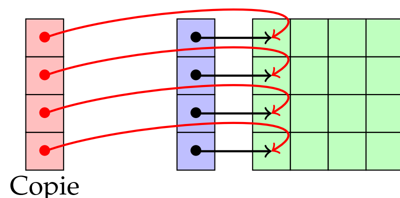
Devoir surveillé n° 4 — Corrigé

I Quelques fonctions utilitaires

Q 1. Une copie directe d'une matrice (un tableau de tableaux) renvoie un *nouveau* tableau dont les cases contiennent les mêmes valeurs que l'ancien, c'est-à-dire les *anciens* sous-tableaux. Une modification d'une case de la copie entraîne donc une modification de la matrice initiale.



Il faut donc faire une copie de chaque tableau ligne :

OCaml

```
let copy_matrix mat =
  let n = Array.length mat in
  let res = Array.make n [||] in
  for i = 0 to n - 1 do
    res.(i) <- Array.copy mat.(i)
  done;
  res
```

Q 2. Cf. exercice 5 du TD n° 3

Q 3. Cf. exercice 5 du TD n° 3

Q 4. Cf. exercice 5 du TD n° 3. Si tous les éléments sont triés par ordre croissant sauf peut-être un (ou même un nombre borné d'entre eux), la complexité reste linéaire (en cela le tri par insertion est intéressant). En effet, pour une liste de taille n, au pire, l'insertion de l'élément en question est en $O(n)$. L'insertion des $O(n)$ autres éléments a un coût de $O(1)$, soit $O(n) + O(n) = O(n)$ au final.

Q 5. Simple adaptation de la fonction `List.mem`.

OCaml

```
let rec mem1 k liste =
  match liste with
  | [] -> false
  | (a, b) :: suite -> (a = k) || mem1 k suite
```

Q 6. On adapte la fonction précédente.

OCaml

```
let rec assoc k liste =
  match liste with
  | [] -> failwith "assoc"
  | (a, b) :: _ when (a = k) -> b
  | _ :: suite -> assoc k suite
```

Q 7. On supprime le couple s'il est en tête sinon l'appel récursif s'en charge.

OCaml

```
let rec suppr k liste =
  match liste with
  | [] -> []
  | (a, b) :: suite when (a = k) -> suite
  | tete :: queue -> tete :: (suppr k queue)
```

II Tables de hachage

Q 8. Dans le pire cas, les trois complexités sont linéaires en fonction du nombre n de couples (clé, élément) présents dans le dictionnaire. Le pire cas correspond à une clé non présente pour `mem1` ou `suppr`, et en dernière position pour `assoc`.

Q 9. La fonction `hache_chaine` est de type `int -> char list -> int`. La première ligne permet de transformer une liste de caractères en la liste de leur codes

ASCII. La fonction auxiliaire `eval`, de type `int list -> int` évalue un polynôme représenté par `liste` au point 256, modulo w, par la méthode de Horner et calcule donc

$$\sum_{k=0}^{n-1} \mathrm{code}(s_k) \times 256^k \mod w$$

On a donc bien une fonction des chaînes de caractères vers $[\![0, w-1]\!]$. Remarquons qu'il est important de bien effectuer les modulo w à chaque étape et non une fois à la toute fin pour éviter le dépassement de capacité de représentation des entiers.

Q 10. Il suffit de bien lire le sujet et de connaître la syntaxe des enregistrements.

OCAML

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

Q 11. Tout est dans le sujet : « on commence par calculer son haché qui détermine l'alvéole adéquate et on est alors ramené à une action sur la liste correspondante ».

OCAML

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

Q 12. On laisse la gestion de l'erreur à `assoc`.

OCAML

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

Q 13. Il ne faut pas oublier le test d'appartenance.

OCAML

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

Q 14. Tester la présence en amont permet de ne pas reconstruire

OCAML

```
let supression t k =
  if not (recherche t k) then
    let i = t.hache k in
    t.donnees.(i) <- suppr k t.donnees.(i)
```

Q 15. Dans le pire cas, toutes les clés sont hachées vers la même alvéole et la complexité des trois opérations est toujours linéaire.

Q 16. La recherche de la clé k commence par le calcul de son haché qui prend un temps $\Theta(1)$ par hypothèse. Ensuite on est ramené à la complexité de `mem1` dans le pire cas puisque la clé n'est pas présente dans la table soit $O(n_i)$ ce qui donne bien la complexité annoncée en $O(1+n_i)$.

Q 17. L'espérance de la complexité de la recherche d'une clé k non présente dans la table est donc

$$\begin{aligned}\sum_{i=0}^{w-1} O(1+n_i)\mathbb{P}(h_w(k)=i) &= \sum_{i=0}^{w-1} O(1+n_i)\frac{1}{w} \\ &= O(1+\frac{1}{w}\sum_{i=0}^{w-1} n_i) \\ &= O(1+\frac{n}{w}) = O(1+\alpha)\end{aligned}$$

Q 18. La seule difficulté est de bien comprendre la structure de données et le type des objets.

OCAML

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

Q 19. On s'aide d'une fonction auxiliaire `migration` qui transvase les éléments d'une alvéole des anciennes données vers les nouvelles. Attention : il est nécessaire de rehacher toutes les clés car la fonction de hachage change ! On pourrait aussi utiliser `List.iter` ici. Il suffit ensuite d'effectuer la migration de chaque alvéole.

OCaml
```
let rearrange_dyn t w2 =
  let donnees2 = Array.make w2 [] in
  let h2 = t.hache w2 in
  let rec migration alveole =
    match alveole with
    | [] -> ()
    | (k, e) :: suite ->
      donnees2.(h2 k) <- (k, e) :: donnees2.(h2 k);
      migration suite
  in
  for i = 0 to t.largeur - 1 do
    migration t.donnees.(i)
  done;
  t.donnees <- donnees2;
  t.largeur <- w2
```

Il est nécessaire de créer un tableau de taille `w2` ce qui induit une complexité en $O(w2)$. En comptant à part l'appel à `migration`, il faut parcourir tous les éléments du premier tableau, avec donc une complexité en $O(w)$. Chaque élément de la table est contenu une et une seule fois dans une liste passée en argument à `migration`, d'où une complexité en $O(n)$. En ajoutant ces complexités on trouve bien la complexité demandée.

Q 20. Il ne faut pas oublier le test d'appartenance. On réarrange la table si nécessaire (ici on réarrange avant d'ajouter, mais on pourrait faire le contraire). Attention aussi à ne pas oublier le `begin ... end`, nécessaire ici.

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

III Le jeu des robots

Q 21. La case de coordonnées $(2, 12)$ n'est entourée d'aucun obstacle. Il est donc impossible pour le robot principal de s'y arrêter sans l'aide des autres robots qui pourraient constituer des obstacles permettant de s'y arrêter.

Q 22. La figure 1 propose une solution en 8 coups.

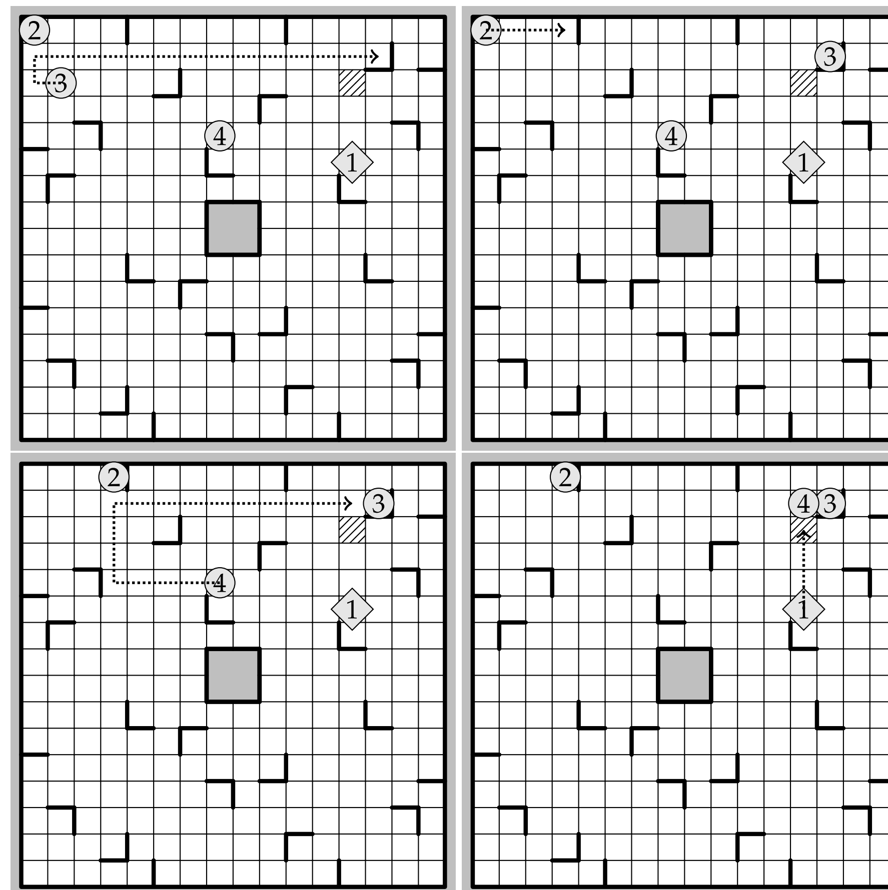


FIGURE 1 – On déplace le robot n° 3 en 3 coups, puis le robot n° 2 en un coup, puis le robot n° 4 en 2 coups et enfin le robot n° 1 en 1 coup, pour une solution optimale en 8 coups.

Q 23. Il y a de très nombreuses variations possibles dans l'écriture de cette fonction.

OCaml

```
let dichotomie a t =
  let debut = ref 0 in
  let fin = ref (Array.length t - 1) in
  (* Invariant : t.(!debut) <= a < t.(!fin) *)
  (* Variant : !fin - !debut *)
  while not (t.(!debut) <= a && a < t.(!debut + 1)) do
    let milieu = (!debut + !fin) / 2 in
    if t.(milieu) <= a then
      debut := milieu
    else
      fin := milieu
  done;
  !debut
```

Q 24. On s'arrête si l'indice `debut` vérifie les conditions demandées. Dans le cas contraire il y a au moins trois éléments dans le sous-tableau considéré et l'appel récursif est effectué sur un sous-tableau de taille strictement inférieure.

OCaml

```
let dichotomie a t =
  (* Invariant : t.(debut) <= a < t.(fin) *)
  let rec cherche_entre debut fin =
    if a < t.(debut + 1) then
      debut
    else
      let milieu = (debut + fin) / 2 in
      if t.(milieu) <= a then
        cherche_entre milieu fin
      else
        cherche_entre debut milieu
  in cherche_entre 0 (Array.length t - 1)
```

Q 25. On cherche dans les tableaux des obstacles les indices des obstacles à l'ouest et au nord à l'aide de la fonction `dichotomie`. Les indices consécutifs donnent alors les obstacles à l'est et au sud. On peut alors déplacer les robots juste avant les obstacles. On fait un nombre constant d'appels à la fonction dichotomie en $O(\log N)$ plus des opérations en temps constant.

OCaml

```
let deplacement_grille (a, b) =
  let gauche = dichotomie b obstacles_lignes.(a) in
  let haut = dichotomie a obstacles_colonnes.(b) in
  [|(a, obstacles_lignes.(a).(gauche));  (* ouest *)
    (a, obstacles_lignes.(a).(gauche + 1) - 1);  (* est *)
    (obstacles_colonnes.(b).(haut), b);  (* nord *)
    (obstacles_colonnes.(b).(haut + 1) - 1, b)|] (* sud *)
```

Q 26. Il suffit de remplir la matrice case par case, pour un coût $O(\log N)$ par case, pour une complexité en $O(N^2 \log N)$ au final.

OCaml

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

Q 27. On regarde si le deuxième robot se trouve sur le chemin du déplacement effectué et mettre à jour la position du robot qui se trouve alors bloqué le cas échéant. On distingue les quatre cas correspondant aux quatre directions. La complexité est bien en $\Theta(1)$.

OCaml

```
let modif t (a, b) (c, d) =
  if (a = c) && (b = d) then
    failwith "Deux robots au même endroit !"
  else if (a = c) && ((snd t.(0)) <= d) && (d < b) then
    t.(0) <- (a, d + 1)
  else if (a = c) && (b < d) && (d <= (snd t.(1))) then
    t.(1) <- (a, d - 1)
  else if (b = d) && ((fst t.(2)) <= c) && (c < a) then
    t.(2) <- (c + 1, b)
  else if (b = d) && (a < c) && (c <= (fst t.(3))) then
    t.(3) <- (c - 1, b)
```

Q 28. Il suffit d'itérer la fonction `modif`, ce que l'on pourrait faire avec `List.iter`.

OCaml
```
let rec iter_modif dep (a, b) robots =
  match robots with
  | [] -> ()
  | (c, d) :: suite ->
      modif dep (a, b) (c, d);
      iter_modif dep (a, b) suite
```

Il faut bien prendre soin de copier la case des déplacements possibles pour ne pas modifier la matrice globale. La complexité est bien linéaire en la taille de la liste `bots` et ne dépend pas de N.

OCaml
```
let deplacements_robots (a, b) bots =
  let dep = Array.copy mat_deplacements.(a).(b) in
  iter_modif dep (a, b) bots;
  dep
```

Q 29. Il faut juste faire attention à l'ordre des autres robots.

OCaml
```
let configuration_initiale = {
  robot = (5, 12);
  autres = [(0, 0); (2, 1); (4, 7)]
}
```

Q 30. On a N^2 choix d'emplacements pour le robot principal. Il reste ensuite à placer les $p-1$ robots sur les N^2-1 cases restantes, ce qui donne donc $N^2\binom{N^2-1}{p-1}$ possibilités. On peut aussi commencer par placer les p robots sur les N^2 cases puis choisir le robot principal (p choix), ce qui donne $p\binom{N^2}{p} = N^2\binom{N^2-1}{p-1}$.

Q 31. Comme indiqué dans l'énoncé, on ne s'attarde pas sur d'éventuels doublons. On commence par ajouter les configurations résultant du mouvement du robot principal. Puis on traite les autres robots avec une fonction auxiliaire `mouv_autres` qui prend comme arguments la liste `fait` des autres robots déjà traités et celle `a_faire` des robots restant à traiter. Afin de conserver la propriété de liste de robots triée on utilise directement la fonction `tri_insertion`.

OCaml
```
let configurations_suivantes c =
  let res = ref [] in
  (* Mouvement du robot principal *)
  let dep = deplacements_robots c.robot c.autres in
  for i = 0 to 3 do
    res := {robot = dep.(i); autres = c.autres} :: !res
  done;
  (* Mouvement des autres robots *)
  let rec mouv_autres fait a_faire =
    match a_faire with
    | [] -> ()
    | un_robot :: suite ->
      (* autres robots sauf un_robot *)
      let reste = fait @ suite in
      let dep =
        deplacements_robots un_robot (c.robot :: reste)
      in
      for i = 0 to 3 do
        let autres = tri_insertion (dep.(i) :: reste) in
        res := {robot = c.robot; autres = autres} :: !res
      done;
      mouv_autres (un_robot :: fait) suite
  in
  mouv_autres [] c.autres;
  !res
```

Q 32. Le calcul de la matrice des déplacements possibles se fait une fois pour toutes en $O(N^2 \log N)$. Déplacer un robot se fait en $O(p)$. À chaque déplacement on a $4 \times p$ choix possibles (on peut déplacer un robot dans une des quatre directions), on a donc au plus $(4p)^k$ séquences de k déplacements à partir de la configuration initiale et donc une complexité en $O(kp(4p)^k + N^2 \log N)$.