Corrigé DS2, sujet X

Question 1.

```
let melange_knuth a =
   let n=Array.length a in
   for i=0 to n-1 do
      let j=Random.int (i+1) in
      let x=a.(j) in a.(j) <- a.(i);
      a.(i) <-x
   done;
```

Quesiton 2.
Montrons que, (1) pour tout $i \in [\![0,n]\!]$, pour tout $(p,q) \in [\![0,i-1]\!]^2$, $Pr(x_p = q) = \frac{1}{i}$ et (2) pour tout $p \in [\![i,n-1]\!]$, $x_p = p$.
Pour $i = 0$, (1) est triviale car $[\![0,i-1]\!] = \emptyset$. De plus, par hypothèse, pour tout $p \in [\![0,n-1]\!]$, $x_p = p$.
Soit $i \in [\![0,n-1]\!]$ tel que (1) et (2) sont vérifiés. Notons x_p, pour tout p, la valeur de la p-ème case avant l'étape $i+1$ et x'_p celle après cette étape.
On choisit $j \in [\![0,i]\!]$ aléatoirement selon la probabilité uniforme et on échange la valeur de x_i, à savoir i, avec celle de x_j. Les événements $x'_i = x_j$, pour $0 \leq j \leq i-1$, forment un système complet d'événements équiprobables ; en outre, $\{x_j \,/\, j \in [\![0,i]\!]\} = [\![0,i]\!]$ d'après la propriété 2. Les événements $x'_i = j$ forment donc, pour $0 \leq j \leq i$, un système complet d'événements équiprobables ; on a donc, $\forall j \in [\![0,i]\!]$, $Pr(x'_i = j) = \frac{1}{i+1}$.
En outre, pour tout $j \in [\![0,i-1]\!]$, pour tout $k \in [\![0,i-1]\!]$, $Pr(x'_j = k) = Pr(x'_j = k, x_j = k)$; l'événement $(x'_j = k, x_j = k)$ est l'événement $(x_j = k)$ et $x'_i \neq k$. Ces deux événements étant indépendants, $P(x'_j = k) = Pr(x_j = k)Pr(x'_i \neq k) = \frac{1}{i} \times (1 - Pr(x'_i = k)) = \frac{1}{i} \times (1 - \frac{1}{i+1}) = \frac{1}{i} \times \frac{i}{i+1} = \frac{1}{i+1}$.
Enfin, pour tout $j \in [\![0,i-1]\!]$, $Pr(x'_j = i) = \sum_{k=1}^{i-1} Pr(x'_j = i, x_j = k) = \sum_{k=1}^{i-1} Pr(x_j = k, x'_i = k) = \sum_{k=1}^{i-1} Pr(x_j = k) \times Pr(x'_i = k) = \sum_{k=1}^{i-1} \frac{1}{i} \times \frac{1}{i+1} = i \times \frac{1}{i} \times \frac{1}{i+1} = \frac{1}{i+1}$.
Ceci prouve (1). Enfin, les valeurs x_p, pour $i+1 \leq p < n$ sont inchangées donc $x_p = p$ pour tout $p \in [\![i+1,n-1]\!]$.
Par récurrence, (1) et (2) sont vérifiés pour tout $i \in [\![0,n]\!]$. En particulier, pour $i = n$, pour tout $(p,q) \in [\![0,n-1]\!]$, $Pr(x_p = q) = \frac{1}{n}$.
Question 3.

```
let labyrinthe1 g = let traites = Array.make g.n false and labyrinthe= graphe_vide g.n in
   let rec dfs v t_adj i =
      if i<Array.length t_adj then let w=t_adj.(i) in
         if traites.(w) then dfs v t_adj (i+1)
         else begin
            ajoute_arete labyrinthe v w;
            traites.(w) <- true;
            let t_adj2=Array.of_list g.adj.(w) in melange_knuth t_adj2;
            dfs w t_adj2 0;
            dfs v t_adj (i+1) end in
   let i=Random.int g.n in traites.(i) <- true;
   let t_adj=g.adj.(i) in melange_knuth g.adj.(i);
   dfs i t_adj 0;
   labyrinthe;;
```

Question 4.
Montrons qu'à chaque étape de l'algorithme, étant donnés deux sommets i et j, il existe un chemin par le labyrinthe entre i et j si et seulement si les deux sont déjà traités (à savoir `traites.(i)` et `traites.(j)` sont `true`) et, dans ce cas, ce chemin est unique.
Initialement, il n'y a qu'un seul sommet i pour lequel traites(i) est vrai. La propriété est donc vraie.
Supposons qu'elle le soit à une étape. On effectue dfs v l. Si l'élément de tête de l est traité, rien ne se passe donc la propriété reste vraie. Si l'élément de tête w de la liste l n'est pas traité, on effectue traité(w) :=Vrai. La seule arête entre w et un sommet traité est alors (w,v). Soit un chemin dans le labyrinthe entre deux éléments s et t traités. Si ce chemin passe par w, il ne peut qu'emprunter l'arête (v,w) et ce une seule fois. Ainsi, quitte à échanger s et t, on peut supposer que w=t ; le chemin (s,...,t) est donc (s,...v,w). Par propriété de récurrence, le chemin (s,...,v) est unique. Le chemin entre s et t est donc unique. Si ni s ni t n'est w, ce chemin ne peut pas passer par w donc, par propriété de récurrence, il est unique.
Ceci prouve que, pour tout couple de sommets du labyrinthe, il ne peut exister qu'au plus un chemin entre eux. Comme le graphe est connexe, tout sommet sera connecté au labyrinthe à la fin de l'algorithme. Ainsi, tout couple de sommets sont reliés par un unique chemin.
Question 5.

```
let rec cd_trouve c i = if c.lien.(i)=i then i else cd_trouve c c.lien.(i);;
```

Question 6.

```
let cd_union c i j = let ri=cd_trouve (c.lien) i and rj=cd_trouve (c.lien) j in
   if ri<>rj then
      if c.rang.(ri)<c.rang.(rj) then (c.lien.(ri) <- rj; c.rang.(rj) <- max c.rang.(rj) (c.rang(ri)+1) )
      else (c.lien(rj) <- ri; c.rang(ri) <- max c.rang(ri) (c.rang(rj)+1) );;
```

Questiojn 7.
Supposons que cette fonction fusionne deux classes qui vérifie les deux propriétés. Supposons que la première classe est de rang k, la

deuxième de rang k'. Quitte à échanger k et k', supposons que $k \leq k'$ et que le représentant sera k'. Alors le représentant de la fusion de ces deux classes sera k' et aura rang $\max(k+1, k')$. Soit n le nombre d'éléments de la première classe, n' celui de la deuxième classe. Alors le nombre d'éléments de la classe fusionnée est $n + n' \geq 2^k + 2^{k'} \geq 2^{k'}$ et $n + n' \geq 2^k + 2^{k'} \geq 2^k + 2^k = 2 \times 2^k = 2^{1+k}$ donc $n + n' \geq \max(2^{k'}, 2^{k+1}) = 2^{\max(k', k+1)}$. En notant k'' le rang de la classe fusionnée, n'' son nombre d'éléments, $n'' \geq 2^{k''}$.
Enfin, notons r et r' les représentants des deux classes fusionnées. Le plus long chemin jusqu'au représentant r' de la classe fusionnée est le plus long chemin jusqu'à r dans la première classe auquel on ajoute l'arête (r, r') ou le plus long chemin jusqu'à r' dans la deuxième classe ; ce premier chemin est de longueur $k+1$, le deuxième k'. Ainsi, le plus long chemin dans la classe fusionnée est de longueur $\max(k', k+1) = k''$.
Ceci prouve les deux propriétés demandées.

Question 8.

Lors de la fusion de deux classes, on utilise la fonction "cd-trouve" pour chacune des deux classes fusionnées. La complexité de cette fonction pour une classe de rang k est en $O(k)$ d'après la propriété 2 de la question 7. En outre, le nombre d'éléments p de cette classe est au moins 2^k. On a donc $k \leq \log_2(p)$ donc $k \leq \log_2(n)$. La complexité de "cd-trouve" est donc $O(\log(n))$. Les autres opérations de la fonction "cd-union" sont en temps constant. En conclusion, la complexité de "cd-union" est $O(\log(n))$.

Quesiton 9.

```
let labyrinthe2 g = let c=cd_init g.n and a=aretes g and h=graphe_vide g.n in melange_knuth a;
   for k in 0 to Array.length a -1 do
      let (v,w)=a.(k) in let rv=cd_trouve c.lien v and rw=cd_trouve c.lien w in
      if rv<>rw then (
         ajoute_arete h v w;
         if c.rang(ri)<c.rang(rj) then (c.lien(ri) <- rj; c.rang(rj) <- max c.rang(rj) (c.rang(ri)+1))
         else (c.lien(rj) <- ri; c.rang.(ri) <- max c.rang(ri) (c.rang(rj)+1)) )
   done;
   h;;
```

J'ai préféré ne pas utiliser la fonction "cd-union" afin de ne pas calculer deux fois "cd-trouve" sur un même argument.

Question 10.

Supposons que l'invariant est vrai avant d'appliquer l'algorithme à l'arête (v, w).
Si v et w sont dans la même classe, il ne se passe rien.
S'ils ne sont pas dans la même classe, notons C_v et C_w ces deux classes ; considérons s et t deux sommets de la classe fusionnée $C_v \cup C_w$.
Si s et t sont dans C_v, il existe, par hypothèse de récurrence, un chemin entre s et t dans C_v donc il existe un chemin dans $C_v \cup C_w$ entre s et t. De plus, un tel chemin ne peut emprunter qu'une fois l'arête (v, w). En outre, cette arête est la seule arête entre un sommet de C_v et un sommet de C_w. Ainsi, un chemin ne pouvant emprunter deux fois cette arête, un chemin entre s et t dans $C_v \cup C_w$ ne peut passer par cette arête ni par C_w. Ainsi, ce chemin ne passe que par C_v, il est donc, par hypothèse de récurrence, unique. De même, si s et t sont deux sommets de C_w, il existe un unique chemin entre s et t. Enfin, si s est un sommet de C_v et t un sommet de C_w, un chemin entre s et t doit passer par (v, w) ; il doit donc commencer par un chemin entre s et v dans C_v (qui est unique), se poursuivre par (v, w) et se terminer par un chemin de w à t dans C_w (qui est unique). Ce chemin est donc unique (et existe bien).
Ceci prouve qu'il existe un unique chemin entre s et t dans le labyrinthe pour tout (s, t) d'une même classe de ce labyrinthe.
Par conséquent, à la fin de l'algorithme, chaque composante connexe du labyrinthe est parfait. Supposons qu'il y ait deux composantes connexes. Le graphe initial étant connexe, il existe une arête entre deux sommets de ces deux composantes ; notons-la (v, w) ; lorsque l'algorithme traite (v, w), ces deux sommets n'étant pas dans la même composante, leurs composantes auraient dû être fusionnées et cette arête ajoutée, ce qui contredit qu'ils ne sont pas dans la même composante à la fin de l'algorithme. Par conséquent, le labyrinthe obtenue est connexe et sa seule composante, à savoir lui-même, est parfaite. Le labyrinthe obtenu est donc parfait.

Question 11.

Montrons par récurrence que, pour chaque ligne i, pour chaque couple v, w appartenant à des lignes précédant la ligne i, il existe au plus un chemin reliant v à w.
Cette propriété est vraie pour $i = 0$.
Supposons la vraie jusqu'à une ligne i avant le traitement de la ligne $i+1$.
Montrons que pour le traitement de chaque arête de la ligne, il y a au plus un chemin entre deux sommets d'une ligne de numéro au plus $i+1$.
C'est vrai par hypothèse de récurrence avant le traitement de la première arête. Supposons que cela soit vrai jusqu'au traitement de l'arête $(v, v+1)$. Si v et $v+1$ sont déjà reliés par un chemin, le labyrinthe reste identique ; sinon, supposons que l'ajout de l'arête $(v, v+1)$ crée un deuxième chemin entre deux sommets u et w. Il y a donc, dans ce cas, un nouveau chemin qui se décompose en un chemin de u à v, un chemin de v à $v+1$ et un chemin de $v+1$ à w ; or, il y a aussi un ancien chemin de u à w ; par conséquent, il y avait déjà un chemin de v à u, un chemin de u à w et un chemin de w à $v+1$; ceci signifie que v et $v+1$ avait déjà le même représentant donc l'arête ne pouvait être ajoutée selon l'algorithme.
Ceci prouve bien que, à la fin de l'étape (a), entre chaque couple de sommets (v, w) d'une ligne au plus $i+1$, il y a au plus un seul chemin.
L'étape (b) n'ajoute que des arêtes dont l'un des sommets n'est relié à aucun autre donc ne peut créer un deuxième chemin.
Ceci prouve, par récurrence, qu'à la fin de l'algorithme, il ne peut y avoir plus d'un chemin entre deux sommets du graphe.
Montrons désormais, par récurrence, que pour chaque ligne i, pour chaque sommet v d'une ligne au plus $i-1$, il existe au moins un sommet de la ligne i relié à v.
Pour $i = 1$, c'est vrai. Supposons le jusqu'à une ligne i. Par hypothèse de récurrence, tout sommet d'une ligne au plus i est relié à un sommet de la ligne i ; or tout sommet de la ligne i est dans l'une des classes des sommets de la ligne i et un élément de chacune de ces classes est relié à un élément de la ligne $i+1$ par l'opération (b) ; par conséquent, tout élément d'une ligne au plus i est relié à un élément de la ligne i qui est relié à un représentant de sa classe qui sera relié à la ligne $i+1$; tout élément d'une ligne au plus i est donc relié à la ligne $i+1$.
Ainsi, par récurrence, chaque sommet est relié à un élément de la dernière ligne. Enfin, grâce à l'opération 2, tous les sommets de la dernière ligne sont dans la même classe d'équivalence. Ainsi, pour tout (v, w) du graphe, il existe v_n et w_n sommets de la dernière ligne tels que v_n et v sont dans la même classe et w_n et w sont dans la même classe. Comme v_n et w_n sont sur la dernière ligne, ils sont dans la même classe donc v et w sont dans la même classe. Ceci prouve qu'il existe un chemin entre v et w, pour tout (v, w).

On a bien prouvé qu'il existe un unique chemin entre v et w pour tout couple (v, w) de sommets. Le labyrinthe fabriqué est donc parfait.

Question 12.

Considérons un labyrinthe connexe L. Supposons avoir construit via l'algorithme d'Eller les i première lignes, $i < n-1$, de ce labyrinthe à la fin de la première étape de la ligne i.

Lors de l'étape 1.b. de l'algorithme d'Eller, on peut ajouter une arête entre tout sommet de la ligne i et son voisin de la ligne $i+1$. Ainsi, il est possible (avec un peu de chance) d'ajouter toute arête entre les lignes i et $i+1$ du labyrinthe L. Supposons (ou prions pour) que cela soit fait.

Passons à la première étape (étape 1.a) de la ligne $i+1$: s'il y a une arête entre v et $v+1$ dans le labyrinthe L, par unicité du chemin entre v et $v+1$, il n'y a pas d'autre chemin, donc sans cette arête, ces sommets ne sont pas reliés. Ainsi, dans le labyrinthe en cours de construction, v et $v+1$ n'étant pas reliés, l'algorithme peut (avec probabilité $1/2$) ajouter cette arête. Il est donc possible (si la chance nous sourit) d'ajouter les arêtes du labyrinthe L à la ligne $i+1$. Il est donc possible (si dieu le veut) d'obtenir un graphe ayant les $i+1$ premières lignes du labyrinthe L.

Supposons avoir construit les $n-1$ premières lignes du labyrinthe L (jusque-là nous sommes très chanceux). Si v et $v+1$ sont sur la dernière ligne et sont reliés dans L, sans cette arête, ils ne sont pas reliés dans L. Dans le labyrinthe en construction, ils ne sont donc pas reliés, donc, l'algorithme d'Eller va automatiquement les relier lors de l'étape 2. S'il n'y a pas d'arête entre v et $v+1$, ils sont reliés par un chemin qui ne contient pas d'arête de la dernière ligne, donc ils étaient déjà reliés dans l'algorithme en construction.

On obtient donc la dernière ligne du labyrinthe L donc - dieu soit loué ! - le labyrinthe L.

Question 13.

```
let ajoute_debut f e = if f.debut>0 then f.debut <- f.debut-1 else f.debut <- Array.length (f.contenu)-1;
   f.taille <- f.taille+1;
   f.contenu.(f.debut) <- e;;
```

Question 14.

```
let retire_debut f = f.taille <- f.taille -1;
   let n=Array.length(f.contenu) in
   if f.debut = n-1 then (f.debut <- 0; f.contenu.(n-1))
   else (f.debut <- f.debut+1; f.contenu(f.debut-1));;
```

Question 15.

La distance de la source s à elle-même est 0, ce qu'effectue l'algorithme. C'est aussi le seul sommet à distance 0 de s.

Pour chaque sommet w différent de la source s, sa distance à la source est $1 + d(s, v)$ où v est le voisin de w le plus proche de s.

Comme une structure de file est utilisée, les sommets du graphe sont ajoutés à la liste par ordre de distance croissante à la source s. Ainsi, lorsque w est découvert pour la première fois, il est découvert depuis son voisin le plus proche de s. Si chaque sommet à distance k de s se voit attribuer la distance k par l'algorithme, alors tout sommet v à distance $k+1$ ayant pour voisin le plus proche de s un sommet à distance k de s, ce sommet v se verra attribué la distance $k+1$ par l'algorithme.

Ceci prouve, par récurrence sur la distance d'un sommet à la source, comme tout sommet est à distance fini de la source s, que l'algorithme attribuera bien à chaque sommet v à distance inférieure à la distance entre source s et destination d la distance entre la source et ce sommet v. En particulier, c'est vrai pour la destination d. L'algorithme renvoie donc bien la distance entre la source s et la destination d.

Question 16.

```
let minimum_monstres g monstre src dst =
   let distance = Array.make g.n (-1,-1) in
   let f = file_vide g.n in
   ajoute_debut src;
   distance.(src) <- if monstre.(src) then (1,0) else (0,0);
   let rec loop () =
      let v=retire_debut f in
      if v = dst then distance.(v) else begin
         List.iter(fun w -> if distance.(w)=(-1,-1) then begin
            let (m,l)=distance.(v) in
            if monstre.(w) then (distance.(w) <- (m+1,l+1); ajoute_fin f w)
            else (distance.(w) <- (m,l+1); ajoute_debut f w)
            end
         ) g.adj.(v);
         loop ()
      end in
   loop ();;
```

Question 17.

La deuxième figure de la page 13 avec source 0, destination 7, va trouver comme chemin $(0, 4, 8, 12, 13, 14, 15, 11, 7)$ qui passe par un monstre alors que le chemin $(0, 1, 2, 3, 7)$ ne passe que par un monstre et est plus court.

Question 18.

étape	`distance`	`f`	`sources-courantes`	`sources-suivantes`
1	$0 \leftarrow (0,0)$	0	$\emptyset$	$\emptyset$
2be	$1 \leftarrow (0,1)$	1		4
	$4 \leftarrow (1,1)$			
2be	$2 \leftarrow (0,2)$	2		4,5
	$5 \leftarrow (1,2)$			
2be	$3 \leftarrow (1,3)$	$\emptyset$		4,5,3,6
	$6 \leftarrow (1,3)$			
2a		4	5,3,6	$\emptyset$
2bde	$8 \leftarrow (1,2)$	5,8	3,6	
2bde	$9 \leftarrow (2,3)$	8,3	6	9
2bde	$12 \leftarrow (1,3)$	3,6,12		9
2be	$7 \leftarrow (1,4)$	6,12,7		9
2be	$10 \leftarrow (2,4)$	12,7		9,10
2be	$13 \leftarrow (1,4)$	7,13		9,10
2be	$11 \leftarrow (1,5)$	13,11		9,10
2be	$14 \leftarrow (1,5)$	11,14		9,10
2bc				

La réponse est $(1,5)$.

Question 19.

On montre que, lors de la sortie d'un sommet `v` de la file `f` qui a valeur (m,l) dans le tableau distance,

-tous les éléments des files `f`, `sources-courantes` et `sources-suivantes` ont comme valeur dans `distance` la valeur (m,l) constitué par "minimum monstres", "minimum longueur" comme définie précédemment par l'énoncé

-tous les éléments de `f` et `sources-courantes` ont même nombre de monstres que celui de `v`, à savoir m

-tous les éléments de `f` ont longueur l ou $l+1$, l étant la longueur de `v`, et ils sont triés par ordre croissant de longueur.

-tous les éléments de `sources-courantes` ont un élément "longueur" valant au moins $l+2$ et sont triés dans l'ordre croissant pour cet élément

-tous les éléments de `sources-suivantes` ont valeur "minimum monstres" égale à $m+1$ et sont triés de façon croissante selon la valeur "longueur".

Ceci découle de la définition de l'algorithme. On vérifie que chaque propriété se propage d'une étape à l'autre de l'algorithme et qu'elles sont vraies initialement.

Ainsi, lorsqu'on arrive à `dst`, la valeur contenue dans `distance.(dst)` est bien le couple (m,l) cherché.

Question 20.

On associe à une arête avec monstre un poids $n+1$ où n est le nombre de sommets et à une arête sans monstre le poids 1.

Le poids d'un chemin est donc $l+mn$ où l est la longueur du chemin et m le nombre de monstres. Considérons un chemin c de longueur l, rencontrant m monstres et un chemin c' de longueur l' rencontrant m' monstres.

Comme le graphe est de taille n, la taille maximale d'un chemin est $n-1$ donc $l<n$ et $l'<n$ donc si $m'>m$, $n(m'-m) \geq n > l-l'$ donc $nm'+l' > nm+l$ donc le chemin de poids minimal est c, celui qui a le plus petit nombre de monstres (indépendamment de sa longueur). Par symétrie, il en est de même si $m'<m$. Si $m=m'$, le chemin de poids minimal est c si $l<l'$, c' sinon.

On a donc bien c de poids inférieur à c' si et seulement si $m<m'$ ou ($m=m'$ et $l \leq l'$).