1 Déplacement d'un robot dans une grille

Question 1 Un exercice classique de programmation. La référence `i1` garde en mémoire un indice d'une valeur plus petite et `i2` d'une valeur **strictement** plus grande que `a`.

```
let dichotomie a t =
   let imin = ref 0 and imax = ref (Array.length t) in
   while !imax - !imin > 1 do
      let i = (!imin + !imax) / 2 in
      if a < t.(i) then
         imax := i
      else
         imin := i
   done;
   !imin;;
```

La complexité est bien logarithmique, car la valeur $i_2 - i_1$ diminue de moitié (à partie entière près) à chaque itération.
Il faut faire à ne pas faire des copies intempestives de tableaux. La complexité serait dégradée.
On peut aussi avoir recours à une fonction récursive à la place de la boucle `while`.

```
let dichotomie a t =
   let rec parcours imin imax =
      if imax - imin <= 1 then imin
      else
         (let i = (imin + imax) / 2 in
           if a < t.(i) then parcours imin i
           else parcours i imax)
   in parcours 0 (Array.lenght t);;
```

Question 2 Pour simplifier l'écriture de cette fonction, on renommera en `ob_li` et `ob_co` les deux vecteurs d'obstacles. On commence par chercher l'indice de l'obstacle juste avant la position (en largeur et hauteur) et on reconstruit les positions atteignables :

```
let deplacements_grille (a, b) =
   let i = dichotomie b ob_li.(a) and j = dichotomie a ob_co.(b) in
   [| a, ob_li.(a).(i) ; a, ob_li.(a).(i + 1) - 1 ;
      ob_co.(b).(j), b ; ob_co.(b).(j + 1) - 1, b |];;
```

Question 3 On construit une matrice vide qu'on remplit case par case avec la fonction précédente :

```
let matrice_deplacements () =
   let m = Array.make_matrix n n [||] in
   for a = 0 to n - 1 do
      for b = 0 to n - 1 do
         m.(a).(b) <- deplacements_grille (a, b)
      done
   done;
   m;;
```

Notons que la complexité de `deplacements_grille` est logarithmique en N, car on y fait deux appels à `dichotomie`. De par la présence des deux boucles `for`, on en déduit que la complexité de cette fonction est $O(N^2 \log N)$.

Question 4
On ne fait d'éventuelles modifications qu'en cas d'égalité d'une des deux coordonnées. On prend alors le plus proche des deux points entre l'obstacle et le robot. On suppose ici que les deux points sont distincts.

```
let modif t (a, b) (c, d) =
   if a = c then begin
      if b > d then
         t.(0) <- (a, max (d + 1) (snd t.(0)))
      else
         t.(1) <- (a, min (d - 1) (snd t.(1)))
   end;
   if b = d then begin
      if a > c then
         t.(2) <- (max (c + 1) (fst t.(2)), b)
      else
         t.(3) <- (min (c - 1) (fst t.(3)), b)
   end;;
```

Question 5 On commence par récupérer le vecteur de déplacements (qu'on copie) en utilisant la matrice de déplacements. La fonction auxiliaire `maj` prend en argument une liste de points et met à jour le vecteur de déplacements en utilisant la fonction précédente.

```
let deplacements_robots (a, b) q =
   let t = Array.copy mat_deplacements.(a).(b) in
   let rec maj q = match q with
      | [] -> ()
      | (c, d) :: tl -> modif t (a, b) (c, d); maj tl in
   maj q;
   t;;
```

Question 6 À chaque itération, on peut déplacer chacun des 4 robots dans une des 4 directions. Cela fait un total de 16 déplacements possibles. On en déduit que le nombre de suite de k déplacements est 16^k. Pour connaître le résultat d'un déplacement, il faut utiliser la fonction précédente qui est exécutée en temps constant (car le nombre de robots est constant). La complexité totale est donc $O(N^2 \log N + 16^k)$ (car il est nécessaire de créer la matrice de déplacements).

2 Quelques fonctions utilitaires

2.1 Une fonction de tri

Question 7 Un classique des fonctions récursives. On compare le premier élément de la liste à x et on agit en conséquence.

```
let rec insertion x q = match q with
   | [] -> [x]
   | y :: tl when x <= y -> x :: y :: tl
   | y :: tl -> y :: (insertion x tl);;
```

Question 8 On trie récursivement la queue de la liste et on lui insère le premier élément.

```
let rec tri_insertion q = match q with
   | [] -> []
   | x :: tl -> insertion x (tri_insertion tl);;
```

Question 9 Dans le pire des cas, la complexité est quadratique en la taille de la liste (par exemple pour une liste triée par ordre décroissant). Dans le meilleure des cas, la complexité est linéaire (pour une liste triée).
Soit $L = (a_0, \dots, a_k, x, a_{k+1}, \dots a_{N-1})$ une liste, telle que seul l'élément x n'est pas à sa place. Regardons alors ce que fait `tri_insertion` sur cette liste.

- Quand `tri_insertion` sera appelé sur la sous-liste $a_{k+1}, \dots a_{N-1}$, d'après ce qui est dit plus haut, chaque élément est directement situé à sa place, en $O(1)$ pour chaque élément.
- Si $x > a_{k+1}$, alors l'insertion de x se fera en l'insérant progressivement à droite jusqu'à sa position. Cette insertion est alors en $O(n)$, et les insertions suivantes se feront alors en $O(1)$. Au final, la complexité est en $O(N)$.
- Si $x < a_k$, alors l'insertion de x se fera en $O(1)$. Par la suite, les éléments qui sont plus grands que x seront insérés derrière x, ce qui se fera avec deux appels à la fonction `insertion`, donc en $O(1)$. Au final, la complexité est en $O(N)$.

Ainsi, le programme est en $O(N)$ dans tous les cas où un seul élément n'est pas à sa place.

2.2 Quelques fonctions sur les listes

Question 10 On teste les éléments un par un jusqu'à trouver le bon (ou renvoyer `false` si on arrive à la fin de la liste).

```
let rec mem1 x q = match q with
   | [] -> false
   | (y, _) :: tl when x = y -> true
   | _ :: tl -> mem1 x tl;;
```

Question 11 Quasi identique à la fonction précédente.

```
let rec assoc x q = match q with
   | [] -> failwith "Non␣trouv\'e"
   | (y, z) :: tl when x = y -> z
   | _ :: tl -> assoc x tl;;
```

3 Tables de hachage

3.1 Une famille de fonctions h_w

Question 12 On utilise la règle de Horner pour faire le calcul et éviter de calculer les puissances de N.

```
let rec hachage_liste w q = match q with
   | [] -> 0
   | (a, b) :: tl -> ((hachage_liste w tl) * n * n + a + b * n) mod w;;
```

3.2 Tables de hachage de largeur fixée

3.2.1 Implantation de la structure de dictionnaire

Question 13 Simple question sur la manipulation de type produits.

```
let creer_table h w =
   { hache = h; donnees = Array.make w []; largeur = w };;
```

Question 14 On utilise les fonctions de la partie II.B.

```
let recherche t k =
   mem1 k t.donnees.(t.hache k);;
```

Question 15

```
let element t k =
   assoc k t.donnees.(t.hache k);;
```

Question 16 On s'assure juste que la clé n'est pas déjà présente.

```
let ajout t k e =
   if not (recherche t k) then
      let hk = t.hache k in
      t.donnees.(hk) <- (k, e) :: t.donnees.(hk);;
```

Question 17 On commence par écrire une fonction auxiliaire qui supprime un élément d'une liste.

```
let suppression t k =
   let rec supp q = match q with
      | [] -> []
      | (a, _) :: tl when a = k -> tl
      | x :: tl -> x :: (supp tl) in
   t.donnees.(t.hache k) <- supp t.donnees.(t.hache k);;
```

3.2.2 Étude de la complexité de la recherche d'un élément

Question 18 Dans un premier temps, notons que la recherche d'une clé dans une alvéole est linéaire en la taille de la liste de l'alvéole. De plus, de par le hachage uniforme, l'espérance de la complexité d'une clé non présente est égale à la moyenne des complexités de recherche dans chaque alvéole. Or, la taille moyenne d'une liste d'une alvéole est exactement $\alpha = \frac{n}{w}$. On en déduit que la complexité est bien $O(1+\alpha)$ (le 1 est présent car α peut être nul, et il faut $O(1)$ opération pour récupérer la liste de l'alvéole de toute façon).

Question 19 Si on prend une clé, chaque autre clé a une probabilité $1/w$ de se trouver dans la même alvéole. L'espérance de la taille de cette alvéole est donc $1 + \frac{n-1}{w} \leq 1 + \alpha$. La recherche se fera avec une complexité $O(1 + \alpha)$ ici encore.

3.3 Tables de hachage dynamique

Question 20

```
let creer_table_dyn h =
   { hache = h; taille = 0; donnees = [| [] |]; largeur = 1 };;
```

Question 21 On utilise une fonction auxiliaire `ajout` qui ajoute tous les éléments d'une liste à un tableau de liste créé précédemment, en hachant chaque clé avec la nouvelle fonction h_{w_2}. On l'applique à toutes les listes de la table de hachage. Une fois que c'est terminé, on modifie la table et sa largeur.

```
let rearrange_dyn t w2 =
   let d = Array.make w2 [] in
   let rec ajout q = match q with
      | [] -> ()
      | (a, b) :: tl -> d.(t.hache w2 a) <- (a, b) :: d.(t.hache w2 a);
                        ajout tl in
   for i = 0 to t.largeur - 1 do
      ajout t.donnees.(i)
   done;
   t.donnees <- d;
   t.largeur <- w2;;
```

La complexité de création du vecteur de listes `d` est $O(w_2)$. La fonction `ajout` a une complexité linéaire en la taille de la liste. La boucle `for` est de taille w, et la complexité totale est de l'ordre de la somme des tailles de toutes les listes, c'est-à-dire $O(n)$. On en déduit bien une complexité totale en $O(n + w + w_2)$.

Question 22 On ajoute un élément comme en question 16, puis on réarrange la table si nécessaire.

```
let ajout_dyn t k e =
   if not (recherche_dyn t k) then begin
      let i = t.hache t.largeur k in
      t.donnees.(i) <- (k, e) :: t.donnees.(i);
      t.taille <- t.taille + 1
   end;
   if t.taille > 3 * t.largeur then
      rearrange_dyn t (3 * t.largeur);;
```

4 Résolution du jeu des robots

4.1 Graphe orienté associé au jeu des robots

Question 23 Pour déterminer un sommet, il faut dans un premier temps déterminer les positions de tous les robots, soit $\binom{N^2}{p}$ possibilités, puis choisir le robot principal, soit p possibilités. Il y a donc $p\binom{N^2}{p}$ sommets dans le graphe. Cela représente 699 170 560 sommets pour $p = 4$ et $N = 16$.

Question 24 On utilise une fonction auxiliaire `creer` qui crée un sommet (pour faciliter l'écriture). De plus, on utilise une fonction `supp` qui supprime un élément dans une liste.
Pour la fonction `sommets_accessibles`, on crée une référence de liste qu'on modifiera chaque fois qu'il faudra ajouter un sommet.
On commence par traiter le cas du déplacement du robot principal, puis on utilise une fonction récursive `ajout_dpt` qui, pour chaque autre robot, calcule ses déplacements possibles, crée les sommets correspondants et les ajoute à la liste.

```
let creer x q = {robot = x; autres_robots = q};;

let rec supp x q = match q with
   | [] -> []
   | y :: tl when x = y -> tl
   | y :: tl -> y :: (supp x tl);;

let sommets_accessibles s =
   let l = ref [] in
   let x = s.robot and q = s.autres_robots in
   let d = deplacements_robots x q in
   for i = 0 to 3 do
      l := (creer d.(i) q) :: !l
   done;
   let rec ajout_dpt q = match q with
      | [] -> ()
      | y :: tl -> let r = supp y q in
                   let d = deplacements_robots y (x :: r) in
                   for i = 0 to 3 do
                      l := (creer x (insertion d.(i) r)) :: !l
                   done;
                   ajout_dpt tl in
    ajout_dpt q;
    !l;;
```

4.2 Parcours en largeur : étude théorique

Question 25 La terminaison ne concerne ici que la boucle **tant que**, car la boucle **pour** termine toujours. Chaque sommet ne peut être ajouté dans la file qu'une seule fois au plus. En effet, un sommet ne sera ajouté s' à la file que si $b_{s'}$ est faux et dans ce cas, on affecte la valeur vraie à $b_{s'}$ ce qui garantit que le sommet ne sera plus ajouté.
A chaque passage dans la boucle **while**, on enlève un sommet à la file. La boucle termine donc après au plus $|S|$ itérations de la boucle **tant que**

Question 26 Soit $s_0 \to s_1 \to \cdots \to s_k = s$ un chemin de s_0 à s. Si on suppose que s n'est pas visité par l'algorithme, alors il existe $i \in [\![1, k]\!]$ tel que b_{s_i} est faux et $b_{s_{i-1}}$ est vrai. Dans ce cas, le sommet s_{i-1} est passé par la file, et lorsqu'il a été défilé, tous ses voisins dont le marqueur valait **faux** ont été marqué en **vrai**, en particulier s_i. Par l'absurde, on en déduit le résultat.

Question 27 Le tableau π permet de retrouver les précédesseurs. On retrouve donc le chemin en le construisant à l'envers : en utilisant les mêmes notations que la question précédente, on a $s_{i-1} = \pi[s_i]$ pour tout $i \in [\![1, k]\!]$.

Question 28 Utilisons quelques notations. Pour $s \in S$, notons d_s la distance de s_0 à s, ℓ_s la longueur du chemin de s_0 à s donné par l'algorithme de parcours en largeur et n_s le rang d'insertion du sommet dans la file d'attente. Prouvons la propriété suivante par induction sur n_s :

1. $d_s = \ell_s$.
2. Pour tout sommet t, $d_t < d_s \Rightarrow n_t < n_s$.

Initialement, comme $n_{s_0} = 1$, la propriété est bien vérifiée. Supposons alors la propriété vraie pour tous les sommets jusqu'au numéro n et soit s le sommet tel que $n_s = n + 1$.

1. Supposons que $d_s < \ell_s$. Et soit un plus court chemin $s_0 \rightsquigarrow t \to s$. Notons de plus u le prédécesseur de s dans π. On a $n_u < n_s$ par hypothèse. De plus, comme on a supposé $d_s < \ell_s$, alors $d_t < d_u$ et donc $n_t < n_u$. Dans ce cas, le sommet t a été défilé avant le sommet u, et donc le prédécesseur de s aurait dû être t et non u. Par l'absurde, on en déduit que $d_s = \ell_s$.
2. Supposons qu'il existe un sommet t tel que $d_t < d_s$ et $n_t > n_s$. Soit $s_0 \rightsquigarrow t' \to t$ un plus court chemin de s_0 à t. Alors $s_0 \rightsquigarrow t'$ est un plus court chemin de s_0 à t' et donc $d_{t'} = d_t - 1$.
   Soit alors s' le prédécesseur de s dans π. On a, par hypothèse, $n_{s'} < n_s$ et par le point prouvé précédemment, $d_{s'} = \ell_{s'} = \ell_s - 1 = d_s - 1$.
   Finalement, on a $d_{t'} < d_{s'}$, et donc par hypothèse d'induction $n_{t'} < n_{s'}$. Cela signifie que t' a été défilé avant s' et donc que t aurait dû être enfilé avant s. Par l'absurde on a bien le second point.

Question 29 Remarquons que chaque arc et chaque sommet n'est parcouru ou visité qu'une seule fois dans le pire des cas. On en déduit que la complexité totale est en $O(|S| + |A|)$.