}$, on considère que le segment ne traverse pas $[p_2, p_3]$. Par contre, il traversera peut-être $[p_3, p_4]$.

ou bien il y a changement de côté au passage de p_2 :



mais ce changement est compté au passage de l'arête $[p_1, p_2]$. Dans les deux cas, il faut donc renvoyer $\varepsilon = 1$;

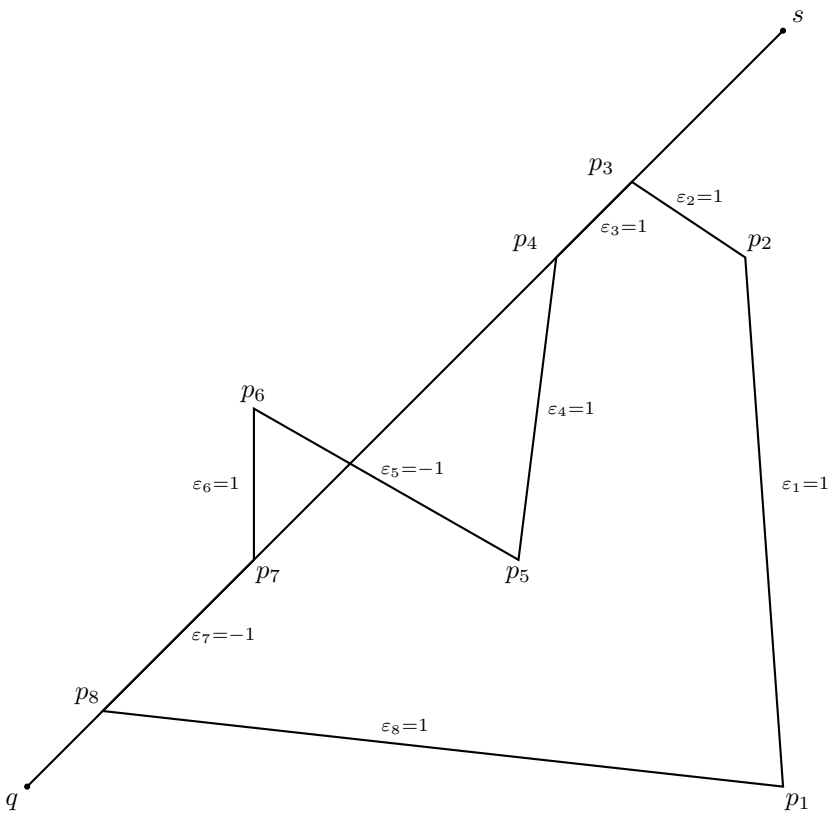
– sinon, $\varepsilon = \varepsilon_1$ car nous sommes dans le cas étudié à la question 6) ;

- si $q \notin [p_2, p_3]$, si $p_2 \in [s, q]$ et si $p_3 \in [s, q]$, on a $\varepsilon =$ `meme_cote` $[s, q]$ p_1 p_4 ;
- si $q \notin [p_2, p_3]$, si $p_2 \notin [s, q]$ et si $p_3 \in [s, q]$, on a $\varepsilon = 1$: on considère que l'arête $[p_2, p_3]$ n'est pas traversée (mais l'arête $[p_3, p_4]$ le sera peut-être) ;
- enfin, dans les autres cas, $[p_2, p_3]$ et $[s, q]$ se coupent ailleurs qu'en p_2 ou p_3 et $\varepsilon = -1$.

Nous pouvons donc écrire un peu laborieusement :

```
let traverse p1 p2 p3 p4 s q =
     if appartient {a=p2;b=p3} q =1 then
        0
     else
        if intersecte {a=p2;b=p3} {a=s;b=q} = (-1) then
           1
        else
           if appartient {a=s;b=q} p2 =1 then
              if appartient {a=s;b=q} p3=(-1) then let epsilon=meme_cote {a=s;b=q} p1 p3 in
                 if epsilon=0 then
                    1
                 else
                    epsilon
              else
                 meme_cote a=s;b=q p1 p4
           else
              if appartient a=s;b=q p3=1 then
                 1
              else
                 -1 ;;
```

Le schéma suivant donne un exemple de polygone et de segment $[s, q]$: chaque arête est étiquetée par la valeur prise par la fonction `traverse`.



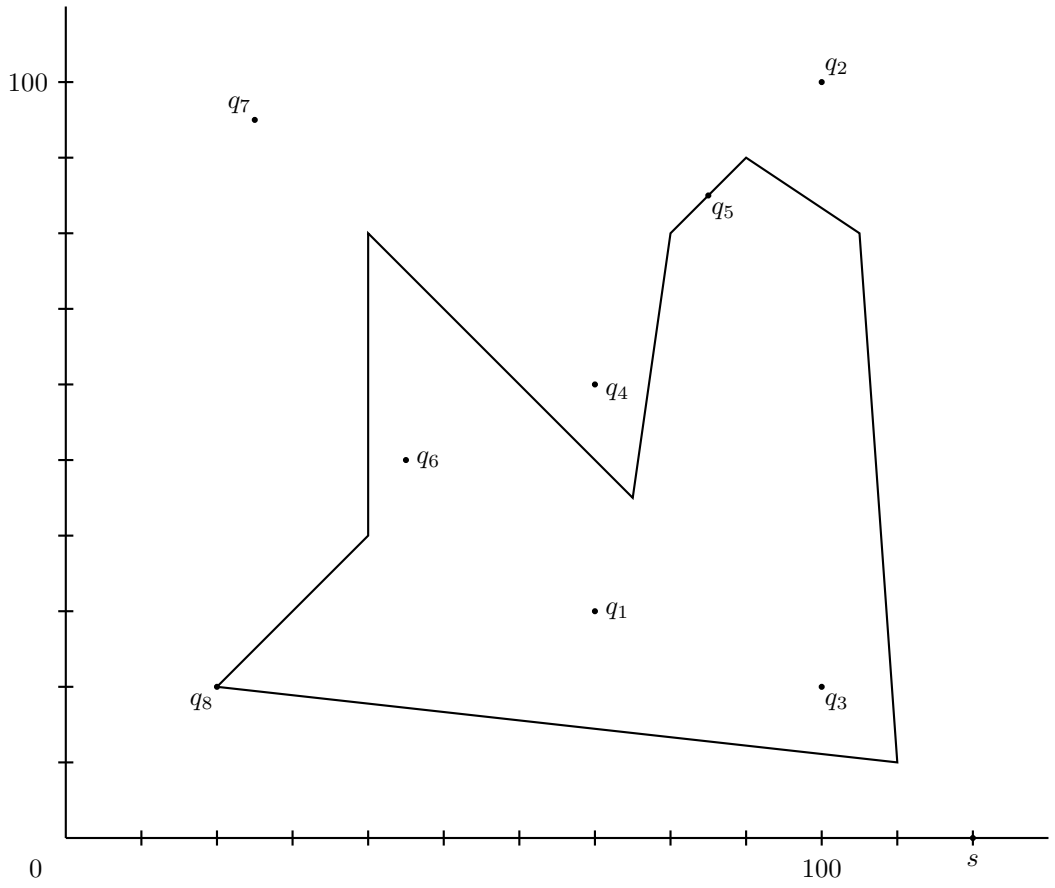
On a donc, pour tout point q :

- s et q sont dans la même composante connexe si et seulement si $\varepsilon_1\varepsilon_2 \dots \varepsilon_n = 1$;
- s et q sont dans des composantes connexes différentes si et seulement si $\varepsilon_1\varepsilon_2 \dots \varepsilon_n = -1$;
- q est sur le polygone si et seulement si $\varepsilon_1\varepsilon_2 \dots \varepsilon_n = 0$.

Ainsi, s étant à l'extérieur du polygone, la fonction `interieur` doit simplement calculer l'opposé du produit des ε_i. Cela se fait en parcourant tous les segments du polygone : comme on a besoin des derniers points pour traiter le premier segment $[p_1, p_2]$, on commence par le segment $[p_2, p_3]$, et on terminera par les segments $[p_{n-1}, p_n]$ et $[p_n, p_1]$. Pour travailler récursivement, nous devons conserver en mémoire les deux premiers points de la liste, car ces deux points servent à traiter les deux dernières arêtes. Cela donne :

```
let rec epsilon p1 p2 s q = function
   [q1;q2;q3] -> traverse q1 q2 q3 p1 s q *(traverse q2 q3 p1 p2 s q)
   | q1::q2::q3::q4::L -> traverse q1 q2 q3 q4 s q * (epsilon p1 p2 s q (q2::q3::q4::L));;

let interieur_quatro P q = - epsilon (hd P) (hd (tl P)) (en_dehors P) q P;;
```

On peut tester cette procédure sur l'exemple suivant :




```
# let P=[{x=110;y=10};{x=105;y=80};{x=90;y=90};{x=80;y=80};{x=75;y=45};{x=40;y=80};
                {x=40;y=40};{x=20;y=20}] ;;
P : point list =[{x=110;y=10};{x=105;y=80};{x=90;y=90};{x=80;y=80};{x=75;y=40};{x=40;y=60};
                {x=40;y=40};{x=20;y=20}]
# let q1={x=70;y=30};; let q2={x=100;y=100};; let q3={x=100;y=20};;
q1 : point={x=70;y=30}
q2 : point={x=100;y=100}
q3 : point={x=100;y=20}
# let q4={x=70;y=60};; let q5={x=85;y=85};; let q6={x=45;y=50};;
q4 : point={x=70;y=60}
q5 : point={x=85;y=85}
q6 : point={x=45;y=50}
# let q7={x=25;y=95};; let q8={x=20;y=20};;
q7 : point={x=25;y=95}
q8 : point={x=20;y=20}
# interieur P q1,interieur P q2,interieur P q3,interieur P q4 ;;
- : int*int*int*int = 1,-1,1,-1
# interieur P q5,interieur P q6,interieur P q7,interieur P q8 ;;
- : int*int*int*int = 0,1,-1,0
```

Partie II : Complexité de communication

A - Communication à sens unique

Remarque et définitions : cette partie est immédiate dès que l'on a compris qu'il suffisait de coder l'ensemble Y (ou l'ensemble des application f_x) à l'aide de mots, par exemple en utilisant un développement en base 2. Par contre, il y a de nombreuses erreurs d'énoncé, les $\lceil \log_2 n \rceil$ devant être (ou pouvant être) à peu près partout remplacés par des $\lceil \log_2(n+1) \rceil - 1$. Quand le résultat demandé est correct, j'ai donné une preuve du résultat demandé utilisant, pour un ensemble E de cardinal n, une application injective i_E de E dans l'ensemble des mots sur $\{0,1\}$ de longueur $k = \lceil \log_2 |E| \rceil$ (une telle application existe car il y a 2^k mots de longueurs k, et $n \leq 2^k$). J'ai alors noté s_E une réciproque à gauche de i_E, i.e. une application de $\{0,1\}^*$ dans E telle que $s_E \circ i_E = Id_E$. Par contre, quand le résultat demandé est faux, j'ai utilisé une injection σ_E de E dans l'ensemble des mots de longueurs au plus $k = \lceil \log_2(n+1) \rceil - 1$. Une telle injection existe , car il y a $1 + 2 + 2^2 + \cdots + 2^k = 2^{k+1} - 1$ mots de longueurs au plus k, et $n \leq 2^{k+1} - 1$. J'ai également noté δ_E une réciproque à gauche de σ_E.

A.1.a) Si z_0 est la valeur constante prise par f, les applications

$$\begin{array}{rccccccccc} g_0 : & X & \longrightarrow & \{0,1\}^* & \text{et} & g_1 : & \{0,1\}^* \times Y & \longrightarrow & Z \\ & x & \longmapsto & \varepsilon & & & (u,y) & \longmapsto & z_0 \end{array}$$

définissent un protocole calculant f, donc $D^1(f) = 0$.

A.1.b) Les applications $g_0 = i_X$ et $g_1 : (u,y) \mapsto f(s_X(u), y)$ définissent un protocole calculant f, de coût $\lceil \log_2 |X| \rceil$, et donc $D^1(f) \leq \lceil \log_2 |X| \rceil$.

Cette majoration n'est pas la "meilleure" : en utilisant σ_X et δ_X à la place de i_X et s_X, nous obtenons la majoration $D^1(f) \leq \lceil \log_2(|X|+1) \rceil - 1$.

A.1.c) Il manque évidemment un indice 2 dans l'énoncé. En notant $E = \{1, \ldots, p\}$ et, pour tout $k \in \mathbb{N}$, $k \bmod p$ l'unique élément de E congru à k modulo p, les applications :

$$\begin{array}{rccccccccc} g_0 : & X & \longrightarrow & \{0,1\}^* & \text{et} & g_1 : & \{0,1\}^* \times Y & \longrightarrow & Z \\ & k & \longmapsto & i_E(k \bmod p) & & & (u,y) & \longmapsto & f(s_E(u), y) \end{array}$$

définissent un protocole calculant f, de coût $\lceil \log_2 p \rceil$, et donc $D^1(f) \leq \lceil \log_2 p \rceil$.

Cette majoration n'est encore pas la "meilleure" : en utilisant σ_E et δ_E à la place de i_E et s_E, nous obtenons la majoration $D^1(f) \leq \lceil \log_2(p+1) \rceil - 1$.

A.1.d) Il y a un problème d'énoncé : comment définit-on $f(x,y)$ quand les deux parties x et y sont vides ? Pour répondre à la question avec la majoration demandée, nous pouvons restreindre X et Y aux parties non vides de $\{1, \ldots, n\}$ (il aurait été plus naturel de conserver X et Y et de poser $\sup \emptyset = 0$, mais la majoration demandée aurait été fausse). Posons cette fois $E = \{1, \ldots, n\}$; les applications

$$\begin{array}{rccccccccc} g_0 : & X & \longrightarrow & \{0,1\}^* & \text{et} & g_1 : & \{0,1\}^* \times Y & \longrightarrow & Z \\ & x & \longmapsto & i_E(\max x) & & & (u,y) & \longmapsto & \max\left(\{s_E(u)\} \cup y\right) \end{array}$$

définissent un protocole calculant f, de coût $\lceil \log_2 n \rceil$, et donc $D^1(f) \leq \lceil \log_2 n \rceil$.

Cette majoration est bien meilleure que celle du b), qui donne seulement $D^1(f) \leq \lceil \log_2(2^n - 1) \rceil = n$.

Remarque : une nouvelle fois, les applications

$$\begin{array}{rccccccccc} g_0 : & X & \longrightarrow & \{0,1\}^* & \text{et} & g_1 : & \{0,1\}^* \times Y & \longrightarrow & Z \\ & x & \longmapsto & \sigma_E(\max x) & & & (u,y) & \longmapsto & \max\left(\{\delta_E(u)\} \cup y\right) \end{array}$$

définissent un protocole calculant f, de coût $\lceil \log_2(n+1) \rceil - 1$. On en déduit que $D^1(f) \leq \lceil \log_2(n+1) \rceil - 1$, inégalité qui permettra de démontrer que la suite de l'énoncé est largement fausse. Il est inadmissible que

l'auteur se perde dans la manipulation de ces inégalités, alors que c'est la seule difficulté du problème (une fois que l'on a pensé à coder les éléments correctement).

A.1.e) Le résultat demandé est faux. Considérons en effet l'exemple du d) avec $n = 3$ (et en enlevant l'ensemble vide). Nous venons de démontrer que $D_1(f) \leq \lceil \log_2(n+1) \rceil - 1 = 1$; avec l'élément $y = \{1\}$, nous avons $\mathrm{Im}\,(f^y) = \{1,2,3\}$ et $\lceil \log_2(|\mathrm{Im}\,(f^y)|) \rceil = 2 > D^1(f)$! Par contre, $D^1(f) \geq \lceil \log_2(|\mathrm{Im}\,(f^y)|+1) \rceil - 1$.

Soit (g_0, g_1) un protocole optimal calculant f et soit $y \in Y$. En notant k le cardinal de l'image de f^y, il existe $x_1, \ldots, x_k \in X$ tels que les $f(x_i, y)$ soient deux à deux distincts. On en déduit que les $g_0(x_i)$ sont également deux à deux distincts, puisque $g_1(g_0(x_i), y) = f(x_i, y)$ pour tout i. Ainsi, il existe k mots distincts sur l'alphabet $\{0,1\}$, de longueur au plus $D^1(f)$: on en déduit que $k \leq 2^{D_1(f)+1} - 1$, soit :

$$\lceil \log_2(|\mathrm{Im}\,(f^y)|+1) \rceil - 1 \leq D_1(f).$$

A.1.f) Il est difficile de répondre à cette question, la définition donnée en d) étant incomplète, et la minoration donnée par l'énoncé en e) étant fausse ! La preuve demandée est sans doute la suivante : en posant $X = Y = \mathcal{P}(\{1, \ldots, n\} \setminus \{\emptyset\}$, nous avons montré que $D^1(f) \leq \lceil \log_2 n \rceil$. En choisissant $y = \{1\}$, nous avons ensuite $\mathrm{Im}\,(f^y) = \{1, 2, \ldots, n\}$ et le e) donnerait, s'il était correct, $D^1(f) \geq \lceil \log_2 n \rceil$, soit $D^1(f) = \lceil \log_2 n \rceil = 2 \ldots$ Malheureusement, la remarque précédente montre que $D^1(f) \leq \lceil \log_2(n+1) \rceil - 1 = 1$!

En corrigeant l'énoncé, on montrerait que, pour n quelconque, $D^1(f) \leq \lceil \log_2(n+1) \rceil - 1$.

A.2.a) Notons :

- E l'ensemble des lignes de la matrices M_f ;
- pour $x \in X$, L_x la x-ème ligne de M_f ;
- $\varphi : E \to X$ une application telle que pour tout élément ℓ de E, $L_{\varphi(\ell)} = \ell$.

Alors les applications :

$$g_0 : \begin{array}{ccc} X & \longrightarrow & \{0,1\}^* \\ k & \longmapsto & i_E(L_x) \end{array} \quad \text{et} \quad g_1 : \begin{array}{ccc} \{0,1\}^* \times Y & \longrightarrow & Z \\ (u,y) & \longmapsto & f(\varphi \circ s_E(u), y) \end{array}$$

définissent un protocole calculant f, de coût $\lceil \log_2 t \rceil$, et donc $D^1(f) \leq \lceil \log_2 t \rceil$.

Cette majoration n'est une nouvelle fois pas la bonne : en utilisant σ_E et δ_E à la place de i_E et s_E, nous obtenons la majoration $D^1(f) \leq \lceil \log_2(t+1) \rceil - 1$.

A.2.b) L'énoncé est une nouvelle fois faux : soit (g_0, g_1) un protocole calculant f, de coût c. En choisissant $x_1, \ldots, x_t$ tels que les lignes $L_{x_1}, \ldots, L_{x_t}$ soient deux à deux distinctes, les mots $g_0(x_1), \ldots, g_0(x_t)$ sont également deux à deux distincts. En effet, si, par exemple, nous avions $g_0(x_1) = g_0(x_2)$, alors pour tout $y \in Y$, nous aurions $f(x_1, y) = g_1(g_0(x_1), y) = g_1(g_0(x_2), y) = f(x_2, y)$ et L_{x_1} serait égale à L_{x_2}. Comme à la question 1.e), nous en déduisons que $t \leq 2^{c+1} - 1$, soit $c \geq \lceil \log_2(t+1) - 1 \rceil$.

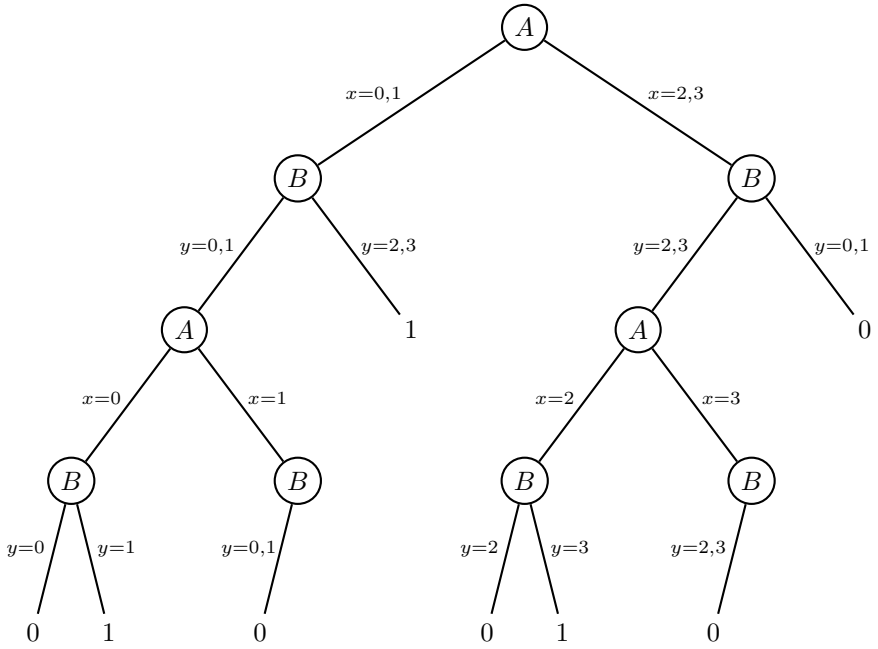
Nous avons donc démontré que $D^1(f) = \lceil \log_2(|\{f_x,\ x \in X\}| + 1) - 1 \rceil$.

A.2.c) Nous avons ici $X = Y = \{u \in \{0,1\}^*,\ |u| = n\}$ et $f : X \times Y \to \{0,1\}$ avec $f(u,v) = 1$ si $u = v$ et $f(u,v) = 0$ sinon. Il est clair que les 2^n applications partielles f_x sont distinctes : on en déduit que $D^1(f) = \lceil \log_2(2^n + 1) - 1 \rceil = n$. On n'est évidemment pas surpris qu'Alice soit obligée d'envoyer une information de longueur n pour que Bob puisse décider de l'égalité des deux mots x et y !

B - Communication avec aller-retour

Remarques : il faut lire Z au lieu de $\mathbb{Z}$ dans la définition d'un protocole. D'autre part, la définition de la "forme compacte" n'est pas claire : on parle de *concentration de chaque série de bits émis consécutivement par le même participant*, alors que dans les exemples donnés, un même participant ne renvoie jamais deux bits consécutivement ! Les formes compactes données sont en fait des traductions linéaires très peu pratiques de la structure arborescente.

B.1.a) Nous obtenons fastidieusement :



et les valeurs prises par f sont données par le tableau :

$x \backslash y$	0	1	2	3
0	0	1	1	1
1	0	0	1	1
2	0	0	0	1
3	0	0	0	0

Ainsi, $f(x,y) = 1$ si $x < y$ et $f(x,y) = 0$ sinon. Le coût du protocole est 4.

B.1.b) Pour $y \in Y$, notons $b_1 b_2 \dots b_k$ le mot $i_E(y)$, avec $k = \lceil \log_2 n \rceil$. Le protocole suivant calcule f :

1. Bob renvoie b_1 ;
2. Bob renvoie b_2 ;

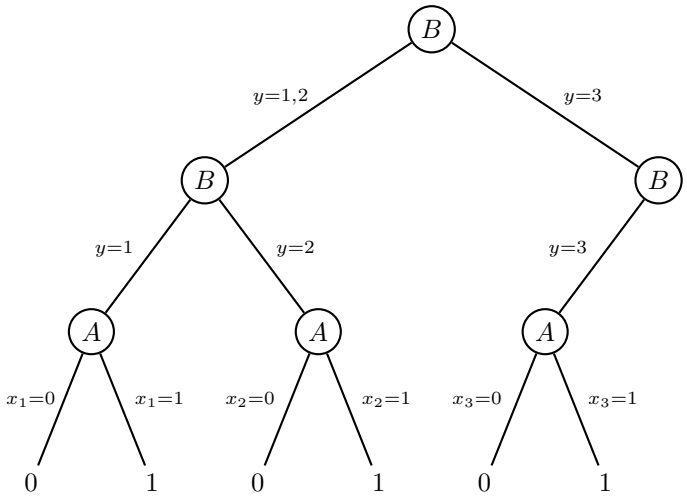
$\vdots$

k. Bob renvoie b_k ;

(k+1). Alice renvoie $a = x_{s_E(b_1 b_2 \dots b_k)}$;

(k+2). Le protocole termine et renvoie $f(x,y) = a$.

Ce protocole étant de coût $k+1$, on en déduit que $D(f) \leq \lceil \log_2 n \rceil + 1$. Par exemple, quand $n = 3$ et en choisissant $i_Y(1) = 00$, $i_Y(2) = 01$ et $i_Y(3) = 10$, le protocole est décrit par l'arbre :



B.1.c) On reprend la même méthode que précédemment : en posant toujours $k = \lceil \log_2 n \rceil$, on code x en un mot $a_1 a_2 \dots a_k$. Pendant les k premières étapes du protocole, Alice transmet les bits $a_1, \dots, a_k$, puis Bob, connaissant $a_1 a_2 \dots a_k$, calcule x et $i = y_x \in \{1, \dots, n\}$: il code alors i en un mot $b_1 b_2 \dots b_k$, qu'il transmet à Alice en k étape. À La $2k+1$-ème étape, Alice calcule i (elle connaît $b_1 b_2 \dots b_k$) et elle renvoie $c = u_i$. Le protocole termine à l'étape $2k+2$ et renvoie c. Nous obtenons ainsi un protocole qui calcule f avec un coût égal à $k+1$: $D(f) \leq 2\lceil \log_2 n \rceil + 1$.

B.2.a) Soit (g_0, g_1) un protocole à sens unique optimal calculant f. Considérons alors le protocole avec aller-retour :

- Dans un premier temps, Alice transmet $g_0(x)$ à Bob, en au plus $D^1(f)$ étapes ;
- Connaissant $g_0(x)$, Bob calcule $f(x,y) = g_1(g_0(x), y)$, qu'il code en un mot $b_1 b_2 \dots b_k$ de longueur $k = \lceil \log_2 |\mathrm{Im}\,(f)\,| \rceil$, par le biais de $i_{\mathrm{Im}(f)}$; il transmet ensuite ce mot en k étapes à Alice
- L'algorithme termine (Alice et Bob connaissent tous les deux $f(x,y)$) et renvoie $f(x,y) = s_{\mathrm{Im}(f)}(b_1 b_2 \dots b_k)$.

Le coût du protocole est égal à $D^1(f) + \lceil \log_2 |\mathrm{Im}\,(f)\,| \rceil$: $D(f)$ est donc au plus égal à $D^1(f) + \lceil \log_2 |\mathrm{Im}\,(f)\,| \rceil$.

B.2.b) Considérons un protocole optimal calculant f : son arbre est de hauteur $k = D(f)$. On en déduit qu'il a au plus 2^k feuilles. Comme les feuilles sont étiquetées par tous les éléments de $\mathrm{Im}\,(f)$, le nombre de feuille est au moins égal au cardinal de $\mathrm{Im}\,(f)$, et donc $|\mathrm{Im}\,(f)\,| \leq 2^k$, soit $\log_2 |\mathrm{Im}\,(f)\,| \leq D(f)$, i.e. $\lceil \log_2 |\mathrm{Im}\,(f)\,| \rceil \leq D(f)$.

Il existe un exemple stupide, qui n'est sans doute pas celui attendu par l'auteur : en choisissant f_0 constant, on a $D^1(f_0) = 0$ (question I.A.1.a) et donc :

$$D^1(f_0) + \lceil \log_2 |\mathrm{Im}\,(f_0)\,| \rceil = \lceil \log_2 |\mathrm{Im}\,(f_0)\,| \rceil \leq D(f_0) \leq D^1(f_0) + \lceil \log_2 |\mathrm{Im}\,(f_0)\,| \rceil.$$

B.2.c) Soit un protocole optimal avec aller-retour calculant f. La hauteur de l'arbre associé à ce protocole est donc égale à $D(f)$ et il possède k feuilles, avec $k \leq 2^{D(f)}$. L'idée qui me semble la plus naturelle consiste à numéroter les feuilles $(F_i)_{1 \leq i \leq k}$ et à poser $g_0(x) = u_1 u_2 \dots u_k$ avec :

$$u_i = \begin{cases} 1 & \text{s'il existe } y \in Y \text{ tel que le protocole avec aller-retour appliqué à } (x,y) \text{ aboutisse à la feuille } F_i \\ 0 & \text{sinon.} \end{cases}$$

Bob pourra alors, connaissant l'arbre et le mot u, calculer $f(x,y)$ en parcourant l'arbre depuis la racine, le choix du parcours s'effectuant de la manière suivante :

- arrivé en un nœud d'étiquette B, il choisit la bonne branche de l'arbre (car il connaît y) ;
- arrivé en un nœud N d'étiquette A, considérons l'ensemble I_g (resp. I_d) des i tels que F_i soit une feuille du fils gauche (resp. droit) de N ; on est alors dans l'un et un seul des deux cas suivants :
  - pour tout $i \in I_g$, $u_i = 0$ et Bob doit descendre dans le fils droit de N ;
  - pour tout $i \in I_d$, $u_i = 0$ et Bob doit descendre dans le fils gauche de N ;
- arrivé à une feuille, le calcul est terminé car $f(x, y)$ est la valeur stockée en cette feuille.

Cette construction définit (un peu informellement) la fonction g_1, et le protocole (g_0, g_1) calcule f avec un coût égal à k. Dans le cas où $k = 2^{D(f)}$, il faut améliorer un peu cette construction pour diminuer le coût d'une unité. Nous allons pour cela distinguer les trois cas possibles :

- Premier cas : il existe un nœud N d'étiquette A, de profondeur $D(f) - p$ avec p compris entre 2 et $D(f)$. On peut alors numéroter les feuilles de sorte que :
  - $F_1, F_2, \ldots, F_{2^{p-1}}$ soient les feuilles du fils gauche de N ;
  - $F_{2^{p-1}+1}, F_{2^{p-1}+2}, \ldots, F_{2^p}$ soient les feuilles du fils droit de N ;
  - $F_{2^p+1}, F_{2^p+2}, \ldots, F_{2^{D(f)}}$ soient les feuilles restantes.

  Si E_g et E_d désignent les étiquettes des arêtes gauche et droite de N, on définit $v = v_0 v_1 \ldots v_{2^{D(f)}-2^{p-1}}$ de la façon suivante :
  - si $y \in E_g$, on pose $v_0 = 0$ et $v_1 \ldots v_{2^{D(f)}-2^{p-1}} = u_1 \ldots u_{2^{p-1}} u_{2^p+1} \ldots u_{2^{D(f)}}$: on supprime donc du mot u les lettres nulles qui correspondent aux feuilles du fils droit de N ;
  - sinon, on pose $v_0 = 1$ et $v_1 \ldots v_{2^{D(f)}-2^{p-1}} = u_{2^{p-1}+1} \ldots u_{2^{D(f)}}$: on supprime du mot u les lettres nulles qui correspondent aux feuilles du fils gauche de N.

  On pose alors $g_0(x) = v$; la connaissance de v permet à Bob de reconstituer le mot u, puis de calculer $f(x, y)$: nous obtenons un protocole de coût $1 + 2^{D(f)} - 2^{p-1} \leq 2^{D(f)} - 1$ (car $p \geq 2$).
- Deuxième cas : tous les nœuds de profondeur k, avec $0 \leq k \leq D(f) - 2$, sont d'étiquette B et il existe un nœud N d'étiquette B et de profondeur $D(f) - 1$. En numérotant correctement les feuilles, on peut supposer que les filles de N sont les feuilles F_1 et F_2. On pose alors $g_0(x) = u_2 u_3 \ldots u_{2^{D(f)}}$; la connaissance de v permet encore à Bob de reconstituer le mot u, puisque $u_1 = u_2$, puis de calculer $f(x, y)$: nous obtenons un protocole de coût $2^{D(f)} - 1$.
- Troisième cas : tous les nœuds de profondeur k, avec $0 \leq k \leq D(f) - 2$, sont d'étiquette B et tous les nœuds de profondeur $D(f) - 1$ sont d'étiquette A. En numérotant les feuilles de gauche à droite, on pose $g_0(x) = u_1 u_3 u_5 \ldots u_{2^{D(f)}-1}$. Bob pourra calculer $f(x, y)$ en descendant dans l'arbre jusqu'à un nœud de profondeur $D(f)$: si F_{2i-1} et F_{2i} sont les filles de ce nœud, comme Bob connaît u_{2i-1}, il saura s'il doit descendre à gauche ou à droite pour obtenir $f(x, y)$. Nous obtenons cette fois un protocole de coût $2^{D(f)-1} \leq 2^{D(f)-1}$.

Dans tous les cas, il existe un protocole (g_0, g_1) de coût inférieur à $2^{D(f)} - 1$, donc $D_1(f) \leq 2^{D(f)} - 1$.

Notons $k = \lfloor \log_2 D^1(f) \rfloor$. Nous avons alors $2^k \leq D^1(f) \leq 2^{D(f)} - 1$, donc $2^k + 1 \leq 2^{D(f)}$, puis $k < D(f)$. Ces nombres étant entiers, $\lfloor \log_2 D^1(f) \rfloor + 1 = k + 1 \leq D(f)$.

B.2.d) Reprenons l'exemple de la question 1b). Pour x, x' éléments distincts de X, il existe $y \in Y$ tel que $x_y \neq x'_y$, et donc $f_x \neq f_{x'}$: on en déduit que $|\{f_x,\ x \in X\}| = |X| = 2^n$. D'après II.A.2) :

$$D^1(f) = \lceil \log_2(|\{f_x,\ x \in X\}| + 1) \rceil - 1 = n$$

et nous obtenons :

$$\lfloor \log_2 D^1(f) \rfloor + 1 \leq D(f) \leq \lceil \log_2 n \rceil + 1 = \lceil \log_2 D^1(f) \rceil + 1.$$

L'égalité demandée est donc vérifiée dès que $\lfloor \log_2 n \rfloor = \lceil \log_2 n \rceil$, i.e. dès que n est une puissance de 2.