

1 Programmation en OCaml

Exercice 1 (Typage) :

1. `val fonction1 : float array -> int -> float = <fun>`
2. `val fonction2 : int list -> int -> int = <fun>`
3. `val fonction3 : (('a -> 'b) -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c = <fun>`

Exercice 2 (Problème de l'arrêt) :

1. `f1` termine sur les entiers pairs. Montrons par récurrence sur k que `f1` termine sur $2k$.
Initialisation : la propriété est vraie pour $k = 0$ (cas de base).
Hérédité : Soit $k > 0$. Supposons que la propriété soit vraie pour $k - 1$. Alors, par HR, `f1(2k - 2)` termine. Donc `f1` termine sur $2k$.
Ainsi, par principe de récurrence, `f1` termine sur tous les entiers pairs.
2. La valeur de i est initialisée à n , et augmente de 1 à chaque passage dans la boucle. De plus, le programme termine lorsque i vaut 10. Ainsi, `f2` termine ssi $n \leq 10$ (et entre dans une boucle infinie sinon).

```

3. 1 let infini n =
    2   while true do
    3     print_string "INFINI !!!\n"
    4   done ;;

```

4. (a) `string -> 'a -> bool.`

(b) 1 `let bizarre code_f x = if arret code_f x then infini 42 ;;`

(c) 1 `let paradoxe code_f = bizarre code_f code_f ;;`

- (d) D'après la propriété demandée sur `paradoxe` à la question précédente, `paradoxe code_paradoxe` termine ssi `paradoxe code_paradoxe` ne termine pas : impossible.

Ainsi, par l'absurde, on a prouvé qu'une telle fonction `arret` n'existe pas.

Exercice 3 (Analyse d'algorithme) :

1. Les valeurs prises par la variable `s` forment une suite strictement croissante à valeurs dans \mathbb{N} , la condition `!s <= n` ne peut donc rester vraie indéfiniment. Ainsi, la boucle `while` (et donc la fonction `f`) termine.
2. Conjecture : `f n = $\lfloor \sqrt{n} \rfloor$` . On prend cette relation comme spécification de `f`.
3. Montrons que la propriété suivante est un invariant de boucle :

$$s = (c + 1)^2 \tag{1}$$

- Avant la boucle, on a $c = 0$ et $s = 1 = (c + 1)^2$.
- Notons c et s les valeurs des variables en haut du corps de la boucle, et c' et s' leurs valeurs en bas du corps de la boucle. Supposons que $s = (c + 1)^2$, et montrons que $s' = (c' + 1)^2$. On a $c' = c + 1$ (ligne 5), et après la ligne 6 on a :

$$s' = \underbrace{s}_{(c+1)^2} + 2c' + 1 = \underbrace{(c+1)^2}_{c'} + 2c' + 1 = c'^2 + 2c' + 1 = (c' + 1)^2$$

Ainsi, la propriété (1) est bien un invariant de boucle.

Lorsqu'on sort de la boucle `while`, on a donc $(c + 1)^2 = s > n$. De plus, la valeur précédente de `s` (qui vaut c^2) respectait la condition du `while`, donc $c^2 \leq n$. Au final, on a $c^2 \leq n < (c + 1)^2$.

Comme $x \mapsto \sqrt{x}$ est croissante, on a : $c \leq \sqrt{n} < c + 1$, donc `f` renvoie $c = \lfloor \sqrt{n} \rfloor$.

La fonction `f` est donc correcte.

4. On passe $c = \lfloor \sqrt{n} \rfloor$ fois dans la boucle while, et on effectue un nombre constant d'opérations élémentaires à chaque passage dans la boucle. On effectue de plus un nombre constant d'opérations élémentaires en dehors de la boucle. La complexité de **f** est donc un $O(\sqrt{n})$.

Exercice 4 (Valeur majoritaire) :

```

1. 1  let occurrences t =
2      (* Renvoie un tableau occ tel que pour tout i
3         occ.(i) est le nombre d'occurrences de t.(i) dans t *)
4      let n = Array.length t in
5      let occ = Array.make n 0 in
6      for i = 0 to n-1 do
7          for j = 0 to n-1 do
8              if t.(i) = t.(j) then
9                  occ.(i) <- occ.(i) + 1
10             done ;
11         done ;
12         occ
13     ;;
14
15     let indice_max t =
16         (* calcule l'indice du max du tableau t *)
17         let n = Array.length t in
18         let imax = ref 0 in (* imax est l'indice du max du tableau *)
19         for i = 1 to n-1 do
20             if t.(i) > t.(!imax) then imax := i
21         done ;
22         !imax
23     ;;
24
25     let valeur_majoritaire t =
26         let imax = indice_max (occurrences t) in
27         t.(imax)
28     ;;

```

2. Toutes les fonctions utilisées ne sont pas récursives et n'utilisent pas de boucle **while**, donc la fonction **valeur_majoritaire** termine.
3. Notons n la longueur de **t**. La fonction **occurrences** parcourt les éléments de **t** à l'aide de deux boucles **for** imbriquées, sa complexité est donc en $O(n^2)$. La fonction **indice_max** a une complexité linéaire, et est utilisée sur un tableau de taille n . Au final, la fonction **valeur_majoritaire** a donc une complexité en $O(n^2)$.

4. fonction **occurrences** :

- Invariant de la boucle externe : $\text{Inv}(i) : \forall k < i \text{ occ.}(k)$ contient le nombre d'occurrences de **t**.(**k**) dans **t**.
- Invariant de la boucle interne : $\text{Inv}'(j) : \text{occ.}(i)$ contient le nombre d'occurrences de **t**.(**i**) parmi **t**.(0), ..., **t**.(**j**-1).

fonction **indice_max** : $\text{Inv}''(i) : \forall k < i, \text{t.}(k) \leq \text{t.}(!\text{imax})$.

5. Une fois le tableau trié, on peut trouver la valeur majoritaire en ne parcourant qu'une seule fois chaque case du tableau, car les valeurs identiques apparaissent désormais consécutivement dans le tableau. Ainsi, bien que la fonction ci-dessous possède deux boucles **while** imbriquées, on ne passe qu'une seule fois par chaque case du tableau, et la complexité de la boucle while externe est donc linéaire. Au final, la solution proposée ci-dessous a une complexité en $O(n \log n)$, ce qui est mieux qu'à la question 1 (l'opération la plus coûteuse est de trier le tableau).

```

1  let valeur_majoritaire' t =
2  let t' = trier t in (* complexite en O(n log n) *)
3    let n = Array.length t' in
4    let i = ref 0 in (* les valeurs de t' entre les indices *)
5    let j = ref 1 in (* i et j seront toutes egales *)
6    let v = ref t'.(0) in (* contiendra la valeur majoritaire *)
7    let occ_max = ref 1 in (* contiendra le nombre d'occurrences de v *)
8    while !i < n do (* complexite en O(n) *)
9      let v_i = t'.(!i) in
10     while !j < n && v_i = t'.(!j) do
11       incr j (* tant que la valeur ne change pas : on avance *)
12     done ;
13     let occ_i = !j - !i in (* nombre d'occurrences de la valeur courante *)
14     if occ_i > !occ_max then
15       begin
16         occ_max := occ_i ;
17         v := v_i
18       end ;
19     (* on passe a la prochaine plage de t' *)
20     i := !j ;
21     j := !i + 1
22   done ;
23   !v
24 ;;
```

Exercice 5 (Recherche dichotomique) :

1. La fonction `dicho` prend en arguments un tableau `t` et un élément `x` comparable aux éléments de `t`, et renvoie un booléen. `val dicho : 'a array -> 'a -> bool = <fun>`
2. Trier le tableau avec la fonction précédente nécessiterait un temps en $O(n \log n)$. La fonction `dicho` est plus efficace : $O(\log n)$, et perdrait alors son intérêt.
3. Supposons que P soit vraie en haut du corps de la boucle, et supposons que x apparaisse dans `t`. Donc x apparaît entre les indices g et d . On a $m = \lfloor \frac{g+d}{2} \rfloor$, et notons $v_m = t.(m)$. Comme le tableau est trié :

- soit $x \leq v_m$, et x se trouve forcément entre g et m ;
- soit $x > v_m$, et x se trouve forcément entre m et d .

Dans tous les cas, P est vraie en bas du corps de la boucle.

4. `dicho [|1;2|] 2 ;;` ne termine pas :
 - au départ, $g = 0$ et $d = 1$;
 - on a donc $m = 0$, et lorsqu'on passe dans la ligne 8, g reste à 0 ;
 - on a donc $g = 0$ et $d = 1$ indéfiniment, et la boucle ne s'arrête jamais.
5. Il suffit de modifier la ligne 8 par : `else g := m + 1`
6. **Terminaison.** Notons u_n la valeur de $d - g$ après n passages dans la boucle. (u_n) est une suite à valeurs dans \mathbb{N} et tend vers 0 (car $u_{n+1} < u_n/2$, on peut donc montrer que $0 \leq u_n \leq \frac{u_0}{2^n}$). Donc à partir d'un certain rang, (u_n) est stationnaire, égale à 0. De plus, la boucle `while` termine lorsque $u_n = 0$. Donc notre fonction termine.

Correction. La propriété P est vraie avant la boucle (car g et d sont alors les indices extrémaux du tableau). De plus, P est toujours préservée à chaque passage dans la boucle malgré notre modification : si $x > v_m$, alors x n'est pas non plus à l'indice m , donc il se trouve forcément entre les indices $m + 1$ et d . Ainsi, à la fin de l'algorithme, on a $g = d$, et P est toujours vérifiée, donc x apparaît dans `t` ssi x apparaît à l'indice g . Le booléen renvoyé est donc correct.

7. (a)

```

1 let tricho t x =
2   let n = Array.length t in
3   let rec aux t g d = match (d=g) with
4     | true -> t.(g) = x
5     | false when t.(g+(d-g)/3) > x -> aux t g (g+((d-g)/3)-1) (* gauche *)
6     | false when t.(g+2*(d-g)/3) < x -> aux t (g+(2*(d-g)/3)+1) d (* droite *)
7     | _ -> aux t (g+(d-g)/3) (g+2*(d-g)/3) (* milieu *)
8   in aux t 0 (n-1)
9 ;;

```

(b) Notons g_k et d_k les valeurs successives de g et d lors des appels récursifs de `aux`, et notons $l_k = d_k - g_k$. On a $l_0 = n - 1$, et $l_{k+1} \leq l_k/3$ (peu importe le cas du pattern matching utilisé). Ainsi, on obtient facilement par récurrence que $\forall k, 0 \leq u_k \leq \frac{n-1}{3^k}$. Donc $u_k \rightarrow 0$, et il existe un rang k_t à partir duquel $u_{k_t} = 0$ (car (u_k) est à valeurs dans \mathbb{N}).

De plus, on a $u_{k_t} = 0$ dès que $\frac{n-1}{3^{k_t}} < 1 \iff n - 1 < 3^{k_t} \iff \log_3(n - 1) < k_t$.

Donc $k_t = \lceil \log_3(n - 1) \rceil = O(\log n)$.

La fonction `tricho` a donc une complexité en $O(\log n)$.

2 Programmation en C

Exercice 6 (Multi-ensembles) :

1. On a : $X(\pi) = 2$; $X(\sqrt{2}) = 3$; $X(e) = 2$; $X(1.5) = 1$, et $X(y) = 0$ pour $y \in \mathbb{R} \setminus \{\pi, \sqrt{2}, e, 1.5\}$.

2. $|X| = 8$.

```

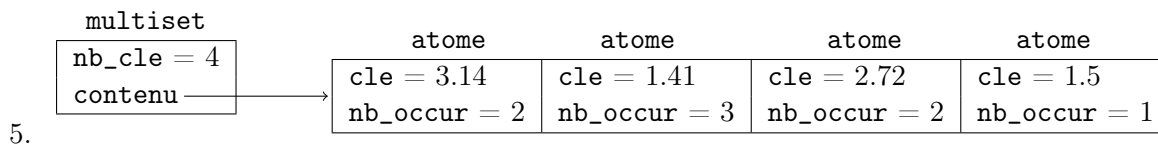
3. 1 struct atome
2   {
3     float cle;
4     int nb_occure;
5   };
6 typedef struct atome atome;

```

```

4. 1 struct multiset
2   {
3     int nb_cle;
4     atome* contenu;
5   };
6 typedef struct multiset multiset;

```



```

6. 1 multiset empty()
2   {
3     multiset e = {0, NULL};
4     return e;
5   }

```

```
7. 1 int cardinal(multiset m)
2   {
3     int card = 0;
4     for (int i = 0; i < m.nb_cle; i++)
5     {
6       card += m.contenu[i].nb_occur;
7     }
8     return card;
9   }
```

```
8. 1 int mem(float cle, multiset m)
2   {
3     for (int i = 0 ; i < m.nb_cle ; i++)
4     {
5       if (m.contenu[i].cle == cle)
6         {return i;}
7     }
8     return -1;
9   }
```

```
9. 1 void ajout_non_present(float cle, int nb_occur, multiset* pm)
2   {
3     atome* contenu_res = (atome*)malloc(sizeof(atome) * (pm->nb_cle+1));
4     for (int i = 0 ; i < pm->nb_cle ; i ++ )
5     {
6       contenu_res[i] = pm->contenu[i];
7     }
8     atome a = {cle, nb_occur};
9     contenu_res[pm->nb_cle] = a;
10    free(pm->contenu);
11    pm->contenu = contenu_res;
12    pm->nb_cle++;
13  }
```

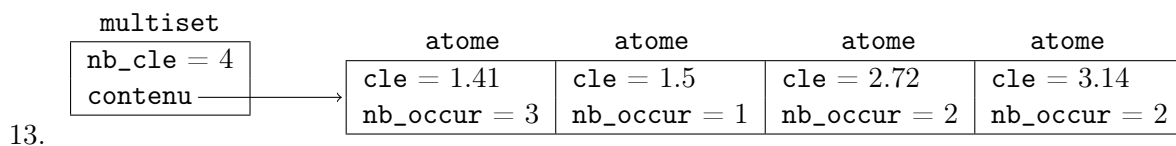
```
10. 1 void ajout(float cle, int nb_occur, multiset* pm)
2   {
3     int x = mem(cle, *pm);
4     if (x >= 0)
5     {
6       pm->contenu[x].nb_occur = pm->contenu[x].nb_occur + nb_occur;
7     }
8     else
9     {
10      ajout_non_present(cle, nb_occur, pm);
11    }
12  }
```

```

11. 1  multiset union1(multiset m1, multiset m2) {
      2      multiset m = empty();
      3      for (int i = 0 ; i < m1.nb_cle ; i ++ ) {
      4          ajout(m1.contenu[i].cle, m1.contenu[i].nb_occur, &m);
      5      }
      6      for (int i = 0 ; i < m2.nb_cle ; i ++ ) {
      7          ajout(m2.contenu[i].cle, m2.contenu[i].nb_occur, &m);
      8      }
      9      return m;
     10 }

```

12. La fonction `ajout` s'effectue en $O(n)$ (une exécution de `mem`, puis éventuellement de `ajout_non_present`), et sera appelée $n = n_1 + n_2$ fois. La fonction `ajout` a donc une complexité en $O(n^2)$.



14. On écrit déjà une fonction comptant le nombre de clés de l'union des deux multi-ensembles (s'ils ont des clés en commun, il faut ne pas les compter deux fois).

Puisque les tableaux sont triés selon les clés, il suffit d'avancer en parallèle dans les deux tableaux pour avoir une complexité linéaire (on a un curseur par multi-ensemble, et on avance d'un cran au niveau de la plus petite des deux clés jusqu'à arriver au bout des deux tableaux).

Code : cf. pages suivantes.

15. En utilisant le fait que nos tableaux sont triés, on a donc une boucle `while` dans laquelle on passera $n = n_1 + n_2$ fois (pour la fonction `nombre_de_cles` et pour la fonction `union2`). La fonction `union2` a donc une complexité en $O(n)$.

```
1 int nombre_de_cles(multiset m1, multiset m2)
2 {
3     int total = 0;
4     int i1 = 0; int i2 = 0;
5     while (i1 < m1.nb_cle || i2 < m2.nb_cle)
6     {
7         if (i1 < m1.nb_cle && i2 < m2.nb_cle)
8         {
9             if (m1.contenu[i1].cle < m2.contenu[i2].cle)
10            {
11                i1++;
12                total++;
13            }
14            else if (m1.contenu[i1].cle > m2.contenu[i2].cle)
15            {
16                i2++;
17                total++;
18            }
19            else
20            {
21                i1++;
22                i2++;
23                total++;
24            }
25        }
26        else if (i1 < m1.nb_cle)
27        {
28            i1++;
29            total++;
30        }
31        else
32        {
33            i2++;
34            total++;
35        }
36    }
37    return total;
38 }
```

```
1 multiset union2(multiset m1, multiset m2)
2 {
3     int sc = nombre_de_cles(m1, m2);
4     atome *rescont = (atome*)malloc(sizeof(atome) * sc);
5     multiset res = { sc, rescont };
6     int i1 = 0; int i2 = 0; int i = 0;
7     while (i1 < m1.nb_cle || i2 < m2.nb_cle)
8     {
9         if (i1 < m1.nb_cle && i2 < m2.nb_cle)
10        {
11            if (m1.contenu[i1].cle < m2.contenu[i2].cle)
12            {
13                rescont[i] = m1.contenu[i1];
14                i++;
15                i1++;
16            }
17            else if (m1.contenu[i1].cle > m2.contenu[i2].cle)
18            {
19                rescont[i] = m2.contenu[i2];
20                i++;
21                i2++;
22            }
23            else
24            {
25                rescont[i] = m1.contenu[i1];
26                rescont[i].nb_occur += m2.contenu[i2].nb_occur;
27                i++;
28                i1++;
29                i2++;
30            }
31        }
32        else if (i1 < m1.nb_cle)
33        {
34            rescont[i] = m1.contenu[i1];
35            i1++;
36            i++;
37        }
38        else
39        {
40            rescont[i] = m2.contenu[i2];
41            i2++;
42            i++;
43        }
44    }
45    return res;
46 }
```