Devoir surveillé n° 4 — Concours blanc — 4h

L'usage de la calculatrice est interdit. Les programmes seront rédigés clairement et proprement, en mettant en valeur l'indentation et en choisissant des noms de variables explicites. On utilisera, si possible, une couleur différente pour le code et pour les commentaires. Les programmes non triviaux doivent être commentés dès qu'il y a une difficulté et il est fortement conseillé de commencer par un brouillon avant de recopier la version finale au propre.

⚠ Les parties I et II d'une part et la partie III d'autre part sont indépendantes et sont à rédiger sur des copies séparées, avec votre nom, prénom et classe.

I Quelques fonctions utilitaires

I.1 Quelques fonctions sur les tableaux

On rappelle la signature des fonctions suivantes, disponibles dans la bibliothèque de CAML, que l'on pourra utiliser librement dans toute la suite :

- `Array.make : int -> 'a -> 'a array`
- `Array.make_matrix : int -> int -> 'a -> 'a array`
- `Array.copy : 'a array -> 'a array`

Q 1. Pourquoi ne peut-on pas utiliser directement `Array.copy` pour copier une matrice ? Écrire une fonction `copy_matrix : 'a array array -> 'a array array` qui réalise la copie d'une matrice.

I.2 Une fonction de tri

Q 2. Écrire une fonction `insertion` de type `'a -> 'a list -> 'a list` prenant en entrée un élément `elt` et une liste `lst` triée dans l'ordre croissant, et renvoyant une liste triée dans l'ordre croissant, constituée de `elt` et des éléments de `lst`.

Q 3. En déduire une fonction `tri_insertion` de type `'a list -> 'a list` permettant de trier une liste dans l'ordre croissant.

Q 4. Rappeler la complexité de ce tri dans le pire et le meilleur cas. Que peut-on dire de la complexité si tous les éléments de la liste excepté peut-être un sont dans l'ordre croissant ?

I.3 Quelques fonctions sur les listes

Q 5. Écrire une fonction `mem1` de type `'a -> ('a * 'b) list -> bool` telle que `mem1 k liste` renvoie vrai si et seulement si la liste comporte un couple dont le premier élément est `k`.

Q 6. Écrire une fonction `assoc` de type `'a -> ('a * 'b) list -> 'b` telle que `assoc k liste` renvoie l'élément `e` du premier couple `(k, e)` appartenant à la liste `liste`. On renverra une erreur si un tel couple n'existe pas.

Q 7. Écrire une fonction `suppr` de type `'a -> ('a * 'b) list -> ('a * 'b) list` telle que l'appel `suppr k liste` supprime le premier couple `(k, e)` appartenant à la liste `liste` s'il existe et renvoie la liste inchangée sinon.

II Tables de hachage

Une structure de dictionnaire est un ensemble de couples (clé, élément). Les clés (nécessairement distinctes) appartenant à un même ensemble K, les éléments à un ensemble E. Par exemple, les clés peuvent être des mots et les éléments des définitions de ces mots. Les clés peuvent aussi être des adresses URL et les éléments le contenu HTML des pages associées. Plus formellement, un dictionnaire de K vers E est une partie de $K \times E$ telle que pour toute clé $k \in K$ il existe au plus un élément $e \in E$ tel que le couple (k, e) appartienne au dictionnaire.

La structure de dictionnaire doit garantir les opérations suivantes :

- recherche d'un élément avec sa clé ;
- ajout d'un couple (clé, élément) ;
- suppression d'un couple connaissant sa clé.

Une implémentation simple d'un dictionnaire peut naturellement être effectuée à l'aide d'une liste de couples de $K \times E$. On peut alors utiliser directement les fonctions `mem1`, `assoc` et `suppr` de la partie I.3.

Q 8. Quelle est alors la complexité dans le pire cas des trois opérations en fonction du nombre n de couples (clé, élément) présents dans le dictionnaire ?

Si les clés sont des entiers de petite taille, par exemple si $K = [\![0, w-1]\!]$ avec $w \in \mathbb{N}^*$, on peut alors directement utiliser un tableau de longueur w. On place directement l'élément d'un couple (clé, élément) dans la case correspondant à sa clé. Mais en général, les clés ne sont pas des entiers et même lorsque c'est le cas, w est souvent très grand. De plus, de nombreuses cases sont vides, ce qui conduit à un gaspillage de mémoire.

Les tables de hachage (*hash tables* en anglais) sont une implémentation d'un dictionnaire plus efficace qu'une liste de couples et généralisant l'idée d'un tableau d'association. On se donne une fonction $h_w : K \to [\![0, w-1]\!]$, appelée *fonction de hachage* et un tableau de w listes (appelées des *alvéoles*) de couples (clé, élément). Le tableau est organisé de façon à ce que la liste d'indice i contienne tous les couples (k, e) de la table tels que $h_w(k) = i$. On appelle w la *largeur* de la table et $h_w(k)$ le *haché* de la clé k.

Ainsi, pour rechercher ou supprimer l'élément de clé k, on commence par calculer son haché qui détermine l'alvéole adéquate et on est alors ramené à une action sur la liste correspondante. De même, pour ajouter un nouvel élément au dictionnaire, on l'ajoute à l'alvéole indiquée par le haché de sa clé.

II.1 Une famille de fonctions h_w

Pour implémenter une table de hachage, il est nécessaire de disposer d'une fonction de hachage h_w de K vers $[\![0, w-1]\!]$. Donnons un exemple de fonction de hachage dans le cas où les clés sont des chaînes de caractères. Le code ASCII d'un caractère est un entier compris entre 0 et 255 identifiant le caractère de manière unique, ce que l'on obtient en CAML avec la fonction `int_of_char : char -> int`. Une chaîne de caractères $s = c_0 \dots c_{n-1}$ peut alors être vue comme la représentation en base 256 de l'entier :

$$\sum_{k=0}^{n-1} \text{code}(c_k) \times 256^k$$

On propose le code CAML suivant :

OCAML
```
let hache_chaine w s =
  let codes_ascii = List.map int_of_char s in
  let rec eval liste =
    match liste with
    | [] -> 0
    | c :: suite -> (c + 256 * (eval suite)) mod w
  in
  eval codes_ascii
```

Q 9. Donner le type de la fonction `hache_chaine`, expliquer succinctement son comportement et indiquer pourquoi on peut l'utiliser comme une fonction de hachage pour les chaînes de caractères.

II.2 Tables de hachage de largeur fixe

Dans cette sous-section, on fixe une largeur de hachage w. On définit en toute généralité le type suivant :

OCaml
```
type ('a, 'b) table_hachage = {
   hache: 'a -> int;
   donnees: ('a * 'b) list array;
   largeur: int
}
```

Q 10. Écrire une fonction `creer_table` de type `('a -> int) -> int -> ('a, 'b) table_hachage` telle que `creer_table h w` renvoie une nouvelle table de hachage vide de largeur `w` munie de la fonction de hachage `h`.

Q 11. Écrire une fonction `recherche` de type `('a, 'b) table_hachage -> 'a -> bool` telle que `recherche t k` renvoie un booléen indiquant si la clé k est présente dans la table t. On pourra utiliser les fonctions de la partie I.3.

Q 12. Écrire une fonction `element` de type `('a, 'b) table_hachage -> 'a -> 'b` telle que `element t k` renvoie l'élément e associé à la clé k dans la table t, si cette clé est bien présente dans la table et une erreur sinon.

Q 13. Écrire une fonction `ajout` de type `('a, 'b) table_hachage -> 'a -> 'b -> unit` telle que `ajout t k e` ajoute l'entrée (k, e) à la table de hachage t. On n'effectuera aucun changement si la clé est déjà présente.

Q 14. Écrire enfin une fonction `suppression` de type `('a, 'b) table_hachage -> 'a -> unit` telle que `suppression t k` supprime l'entrée de la clé k dans la table t. On n'effectuera aucun changement si la clé n'est pas présente.

II.3 Étude de la complexité de la recherche d'un élément

Q 15. Dans le pire cas, quelle est la complexité des trois opérations (appartenance, ajout, suppression) en fonction du nombre n de couples (clé, élément) présents dans la table de hachage ?

Si la fonction de hachage h_w est bien choisie, on peut espérer que les clés vont se répartir de façon relativement uniforme dans les alvéoles, ce qui donnera une complexité bien meilleure.

Nous faisons ici l'hypothèse de *hachage uniforme simple* : pour une clé donnée, la probabilité d'être hachée dans l'alvéole i est $1/w$, indépendante des autres clés. On note n le nombre de clés stockées dans la table et on appelle $\alpha = n/w$ le *facteur de remplissage de la table*. On suppose de plus, que le calcul du *haché* d'une clé se fait en temps constant.

Q 16. On se donne une clé k non présente dans la table, on note $i = h_w(k)$ son haché et n_i la taille de l'alvéole d'indice i. Montrer que la complexité de la recherche de k dans la table est un $O(1 + n_i)$.

Q 17. En déduire que l'espérance de la complexité de la recherche d'une clé k non présente dans la table est un $O(1 + \alpha)$.

On admet que la recherche d'une clé présente dans la table se fait aussi, en moyenne, en $O(1 + \alpha)$.

II.4 Tables de hachage dynamiques

On peut donc assurer une complexité moyenne de recherche dans une table de hachage constante, sous réserve que le facteur de remplissage α soit borné. Il en va de même des opérations d'insertion et de suppression, pour peu que les clés à ajouter/supprimer vérifient des hypothèses d'indépendance. Mais

souvent, on ne sait pas à l'avance quel sera le nombre de clés à stocker dans la table, et on préfère ne pas surestimer ce nombre pour garder un espace mémoire linéaire en le nombre de clés stockées. Ainsi, il est utile de faire varier la largeur w de la table de hachage : si le facteur de remplissage devient trop important, on réarrange la table sur une largeur plus grande (de même, on peut réduire la largeur de la table lorsque le facteur de remplissage devient petit). On parle alors de tables de hachage dynamiques pour ces tables de hachage à largeur variable.

À une table de hachage dynamique est associée une *famille de fonctions de hachage* (h_w). Par exemple, pour les chaînes de caractères, la fonction `hache_chaine` précédemment écrite fournit une telle famille.

On définit en toute généralité le type suivant :

OCaml
```
type ('a,'b) table_dyn = {
   hache: int -> 'a -> int;
   mutable taille: int;
   mutable donnees: ('a * 'b) list array;
   mutable largeur: int};;
```

On notera trois différences par rapport au type précédent :

- La fonction `hache` possède un paramètre supplémentaire qui est la largeur de hachage : elle correspond maintenant à la famille de fonctions de hachage (h_w) ;
- on a rendu les champs `donnees` et `largeur` modifiables ;
- un champ `taille` (modifiable) est rajouté, il doit à tout moment contenir le nombre de clés présentes dans la table.

Q 18. Écrire une fonction `creer_table_dyn` de type `(int -> 'a -> int) -> ('a, 'b) table_dyn` telle que `creer_table_dyn h` renvoie une table de hachage dynamique initialement vide, avec la famille de fonctions de hachage `h` et de largeur initiale 1.

On peut réécrire les deux fonctions `recherche_dyn` et `element_dyn`, variantes des fonctions `recherche` et `element` précédentes, basées sur le même principe :

OCaml
```
let recherche_dyn t k = mem1 k t.donnees.(t.hache t.largeur k)

let element_dyn t k = assoc k t.donnees.(t.hache t.largeur k)
```

On va maintenant développer une stratégie pour maintenir à tout moment un facteur de remplissage borné.

Q 19. Écrire une fonction `rearrange_dyn` de type `('a, 'b) table_dyn -> int -> unit` telle que `rearrange_dyn t w2` prend en entrée une table de hachage dynamique et une nouvelle largeur de hachage `w2`, et réarrange la table sur une largeur `w2`. En supposant que le calcul des valeurs de hachage se fasse en temps constant, la complexité doit être en $O(n + w + w_2)$ où n est le nombre de clés présentes dans la table (sa taille), w est l'ancienne largeur de la table, w_2 la nouvelle. On justifiera cette complexité.

Une stratégie heuristique simple pour garantir que le facteur de remplissage reste borné est d'utiliser les puissances de 3 comme largeurs de hachage. Après ajout d'un élément à la table, si celle-ci est de taille strictement supérieure à trois fois sa largeur w, on la réarrange sur une largeur $w_2 = 3w$.

Q 20. Écrire une fonction `ajout_dyn` de type `('a, 'b) table_dyn -> 'a -> 'b -> unit` telle que `ajout_dyn t k e` ajoute le couple (k, e) à la table de hachage (si la clé k n'est pas présente), en réarrangeant si nécessaire la table, en suivant le principe ci-dessus.

Dans l'hypothèse que chaque ajout se fait en temps $O(1 + \alpha)$, où α est le facteur de remplissage de la table, on peut montrer qu'une série de p ajouts dans une table initialement vide prend un temps $O(p)$.

On pourrait écrire de même une fonction de suppression dynamique.

III Le jeu des robots

Dans cette partie on s'intéresse au jeu *Ricochet Robots*. Ce jeu se déroule sur un plateau de 16×16 cases, avec 4 robots et des murs. À chaque tour de jeu, on peut déplacer un des robots dans une des quatre directions. Le robot se déplace alors dans cette direction sans s'arrêter jusqu'à rencontrer un obstacle (un mur ou autre robot), ce qui compte pour un mouvement (un coup). Le but est de trouver le nombre de coups minimal pour amener le robot principal n° 1 de son point de départ jusqu'à une case précise du plateau. Un exemple en 8 coups est donné sur la figure 1. Une autre configuration initiale est donnée à la figure 2.

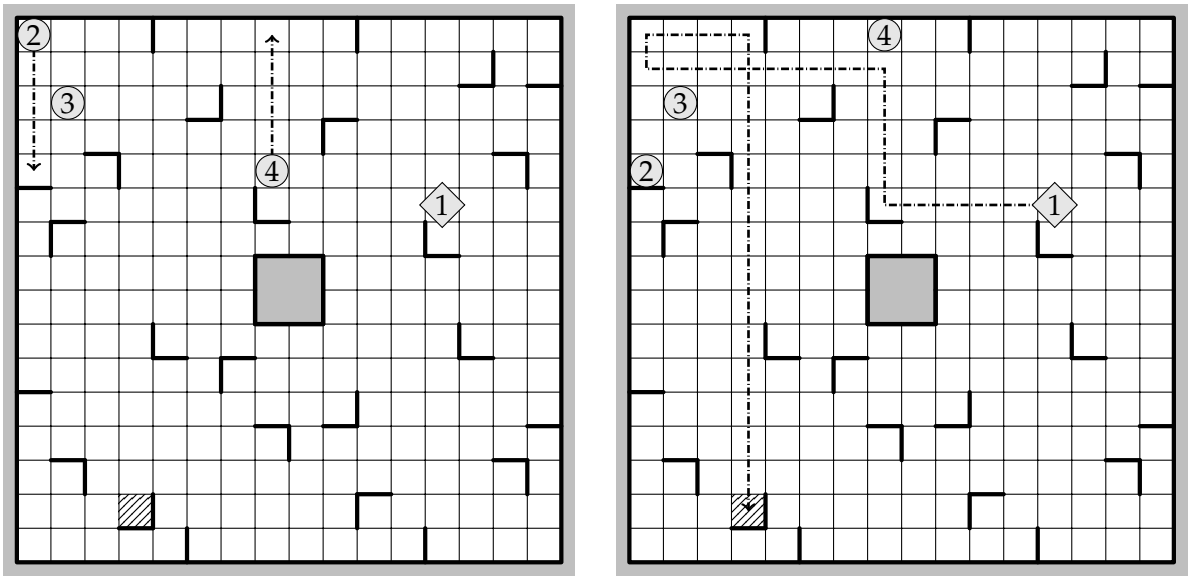

FIGURE 1 – Le but est d'amener le robot n° 1 sur la case hachurée. À gauche : un déplacement du robot n° 2 puis n° 4 ; puis, à droite : six déplacements du robot n° 1. Le jeu est résolu en 8 coups (solution optimale).

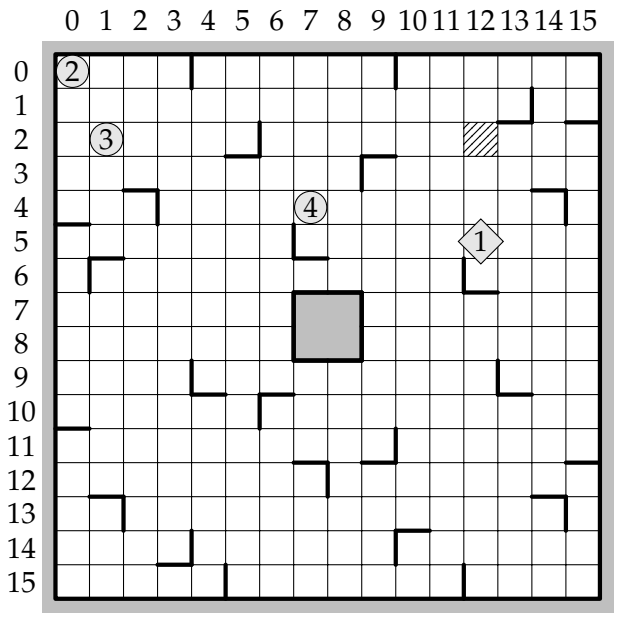

FIGURE 2 – Une configuration à résoudre : la case à atteindre par le robot n° 1 est la case $(2, 12)$.

Q 21. Expliquer succinctement pourquoi il est impossible d'amener le robot principal (le robot n° 1) sur la case hachurée de coordonnées $(2, 12)$ sans utiliser les autres robots (figure 2).

Q 22. Trouver une suite de 8 déplacements permettant de résoudre cette instance du jeu (8 coups est une solution optimale).

On considère pour le moment une grille sans robots. Notons N le nombre de cases par ligne et colonne de la grille (16 dans le jeu originel). *Dans toutes les fonctions demandées, on supposera que N est une variable globale.*

On numérote chaque case par un couple (a, b) de $[\![0, N-1]\!]^2$, correspondant à la ligne a et à la colonne b. On numérote également les lignes horizontales et verticales séparant les cases à l'aide d'un entier de $[\![0, N]\!]$, de sorte que la case (a, b) est délimitée par les lignes horizontales a (au-dessus) et $a+1$ (en dessous), de même que par les lignes verticales b (à gauche) et $b+1$ (à droite). Cf. figure 3.

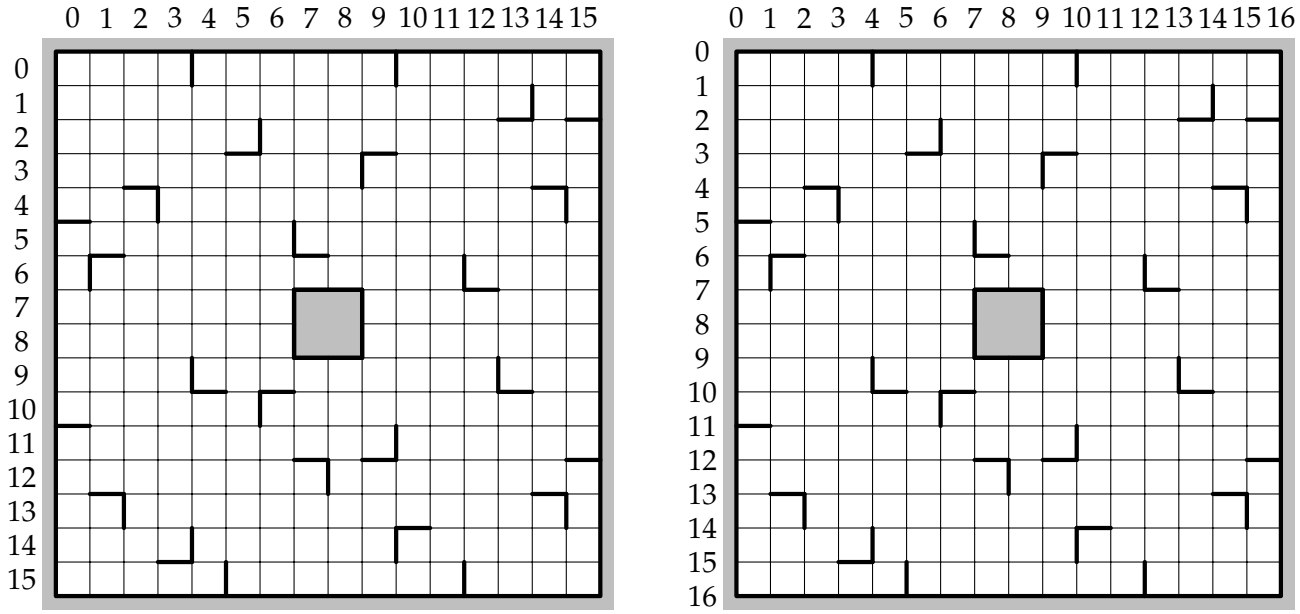

FIGURE 3 – À gauche : numérotation des cases par ligne/colonne ; à droite : numérotation des lignes horizontales et verticales.

Pour représenter en CAML la grille avec ses obstacles, on se donne deux tableaux de taille N. Le premier contient les obstacles verticaux sur chacune des lignes, le second contient les obstacles horizontaux sur chacune des colonnes. Un obstacle est donné par le numéro de la ligne (verticale ou horizontale) auquel il appartient. Les obstacles sur une ligne (ou colonne) sont donnés sous la forme d'un tableau ordonné dans l'ordre croissant. Par exemple, la représentation du plateau de la figure 1 est donnée par les deux variables globales suivantes :

OCAML
```
let obstacles_lignes = [|
  [|0; 4; 10; 16|];
  [|0; 14; 16|];
  [|0; 6; 16|];
  [|0; 9; 16|];
  [|0; 3; 15; 16|];
  [|0; 7; 16|];
  [|0; 1; 12; 16|];
  [|0; 7; 9; 16|];
  [|0; 7; 9; 16|];
  [|0; 4; 13; 16|];
  [|0; 6; 16|];
  [|0; 10; 16|];
  [|0; 8; 16|];
  [|0; 2; 15; 16|];
  [|0; 4; 10; 16|];
  [|0; 5; 12; 16|]
|]
```

OCAML
```
let obstacles_colonnes = [|
  [|0; 5; 11; 16|];
  [|0; 6; 13; 16|];
  [|0; 4; 16|];
  [|0; 15; 16|];
  [|0; 10; 16|];
  [|0; 3; 16|];
  [|0; 10; 16|];
  [|0; 6; 7; 9; 12; 16|];
  [|0; 7; 9; 16|];
  [|0; 3; 12; 16|];
  [|0; 14; 16|];
  [|0; 16|];
  [|0; 7; 16|];
  [|0; 2; 10; 16|];
  [|0; 4; 13; 16|];
  [|0; 2; 12; 16|]
|]
```

Notez que les bordures de la grille sont considérées comme des obstacles. Ainsi, les entiers 0 et N sont présents dans les tableaux associés à chaque ligne/colonne. *Dans les fonctions demandées, on supposera que les tableaux* `obstacles_lignes` *et* `obstacles_colonnes` *sont des variables globales.*

On va commencer par écrire une fonction `dichotomie` de signature `int -> int array -> int` telle que si `t` est un tableau d'entiers strictement croissants et `a` un élément supérieur ou égal au premier élément du tableau et strictement inférieur au dernier, `dichotomie a t` renvoie l'unique indice `i` tel que `t.(i) <= a < t.(i+1)`. Attention, on ne suppose pas que `a` est dans le tableau. On va procéder par dichotomie pour avoir une complexité logarithmique en la taille du tableau.

Q 23. Écrire cette fonction `dichotomie` dans un style impératif (sans utiliser de fonctions récursives). Trouver (en justifiant) un invariant de boucle et prouver la correction de la fonction. En exhibant un variant de boucle, et en faisant bien attention, montrer que cette fonction termine. On attend une complexité logarithmique en la taille du tableau, mais on ne demande pas de la justifier.

Q 24. Écrire cette fonction `dichotomie` en utilisant une fonction récursive auxiliaire (et sans utiliser de références ou d'effets de bords). On fera bien attention au choix du cas de base.

On considère un robot positionné en (a, b), avec $0 \leq a, b < N$. Il y a 4 déplacements possibles, dans les 4 directions cardinales (ouest/est/nord/sud) représentées sur la figure 4). On rappelle que le robot se déplace jusqu'à rencontrer un obstacle. Si le robot est contre un obstacle dans une direction donnée, alors il ne bouge pas, c'est-à-dire que l'on considère que le résultat du déplacement dans cette direction est la case (a, b) elle-même.

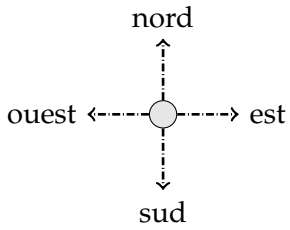

FIGURE 4 – Déplacements cardinaux (ouest/est/nord/sud).

Q 25. Écrire une fonction `deplacements_grille` de type `int * int -> (int * int) array` telle que `deplacement_grille (a, b)` renvoie, dans cet ordre, les 4 cases atteintes par les déplacements ouest/est/nord/sud d'un robot positionné en `(a, b)`, sous forme d'un tableau à 4 éléments. On demande une complexité en $O(\log N)$, ce que l'on justifiera rapidement. *Indication : on utilisera astucieusement la fonction* `dichotomie`.

Q 26. Écrire une fonction `matrice_deplacements : unit -> (int * int) array array array` produisant une matrice `m` telle que `m.(a).(b)` contienne le tableau des déplacements possibles pour un robot depuis la case (a, b), et ce pour tous $0 \leq a, b < N$. Donner la complexité de création de la matrice.

On cherche maintenant à intégrer les positions d'autres robots dans le déplacement d'un robot. On utilise la fonction précédente pour créer une matrice `mat_deplacements` que l'on considérera comme globale et que l'on évitera donc de recréer inutilement.

OCAML
```
let mat_deplacements = matrice_deplacements ()
```

Q 27. Écrire une fonction `modif : (int * int) array -> int * int -> int * int -> unit` telle que si `t` est le tableau de taille 4 donnant les déplacements ouest/est/nord/sud d'un robot placé en (a, b) dans la grille ne contenant pas d'autres robots et (c, d) la position d'un autre robot, alors l'appel `modif t (a, b) (c, d)` modifie, si nécessaire, le tableau `t` en prenant en compte le robot en (c, d). On garantira une complexité en $\Theta(1)$.

On s'intéresse maintenant au déplacement d'un robot situé en (a, b) dans la grille, avec d'autres robots éventuellement présents, dont les positions sont stockées dans une liste.

Q 28. Déduire des questions précédentes une fonction `deplacements_robots` de type `int * int -> (int * int) list -> (int * int) array` telle que `deplacement_robots (a, b) bots` donne le tableau des déplacements ouest/est/nord/sud d'un robot situé en (a, b) dans la grille, les positions des autres robots étant stockées dans la liste `bots`. Attention : on ne modifiera pas la matrice `mat_deplacements` : on souhaite une copie modifiée de `mat_deplacements.(a).(b)`. On garantira une complexité linéaire en le nombre de robots, et en particulier indépendante de N.

Une configuration du jeu est la donnée des positions des robots. On distingue le *robot principal* (celui que l'on veut amener sur une case donnée) et les autres robots. Ainsi, on représente une configuration avec le type suivant :

OCaml
```
type configuration = {robot: int * int; autres: (int * int) list};;
```

On impose que la liste `autres` soit toujours triée dans l'ordre croissant en suivant l'ordre lexicographique (l'ordre naturel pour les couples en CAML). Cet ordre est défini par $(a, b) \leq (a', b')$ si et seulement si $a < a'$ ou $a = a'$ et $b < b'$. Par exemple, CAML évalue l'expression `(2, 3) < (3, 0)` en `true`.

Q 29. Donner une expression CAML qui représente la configuration de la figure 2.

Q 30. Avec p robots en tout sur un plateau de taille $N \times N$, quel est le nombre de configurations possibles ?

Une configuration c' est *voisine* d'une configuration c si on peut passer de c à c' par un mouvement licite d'un des robots.

Q 31. Écrire une fonction `configuration_suivantes` de type `configuration -> configuration list` prenant en entrée une configuration et renvoyant la liste des configurations voisines. S'il y a p robots en tout, la fonction renverra donc une liste de $4p$ configurations. Certaines configurations de cette liste peuvent être égales : elles correspondent au mouvement d'un robot dans une direction où il est bloqué. Exceptionnellement, on pourra utiliser une référence de liste, qui simplifie l'écriture.

Q 32. Si on suppose que la solution optimale demande au plus k mouvements, une solution possible pour résoudre le jeu consiste à générer toutes les suites possibles de k déplacements. Avec p robots en tout, estimer la complexité d'une telle approche (on utilisera la notation O) en fonction de p, k et N, en justifiant précisément.

— FIN DU SUJET —