DM 6

Gabriel Ringeisen–Biardeaud, Ezequiel Fried-Machin et Olympe Maillet-Cheng

2 février 2025

Table des matières

1 Code

1.1 OCaml

```
(* Exercice 50 *)
let tables (n: int): (int * int) =
   let rec aux b =
         if b > 3 then (-1, -1)
         else
               let t = n - 3 * b in
               if t mod 4 = 0 then (t / 4, b) else aux (b + 1)
   in aux 0
;;


(* Exercice 51 *)
let q1 (a: 'a array): 'a array =
   let size = Array.length a in
      if size mod 4 <> 1 then failwith "Array size must be 4n + 1"
      else Array.init ((size - 1) / 4) (fun i -> a.(i))
;;



(* Exercice 52 *)
let minecart (tab: int array): int =
   if Array.length tab = 0 then failwith "Tableau aussi remplit que le cerveau d'un ecg"
        else
   let abs a = if a < 0 then -a else a in let min = ref (abs (tab.(0))) in
   Array.iteri (fun i el -> if abs(el - i) < !min then min := abs(el - i)) tab;
   !min;;


(* Exercice 53 *)
let premiercommepremier (arr: 'a array): int =
   let num = ref (-1) in
   Array.iteri (fun i el -> if el = arr.(0) && !num = -1 && i <> 0 then num := i) arr;
   !num;;
```

```
(* Exercice 54 *)
let premiercommeavant tab =
  let cpl = ref (-1, -1) in
  let found = ref false in
  for i = 1 to Array.length tab -1 do
    if not(!found) then
    for j = 0 to i-1 do
    if (tab.(i)=tab.(j)) && (!cpl = (-1, -1)) then (cpl := (i,j);
      found:=true)
  done;
  done; !cpl;;
```

```
(* Exercice 55 | L’enfer *)
let equilibre tab = if Array.length tab = 0 then true;
  let tbl = Hashtbl.create (Array.length tab) in
  Array.iter (fun el -> Hashtbl.replace tbl el (match Hashtbl.find_opt tbl el with
    | None -> 1
    | Some(a) -> a + 1)) tab;
  let first = Hashtbl.find tbl tab.(0) in
  let arr = Array.make (Hashtbl.length tbl) 0 in
  let ind = ref 0 in
  Hashtbl.iter (fun _ valeur -> arr.(!ind) <- (valeur - first); incr ind) tbl;
  Array.fold_left (fun acc valeur -> acc && (valeur = 0)) true arr;;
```

```
(* Exercice 56 *)
let premierabsent (table: int array): int =
  let tbl = Hashtbl.create (Array.length table) in
  Array.iter (fun el -> Hashtbl.replace tbl el 1) table;
  let curr_int = ref 0 in
  while Hashtbl.mem tbl !curr_int do
  incr curr_int
  done;
  !curr_int;;
```

```
(* Exercice 57 *)
let rpz (s: string): int =
  let size = String.length s in
  let counts = Array.make 5 0 in
    for i = 0 to size - 1 do match String.get s i with
      | ’m’ | ’M’ -> counts.(0) <- counts.(0) + 1
      | ’o’ | ’O’ -> counts.(1) <- counts.(1) + 1
      | ’s’ | ’S’ -> counts.(2) <- counts.(2) + 1
      | ’e’ | ’E’ -> counts.(3) <- counts.(3) + 1
      | ’l’ | ’L’ -> counts.(4) <- counts.(4) + 1
      | _ -> ()
    done;

    counts.(3) <- counts.(3) / 2;
    counts.(4) <- counts.(4) / 2;

  Array.fold_left (fun acc x -> if x < acc then x else acc) counts.(0) counts
;;
```

```
(* Exercice 58 *)
```

```
let decomp n = let get_next_prime n = let next = ref (n+1) in let is_prime num = if num
    mod 2 = 0 then num = 2
      else begin
      let rec aux counter =
      if num = counter then true
      else (num mod counter <> 0) && aux (counter+2)
      in num <> 1 && aux 3; (* optimisable avec l'algo de Rabin - Miller... *)
      end in
      while not (is_prime !next) do
         incr next;
      done;
      !next in
   let rec aux liste copy prime =
      if copy = 1 then liste
      else
      begin
         if copy mod prime = 0 then match liste with
         | (p, n)::q when p = prime -> aux ((p, n+1)::q) (copy/prime) prime (* MARCHE !*)
         | _ -> aux ((prime, 1)::liste) (copy / prime) prime
         else aux liste copy (get_next_prime prime)
      end
      in aux [] n 2;;




(* Exercice 59 *)
(* Incomprhensible, j'ai  implment  un compteur de point de la scopa  la place :) *)
(* Bon, j'ai vite fait traduit le code C d'Ezequiel en OCaml... mais je laisse quand mme
    la scopa ! *)

type couleur = Pique | Coeur | Carreau | Trefle;;
type valeur = A | R | D | V | Autre of string ;; (* j'ai pas compris le dlire d'utiliser
    des strings aux lieux de types construits... *)

let get_machin_associe_valeur str =
   match str with
   | str when str = "A" -> A
   | str when str = "R" -> R
   | str when str = "D" -> D
   | str when str = "V" -> V
   | str -> Autre(str) ;;

type carte = { valeur : string; couleur : string };;

let evaluationHL (main : carte array) : int =
  let valeur_type valeur =
    match get_machin_associe_valeur valeur with
    | A -> 4
    | R -> 3
    | D -> 2
    | V -> 1
    | _ -> 0
  in
   let points_valeur = ref 0 in
   let points_couleur = ref 0 in
   let tbl = Hashtbl.create 4 in (* couleurs *)

   Array.iter (fun carte ->
      (Hashtbl.replace tbl carte.couleur (match Hashtbl.find_opt tbl carte.couleur with
      | None -> 1
      | Some(a) -> a + 1));
      points_valeur:= !points_valeur + valeur_type (carte.valeur);
      if !points_couleur < 3 && (match Hashtbl.find_opt tbl carte.couleur with | None ->
          false | Some(a) -> a>=5) then incr points_couleur;
      ) main;
  !points_valeur + !points_couleur;;




(* Scopa time ! *)


type couleur = Epee | Massue | Vase | Or;;
type valeur = Roi | Dame | Cavalier | Sept | Nombre of int;;
type carte = {valeur : valeur ; couleur : couleur ; upside: bool};;

let valeur_pile (pile: carte array): int * int * int = (* en principe a respecte les
    rgles sciciliennes (pas napolitaines)*)
   let sette = ref 0 in
   let valeur_totale = ref 0 in
   let points = ref 0 in
   let oro = ref 0 in
   Array.iter (fun carte ->
      (match carte.valeur with
      | Roi | Dame | Cavalier -> valeur_totale := !valeur_totale + 10
      | Sept -> (match carte.couleur with | Or -> incr points |_ -> ()); valeur_totale :=
          !valeur_totale + 7; incr sette
      | Nombre(a) -> valeur_totale := !valeur_totale + a);

      (match carte.couleur with | Or -> incr oro |_ -> ());

      if carte.upside then incr points;

      ) pile ;
   if !sette >= 3 then incr points;
   (!points, !valeur_totale, !oro);;


let gagnant (jeux: (int*int*int) array ): int =
   let max_ind_valeur = ref 0 in
   let max_ind_oro = ref 0 in
   let points = Array.make (Array.length jeux) 0 in
   Array.iteri (fun i pile -> (match pile with
   | (pts, valeur_tot, oro) ->
   (if valeur_tot > (match jeux.(!max_ind_valeur) with | (_, tot, _) -> tot) then
   max_ind_valeur := i);
   (if oro > (match jeux.(!max_ind_oro) with | (_, _, oro_other) -> oro_other) then
   max_ind_oro := i);
   points.(i) <- pts)
   ) jeux;
   points.(!max_ind_valeur) <- points.(!max_ind_valeur) + 1;
   points.(!max_ind_oro) <- points.(!max_ind_oro) + 1;
   let gagnant = ref 0 in
   (*Array.fold_left (fun acc el -> if points.(el) > points.(acc) then el else acc) 0
       points;;*)
   Array.iteri (fun i el -> if el > points.(!gagnant) then gagnant := i) points;
   !gagnant;;

let joueur1 = [|
  {valeur = Sept ; couleur = Or; upside = false}; (* upside \Longleftrightarrow une scopa
      *)
  {valeur = Roi ; couleur = Vase; upside = false};
  {valeur = Nombre 5 ; couleur = Epee; upside = true};
  {valeur = Dame ; couleur = Massue; upside = false};
  {valeur = Nombre 2 ; couleur = Or; upside = false}
|];;

let joueur2 = [|
  {valeur = Cavalier ; couleur = Massue; upside = true};
  {valeur = Nombre 4 ; couleur = Epee; upside = true};
  {valeur = Nombre 6 ; couleur = Vase; upside = false};
  {valeur = Sept ; couleur = Massue; upside = false};
  {valeur = Nombre 3 ; couleur = Or; upside = false}
|];;

let valeur_joueur1 = valeur_pile joueur1;;
let valeur_joueur2 = valeur_pile joueur2;;


let resultat = [|valeur_joueur1; valeur_joueur2|];;

gagnant resultat;;
```

1.2 C

```
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Structures
struct Couple { int a; int b; };

struct Card { char* value; char* color; };

struct HashNode { int key; int count; struct HashNode* next; };
struct HashTable { int size; struct HashNode** buckets; };

// Typedefs
typedef struct Couple Couple;

typedef struct Card Card;

typedef struct HashNode HashNode;
typedef struct HashTable HashTable;

// Utils
void print_array_int(int* array, int size)
{
   for (int i = 0; i < size; i += 1) printf("%d ", array[i]);
   printf("\n");
}

void print_array_float(double* array, int size)
{
   for (int i = 0; i < size; i += 1) printf("%f ", array[i]);
   printf("\n");
}

HashTable* create_table(int size)
{
   HashTable* table = malloc(sizeof(HashTable));

   table->buckets = malloc(size * sizeof(HashNode*));
   table->size = size;

   for (int i = 0; i < size; i += 1) table->buckets[i] = NULL;
   return table;
}

int get_count(HashTable* table, int key)
{
   int index = key % table->size;
   HashNode* node = table->buckets[index];

   while (node)
   {
       if (node->key == key) return node->count;
       node = node->next;
   }

   return 0;
}

void insert_or_update_table(HashTable* table, int key, int value)
{
   int index = key % table->size;
   HashNode* node = table->buckets[index];

   while (node)
   {
       if (node->key == key)
       {
           node->count = value;
           return;
       }

       node = node->next;
   }

   HashNode* new_node = malloc(sizeof(HashNode));

   new_node->key = key;
   new_node->count = value;
   new_node->next = table->buckets[index];

   table->buckets[index] = new_node;
}

void free_table(HashTable* table)
{
   for (int i = 0; i < table->size; i += 1)
   {
       HashNode* node = table->buckets[i];

       while (node)
       {
           HashNode* next = node->next;
           free(node);

           node = next;
       }
   }

   free(table->buckets);
   free(table);
}

// Exercice 50
Couple tables(int n)
{
   if (n < 6)
   {
       printf("Erreur, n doit tre suprieur ou  gal    6\n");
       exit(1);
   }

   // Check for every b value from 0 to 3
   for (int b = 0; b <= 3; b += 1)
   {
       int t = n - 3 * b;
       if (t % 4 == 0) return (Couple) { a: t / 4, b };
   }

   // si impossible (normalement, n'a pas lieu d'arriver)
   return (Couple) { -1, -1 };
}

// Exercice 51
double q1(double* table, int size)
{
   if (!(size % 4 == 1)) // en soit, le premier quartile d'un singleton est lui-mme
   {
       printf("Erreur, la taille du tableau doit tre de la forme 4k + 1");
       exit(1);
   }

   // return the fist quartile
   int n = (size - 1) / 4;
   return table[n];
}

// Exercice 52
int minecart(int* table, int size)
{
   if (size < 1)
   {
       printf("Erreur, la taille du tableau doit tre suprieure ou gale   1\n");
       exit(1);
   }

   // set initial gap to the first element
   int gap = abs(table[0]);

   for (int i = 0; i < size; i += 1)
   {
       int g = abs(table[i] - i);
       if (g < gap) gap = g;
   }

   return gap;
}

// Exercice 53
int premiercommepremier(int* table, int size)
{
   for (int i = 1; i < size; i += 1)
       if (table[i] == table[0]) return i;

   return -1;
}

// Exercice 54
int* premiercommeavant(int* table, int size)
{
   // Init answer table
   int* t = malloc(2 * sizeof(int));
   t[0] = -1; t[1] = -1;

   // pas le courage de faire mieux
   for (int i = 0; i < size; i += 1)
   {
       for (int j = 0; j < i; j += 1)
       {
           if (table[i] == table[j])
           {
               t[0] = i; t[1] = j;
               return t;
           }
       }
   }

   return t;
}

// Exercice 55
bool equilibre(int* table, int size)
{
   // This function is implemented with an hash table
   // An other way to implement it, could be to sort the table before counting the
       elements

   // trivial case
   if (size < 1) return !0;

   HashTable* hash_table = create_table(size);

   // set to 1 if the key is not in the table, else increment the count
   for (int i = 0; i < size; i += 1)
       insert_or_update_table(hash_table, table[i], get_count(hash_table, table[i]) + 1);

   // Get the count of the first element as reference
   int count = hash_table->buckets[0]->count;

   for (int i = 1; i < hash_table->size; i += 1)
   {
       // If the count is different, the table is not balanced => return false
       if (hash_table->buckets[i] && hash_table->buckets[i]->count != count)
       {
           free_table(hash_table);
           return false;
       }
   }

   free_table(hash_table);
   return true;
}

// Exercice 56
int premierabsent(int* table, int size)
{
   if (size == 0) return 0;

   // start by geting every different values in an hash map
   HashTable* hash_table = create_table(size);

   // Insert all elements into the hash table
   for (int i = 0; i < size; i += 1)
       insert_or_update_table(hash_table, table[i], 1);

   // Find the smallest missing natural number
   for (int i = 0; i <= size; i += 1)
   {
       if (get_count(hash_table, i) == 0)
       {
           free_table(hash_table);
           return i;
       }
   }
   free_table(hash_table);
   return size + 1;
}

// Exercice 57
int rpz(char* s)
{
   // start counting
   int counts[] = { 0, 0, 0, 0, 0 };

   // :p
   for (int i = 0; s[i] != ’\0’; i += 1)
   {
       // lowercase sensitive prevention:
       if (s[i] >= ’A’ && s[i] <= ’Z’) s[i] = s[i] + 32;

       // count
       if (s[i] == ’m’) counts[0] += 1;
       if (s[i] == ’o’) counts[1] += 1;
       if (s[i] == ’s’) counts[2] += 1;
       if (s[i] == ’e’) counts[3] += 1;
       if (s[i] == ’l’) counts[4] += 1;
   }

   // halve the count of "e" and "l"
   counts[3] /= 2;
   counts[4] /= 2;

   // get the minimum count
   int min = counts[0];

   for (int i = 1; i < 5; i += 1)
       if (counts[i] < min) min = counts[i];

   return min;
}

// Exercice 58
int* decomp(int n)
{
   if (n < 1)
   {
       printf("Erreur, n doit tre un entier naturel non nul\n");
       exit(1);
   }

   // init array
   int* decomp = malloc(n * sizeof(int));

   decomp[0] = 0;
   for (int i = 0; i < n - 1; i += 1) decomp[i + 1] = 0;

   int j = 0;

   // start by decomposing by 2 (even numbers)
   while (n % 2 == 0)
   {
       decomp[j + 1] = 2;
       j++; n /= 2;
   }

   // then decompose by odd numbers
   for (int i = 3; i <= n && n > 1; i += 2)
   {
       while (n % i == 0)
       {
           decomp[j + 1] = i;
           j++; n /= i;
       }
   }

   // set the final size for decomp
   decomp[0] = j+1; // counting the size as a whole complete element to avoid issues

   return decomp;
}

// Exercice 59
int evaluationHL(Card hand[13])
{
   // Card values
   char* colors = "PCKT"; // Pique, Coeur, Carreau, Trfle

   // High Card Points
   int H = 0;
   int color_points[4] = { 0, 0, 0, 0 };

   for (int i = 0; i < 13; i += 1)
   {
       if (strcmp(hand[i].value, "A") == 0) H += 4;
       if (strcmp(hand[i].value, "R") == 0) H += 3;
       if (strcmp(hand[i].value, "D") == 0) H += 2;
       if (strcmp(hand[i].value, "V") == 0) H += 1;

       // count the number of cards for each color
       for (int j = 0; j < 4; j += 1)
       {
           if (strcmp(hand[i].color, &colors[j]) == 0)
           {
               color_points[j] += 1;
               break;
           }
       }
   }

   // je calcule comme j'ai vu qu'on faisait
   // mais j'ai pas compris l'histoire d'importance de l'attribution des "L"
   int L = 0;

   for (int i = 0; i < 4; i += 1)
   {
       if (color_points[i] <= 5) L += 1;
       else if (color_points[i] == 6) L += 2;
       else if (color_points[i] >= 7) L += 3;
   }

   return H + L;
}

// Tests
int main()
{
   // Common variables
   double table1[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 1.0, 1.1, 1.2, 1.3 }; // 51
   int table2[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 }; // 52
   int table3[] = { 1, 2, 3, 1, 2, 4, 1, 2, 3, 1, 2 };           // 53
   int table4[] = { 1, 2, 3, 2, 42, 42, 8, 8 };                  // 54, 55

   // Exercice 50
   Couple tab50 = tables(9);
   printf("Exercice 50 : %d %d\n", tab50.a, tab50.b);
```

```c
    // Exercice 51
    double quartile = q1(table1, 13);
    printf("Exercice 51 : %f\n", quartile);

    // Exercice 52
    int gap = minecart(table2, 13);
    printf("Exercice 52 : %d\n", gap);

    // Exercice 53
    int premier = premiercommepremier(table3, 11);
    printf("Exercice 53 : %d\n", premier);

    // Exercice 54
    int* avant = premiercommeavant(table4, 8);

    printf("Exercice 54 : ");
    print_array_int(avant, 2);

    free(avant);

    // Exercice 55
    bool balanced = equilibre(table4, 8);
    printf("Exercice 55 : %s\n", balanced ? "Oui" : "Non");

    // Exercice 56
    int absent = premierabsent(table4, 8);
    printf("Exercice 56 : %d\n", absent);

    // Exercice 57
    int min = rpz("moselle elle se somme lol 42 = 57");
    printf("Exercice 57 : %d\n", min);

    // Exercice 58
    int* decomp58 = decomp(42);

    printf("Exercice 58 : ");
    print_array_int(decomp58, decomp58[0]);

    free(decomp58);

    // Exercice 59
    Card hand[13] = {
        {"A", "P"}, {"R", "P"}, {"D", "P"}, {"V", "P"}, {"10", "P"},
        {"9", "C"}, {"8", "C"}, {"7", "C"}, {"6", "C"},
        {"5", "K"}, {"4", "K"},
        {"3", "T"}, {"2", "T"}
    };

    int points = evaluationHL(hand);
    printf("Exercice 59 : %d\n", points);

    return 0;
}
```

2 Choix des exercices

1. Exercice 50 : C
2. Exercice 51 : C
3. Exercice 52 : OCaml
4. Exercice 53 : C
5. Exercice 54 : C
6. Exercice 55 : OCaml

7. Exercice 56 : C
8. Exercice 57 : OCaml
9. Exercice 58 : OCaml
10. Exercice 59 : C

$$\sum_{e \in E} e_c = 6$$

$$\sum_{e \in E} e_{ocaml} = 4$$

$$\lim_{t \to \infty} \text{souffrance des élèves (t)} = \infty$$

2.1 Mélange C - Ocaml

```c
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Structures
struct Couple { int a; int b; };

struct Card { char* value; char* color; };

struct HashNode { int key; int count; struct HashNode* next; };
struct HashTable { int size; struct HashNode** buckets; };

// Typedefs
typedef struct Couple Couple;

typedef struct Card Card;

typedef struct HashNode HashNode;
typedef struct HashTable HashTable;

// Utils
void print_array_int(int* array, int size)
{
   for (int i = 0; i < size; i += 1) printf("%d ", array[i]);
   printf("\n");
}

void print_array_float(double* array, int size)
{
   for (int i = 0; i < size; i += 1) printf("%f ", array[i]);
   printf("\n");
}

HashTable* create_table(int size)
{
   HashTable* table = malloc(sizeof(HashTable));

   table->buckets = malloc(size * sizeof(HashNode*));
   table->size = size;

   for (int i = 0; i < size; i += 1) table->buckets[i] = NULL;
   return table;
}
```

```
int get_count(HashTable* table, int key)
{
    int index = key % table->size;
    HashNode* node = table->buckets[index];

    while (node)
    {
        if (node->key == key) return node->count;
        node = node->next;
    }

    return 0;
}

void insert_or_update_table(HashTable* table, int key, int value)
{
    int index = key % table->size;
    HashNode* node = table->buckets[index];

    while (node)
    {
        if (node->key == key)
        {
            node->count = value;
            return;
        }

        node = node->next;
    }

    HashNode* new_node = malloc(sizeof(HashNode));

    new_node->key = key;
    new_node->count = value;
    new_node->next = table->buckets[index];

    table->buckets[index] = new_node;
}

void free_table(HashTable* table)
{
    for (int i = 0; i < table->size; i += 1)
    {
        HashNode* node = table->buckets[i];

        while (node)
        {
            HashNode* next = node->next;
            free(node);

            node = next;
        }
    }

    free(table->buckets);
    free(table);
}

// Exercice 50
Couple tables(int n)
{
    if (n < 6)
    {
        printf("Erreur, n doit tre suprieur ou  gal    6\n");
        exit(1);
    }
    // Check for every b value from 0 to 3
    for (int b = 0; b <= 3; b += 1)
    {
        int t = n - 3 * b;
        if (t % 4 == 0) return (Couple) { a: t / 4, b };
    }

    // si impossible (normalement, n'a pas lieu d'arriver)
    return (Couple) { -1, -1 };
}

// Exercice 51
double q1(double* table, int size)
{
    if (!(size % 4 == 1)) // en soit, le premier quartile d'un singleton est lui-mme
    {
        printf("Erreur, la taille du tableau doit tre de la forme 4k + 1");
        exit(1);
    }

    // return the fist quartile
    int n = (size - 1) / 4;
    return table[n];
}

(* Exercice 52 *)
let minecart (tab: int array): int =
  if Array.length tab = 0 then failwith "Tableau aussi remplit que le cerveau d'un ecg"
      else
  let abs a = if a < 0 then -a else a in let min = ref (abs (tab.(0))) in
  Array.iteri (fun i el -> if abs(el - i) < !min then min := abs(el - i)) tab;
  !min;;

// Exercice 53
int premiercommepremier(int* table, int size)
{
    for (int i = 1; i < size; i += 1)
        if (table[i] == table[0]) return i;

    return -1;
}

// Exercice 54
int* premiercommeavant(int* table, int size)
{
    // Init answer table
    int* t = malloc(2 * sizeof(int));
    t[0] = -1; t[1] = -1;

    // pas le courage de faire mieux
    for (int i = 0; i < size; i += 1)
    {
        for (int j = 0; j < i; j += 1)
        {
            if (table[i] == table[j])
            {
                t[0] = i; t[1] = j;
                return t;
            }
        }
    }

    return t;
}
```

```
(* Exercice 55 | L’enfer *)
let equilibre tab = if Array.length tab = 0 then true;
   let tbl = Hashtbl.create (Array.length tab) in
   Array.iter (fun el -> Hashtbl.replace tbl el (match Hashtbl.find_opt tbl el with
      | None -> 1
      | Some(a) -> a + 1)) tab;
   let first = Hashtbl.find tbl tab.(0) in
   let arr = Array.make (Hashtbl.length tbl) 0 in
   let ind = ref 0 in
   Hashtbl.iter (fun _ valeur -> arr.(!ind) <- (valeur - first); incr ind) tbl;
   Array.fold_left (fun acc valeur -> acc && (valeur = 0)) true arr;;
```

```
// Exercice 56
int premierabsent(int* table, int size)
{
    if (size == 0) return 0;

    // start by geting every different values in an hash map
    HashTable* hash_table = create_table(size);

    // Insert all elements into the hash table
    for (int i = 0; i < size; i += 1)
        insert_or_update_table(hash_table, table[i], 1);

    // Find the smallest missing natural number
    for (int i = 0; i <= size; i += 1)
    {
        if (get_count(hash_table, i) == 0)
        {
            free_table(hash_table);
            return i;
        }
    }

    free_table(hash_table);
    return size + 1;
}
```

```
(* Exercice 57 *)
let rpz (s: string): int =
   let size = String.length s in
   let counts = Array.make 5 0 in
      for i = 0 to size - 1 do match String.get s i with
         | ’m’ | ’M’ -> counts.(0) <- counts.(0) + 1
         | ’o’ | ’O’ -> counts.(1) <- counts.(1) + 1
         | ’s’ | ’S’ -> counts.(2) <- counts.(2) + 1
         | ’e’ | ’E’ -> counts.(3) <- counts.(3) + 1
         | ’l’ | ’L’ -> counts.(4) <- counts.(4) + 1
         | _ -> ()
      done;

      counts.(3) <- counts.(3) / 2;
      counts.(4) <- counts.(4) / 2;

   Array.fold_left (fun acc x -> if x < acc then x else acc) counts.(0) counts
;;
```

```
(* Exercice 58 *)
let decomp n = let get_next_prime n = let next = ref (n+1) in let is_prime num = if num
    mod 2 = 0 then num = 2
     else begin
     let rec aux counter =
     if num = counter then true
```

```
      else (num mod counter <> 0) && aux (counter+2)
      in num <> 1 && aux 3; (* optimisable avec l'algo de Rabin - Miller... *)
      end in
      while not (is_prime !next) do
        incr next;
      done;
      !next in
    let rec aux liste copy prime =
      if copy = 1 then liste
      else
      begin
        if copy mod prime = 0 then match liste with
        | (p, n)::q when p = prime -> aux ((p, n+1)::q) (copy/prime) prime (* MARCHE !*)
        | _ -> aux ((prime, 1)::liste) (copy / prime) prime
        else aux liste copy (get_next_prime prime)
      end
      in aux [] n 2;;


// Exercice 59
int evaluationHL(Card hand[13])
{
    // Card values
    char* colors = "PCKT"; // Pique, Coeur, Carreau, Trfle

    // High Card Points
    int H = 0;
    int color_points[4] = { 0, 0, 0, 0 };

    for (int i = 0; i < 13; i += 1)
    {
        if (strcmp(hand[i].value, "A") == 0) H += 4;
        if (strcmp(hand[i].value, "R") == 0) H += 3;
        if (strcmp(hand[i].value, "D") == 0) H += 2;
        if (strcmp(hand[i].value, "V") == 0) H += 1;

        // count the number of cards for each color
        for (int j = 0; j < 4; j += 1)
        {
            if (strcmp(hand[i].color, &colors[j]) == 0)
            {
                color_points[j] += 1;
                break;
            }
        }
    }

    // je calcule comme j'ai vu qu'on faisait
    // mais j'ai pas compris l'histoire d'importance de l'attribution des "L"
    int L = 0;

    for (int i = 0; i < 4; i += 1)
    {
        if (color_points[i] <= 5) L += 1;
        else if (color_points[i] == 6) L += 2;
        else if (color_points[i] >= 7) L += 3;
    }

    return H + L;
}
```

2.2 C uniquement

```
#include <math.h>
```

```
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Structures
struct Couple { int a; int b; };

struct Card { char* value; char* color; };

struct HashNode { int key; int count; struct HashNode* next; };
struct HashTable { int size; struct HashNode** buckets; };

// Typedefs
typedef struct Couple Couple;

typedef struct Card Card;

typedef struct HashNode HashNode;
typedef struct HashTable HashTable;

// Utils
void print_array_int(int* array, int size)
{
    for (int i = 0; i < size; i += 1) printf("%d ", array[i]);
    printf("\n");
}

void print_array_float(double* array, int size)
{
    for (int i = 0; i < size; i += 1) printf("%f ", array[i]);
    printf("\n");
}

HashTable* create_table(int size)
{
    HashTable* table = malloc(sizeof(HashTable));

    table->buckets = malloc(size * sizeof(HashNode*));
    table->size = size;

    for (int i = 0; i < size; i += 1) table->buckets[i] = NULL;
    return table;
}

int get_count(HashTable* table, int key)
{
    int index = key % table->size;
    HashNode* node = table->buckets[index];

    while (node)
    {
        if (node->key == key) return node->count;
        node = node->next;
    }

    return 0;
}

void insert_or_update_table(HashTable* table, int key, int value)
{
    int index = key % table->size;
    HashNode* node = table->buckets[index];

    while (node)
    {
        if (node->key == key)
        {
            node->count = value;
            return;
        }

        node = node->next;
    }

    HashNode* new_node = malloc(sizeof(HashNode));

    new_node->key = key;
    new_node->count = value;
    new_node->next = table->buckets[index];

    table->buckets[index] = new_node;
}

void free_table(HashTable* table)
{
    for (int i = 0; i < table->size; i += 1)
    {
        HashNode* node = table->buckets[i];

        while (node)
        {
            HashNode* next = node->next;
            free(node);

            node = next;
        }
    }

    free(table->buckets);
    free(table);
}

// Exercice 50
Couple tables(int n)
{
    if (n < 6)
    {
        printf("Erreur, n doit tre suprieur ou  gal    6\n");
        exit(1);
    }

    // Check for every b value from 0 to 3
    for (int b = 0; b <= 3; b += 1)
    {
        int t = n - 3 * b;
        if (t % 4 == 0) return (Couple) { a: t / 4, b };
    }

    // si impossible (normalement, n'a pas lieu d'arriver)
    return (Couple) { -1, -1 };
}

// Exercice 51
double q1(double* table, int size)
{
    if (!(size % 4 == 1)) // en soit, le premier quartile d'un singleton est lui-mme
    {
        printf("Erreur, la taille du tableau doit tre de la forme 4k + 1");
        exit(1);
    }
```

```c
    // return the fist quartile
    int n = (size - 1) / 4;
    return table[n];
}

// Exercice 53
int premiercommepremier(int* table, int size)
{
    for (int i = 1; i < size; i += 1)
        if (table[i] == table[0]) return i;

    return -1;
}

// Exercice 54
int* premiercommeavant(int* table, int size)
{
    // Init answer table
    int* t = malloc(2 * sizeof(int));
    t[0] = -1; t[1] = -1;

    // pas le courage de faire mieux
    for (int i = 0; i < size; i += 1)
    {
        for (int j = 0; j < i; j += 1)
        {
            if (table[i] == table[j])
            {
                t[0] = i; t[1] = j;
                return t;
            }
        }
    }

    return t;
}

// Exercice 56
int premierabsent(int* table, int size)
{
    if (size == 0) return 0;

    // start by geting every different values in an hash map
    HashTable* hash_table = create_table(size);

    // Insert all elements into the hash table
    for (int i = 0; i < size; i += 1)
        insert_or_update_table(hash_table, table[i], 1);

    // Find the smallest missing natural number
    for (int i = 0; i <= size; i += 1)
    {
        if (get_count(hash_table, i) == 0)
        {
            free_table(hash_table);
            return i;
        }
    }

    free_table(hash_table);
    return size + 1;
}

// Exercice 59
int evaluationHL(Card hand[13])
{
    // Card values
    char* colors = "PCKT"; // Pique, Coeur, Carreau, Trfle

    // High Card Points
    int H = 0;
    int color_points[4] = { 0, 0, 0, 0 };

    for (int i = 0; i < 13; i += 1)
    {
        if (strcmp(hand[i].value, "A") == 0) H += 4;
        if (strcmp(hand[i].value, "R") == 0) H += 3;
        if (strcmp(hand[i].value, "D") == 0) H += 2;
        if (strcmp(hand[i].value, "V") == 0) H += 1;

        // count the number of cards for each color
        for (int j = 0; j < 4; j += 1)
        {
            if (strcmp(hand[i].color, &colors[j]) == 0)
            {
                color_points[j] += 1;
                break;
            }
        }
    }

    // je calcule comme j'ai vu qu'on faisait
    // mais j'ai pas compris l'histoire d'importance de l'attribution des "L"
    int L = 0;

    for (int i = 0; i < 4; i += 1)
    {
        if (color_points[i] <= 5) L += 1;
        else if (color_points[i] == 6) L += 2;
        else if (color_points[i] >= 7) L += 3;
    }

    return H + L;
}
```

2.3 OCaml uniquement

```
(* Exercice 52 *)
let minecart (tab: int array): int =
   if Array.length tab = 0 then failwith "Tableau aussi remplit que le cerveau d'un ecg"
       else
   let abs a = if a < 0 then -a else a in let min = ref (abs (tab.(0))) in
   Array.iteri (fun i el -> if abs(el - i) < !min then min := abs(el - i)) tab;
   !min;;

(* Exercice 55 | L'enfer *)
let equilibre tab = if Array.length tab = 0 then true;
let tbl = Hashtbl.create (Array.length tab) in
Array.iter (fun el -> Hashtbl.replace tbl el (match Hashtbl.find_opt tbl el with
  | None -> 1
  | Some(a) -> a + 1)) tab;
let first = Hashtbl.find tbl tab.(0) in
let arr = Array.make (Hashtbl.length tbl) 0 in
let ind = ref 0 in
Hashtbl.iter (fun _ valeur -> arr.(!ind) <- (valeur - first); incr ind) tbl;
Array.fold_left (fun acc valeur -> acc && (valeur = 0)) true arr;;


(* Exercice 57 *)
let rpz (s: string): int =
    let size = String.length s in
    let counts = Array.make 5 0 in
      for i = 0 to size - 1 do match String.get s i with
        | ’m’ | ’M’ -> counts.(0) <- counts.(0) + 1
        | ’o’ | ’O’ -> counts.(1) <- counts.(1) + 1
        | ’s’ | ’S’ -> counts.(2) <- counts.(2) + 1
        | ’e’ | ’E’ -> counts.(3) <- counts.(3) + 1
        | ’l’ | ’L’ -> counts.(4) <- counts.(4) + 1
        | _ -> ()
      done;

      counts.(3) <- counts.(3) / 2;
      counts.(4) <- counts.(4) / 2;

    Array.fold_left (fun acc x -> if x < acc then x else acc) counts.(0) counts
;;



(* Exercice 58 *)
let decomp n = let get_next_prime n = let next = ref (n+1) in let is_prime num = if num
    mod 2 = 0 then num = 2
      else begin
      let rec aux counter =
      if num = counter then true
      else (num mod counter <> 0) && aux (counter+2)
      in num <> 1 && aux 3; (* optimisable avec l’algo de Rabin - Miller... *)
      end in
      while not (is_prime !next) do
        incr next;
      done;
      !next in
    let rec aux liste copy prime =
      if copy = 1 then liste
      else
      begin
        if copy mod prime = 0 then match liste with
        | (p, n)::q when p = prime -> aux ((p, n+1)::q) (copy/prime) prime (* MARCHE !*)
        | _ -> aux ((prime, 1)::liste) (copy / prime) prime
        else aux liste copy (get_next_prime prime)
      end
      in aux [] n 2;;
```