Correction du DS 4

Julien Reichert

Questions de cours

Question de cours A1 : On peut procéder par induction structurelle comme pour le cas des feuilles dans un arbre binaire strict, mais aussi par une simple récurrence en signalant que chaque nœud ajouté à la place d'un vide retire le vide en question mais en crée deux à la place (les enfants du nouveau nœud), ce qui augmente le nombre de nœuds et le nombre de vides d'un. Comme on peut créer un arbre binaire en créant d'abord la racine (un nœud, deux vides donc initialisation ok, et par ailleurs pour un arbre vide la propriété est vraie aussi mais ne sert pas forcément pour la preuve d'hérédité qu'on prépare) et en faisant des insertions étape par étape en remplaçant des vides, la propriété est bien héréditaire.

Question de cours A2 : Un peigne droit est un arbre binaire dont chaque nœud a son fils gauche vide (définition alternative dans le cas d'arbres binaires pouvant posséder des feuilles : chaque nœud interne a une feuille pour fils gauche).

Question de cours A3 : Un arbre binaire complet de hauteur $h \geq 0$ (donc non vide) est de taille $2^{h+1} - 1$. Ceci se prouve par induction : si $h = 0$, l'arbre est réduit à sa racine, ce qui fait $2^{0+1} - 1$ nœuds, et un arbre binaire complet de hauteur $h + 1$ est une racine avec deux enfants qui sont des arbres binaires complets forcément tous deux de hauteur h, pour un nombre total de nœuds de $1 + (2^{h+1} - 1) + (2^{h+1} - 1) = 2^{h+2} - 1$.

Questions de cours en OCaml

Question de cours B1 :

```
type 'a arbre = Noeud of 'a * 'a arbre list;; (* type somme *)

type 'a arbre = { etiq : int ; fils : 'a arbre list };; (* type enregistrement, vide impossible *)
```

Question de cours B2 :

Test d'égalité : `=`; test de différence : `<>`; modification d'une référence : `:=`; modification d'un élément dans un tableau : `<-`.

Question de cours B3 :

```
type soma = A | B of somb
and  somb = C | D of soma;;

let somafini = B (D (A));;
let rec somainfini = B (D (somainfini));;

type enrg = { a : bool ; b : enrg option };;

let enrgfini = { a = false ; b = None };;
let rec enrginfini = { a = true ; b = Some enrginfini };;
```

Questions de cours en C

Question de cours C1 : L'option est `-fsanitize=address`.

Question de cours C2 :

```
struct ibtn { int etiq; struct ibtn* fg; struct ibtn* fd; };
typedef struct ibtn nabi;

struct ibt { nabi* racine; };
typedef struct ibt abi;
```

Question de cours C3 :

Création des structures :

```
struct recb;

struct reca { int a; struct recb* b; };
struct recb { int c; struct reca* d; };
```

Instances en question (ici dans une fonction `main`) :

```
int main()
{
  struct recb bfini = { .c = 42 , .d = NULL };
  struct reca afini = { .a = 73 , .b = &bfini };

  printf("%d\n", afini.b->c);

  struct recb binfini = { .c = 1 , .d = NULL };
  struct reca ainfini = { .a = 0 , .b = &binfini };
  binfini.d = &ainfini;

  printf("%d\n", ainfini.b->d->b->d->b->d->a);

  /*
  // Une magnifique boucle infinie heureusement inhibée par la délimitation d'un commentaire !
  struct reca* p = &ainfini;
  while (p != NULL)
  {
    printf("%d\n%d\n", p->a, p->b->c);
    p = p->b->d;
  }
  */

  return 0;
}
```

Exercices

Exercice 1 :

```
type ab = V | N of ab * ab;;

let creer_ab_complet h = (* int -> ab *)
  assert (h >= -1); (* Vérification une seule fois *)
  let rec aux h = match h with (* Donc une récursion un peu artificielle... *)
  | -1 -> V
  | h -> let ahmu = aux (h-1) in N(ahmu, ahmu)
  in aux h;;

let est_complet a = (* ab -> bool *)
  let rec completude a = match a with
  | V -> (true, -1)
  | N(g, d) ->
    let (b1, h1) = completude g
    and (b2, h2) = completude d
    in (b1 && b2 && h1 = h2, 1 + h1) (* Si le booléen est faux, on se fiche de la hauteur *)
  in fst (completude a);;
```

En créant la variable `ahmu` dans la première fonction, on se rend compte que la complexité en temps est linéaire (la variable peut être vue comme un pointeur, elle n'est pas parcourue). Mais comme pour créer un espace d'une certaine taille il faut un temps supérieur ou égal, l'espace aussi est linéaire. C'est le grand avantage des structures persistantes : en pratique le fils gauche et le fils droit d'un arbre binaire créé par cette fonction sont au même endroit dans la mémoire.

Concernant la fonction `est_complet`, la complexité en temps se calcule selon la formule (valable pour les arbres binaires complets, sinon c'est un $\leq$) $c_h = 2c_{h-1} + \mathcal{O}(1)$, soit une complexité exponentielle en la hauteur, autrement dit linéaire en la taille de l'arbre en tant que l'objet abstrait traité (mais pas forcément linéaire en la taille de l'argument vu que celui-ci peut être représenté de manière concise, comme on vient de le voir). En ce qui concerne la complexité en espace, lorsque l'algorithme part sur un des fils, l'autre est laissé en attente, de sorte qu'on puisse imaginer que sur l'ensemble de la branche allant jusqu'au nœud traité tous les autres nœuds sont l'occasion d'avoir un élément sur la pile d'appels, celle-ci sera alors de taille inférieure ou égale à la hauteur de l'arbre.

Pour se convaincre des complexités en temps annoncées, on pourra tester la première fonction avec un argument de l'ordre du million, elle termine immédiatement, mais la deuxième supportera difficilement des arbres de hauteur au-delà de la vingtaine, même s'ils sont obtenus par la fonction précédente.

On notera que ce n'est pas du tout la même chose que si on avait écrit une autre version de la première fonction (voir aussi l'exercice 8 à ce sujet) :

```
let rec creer_ab_complet h =
match h with
  | _ when h < -1 -> failwith "Hauteur inférieure à -1"
  | -1 -> V
  | h -> N(creer_ab_complet (h-1), creer_ab_complet (h-1));;
```

Exercice 2 :

```
struct m_l_c_i { int valeur; struct m_l_c_i* suivant; };
typedef struct m_l_c_i milc;

struct l_c_i { milc* debut; };
typedef struct l_c_i ilc;
```

L'allocation de mémoire doit se faire pour toute création de maillon par une fonction devant muter une liste chaînée. Si on crée une liste chaînée dans le `main`, ce n'est pas nécessaire, par exemple :

```
int main()
{
  milc m0 = { .valeur = 0 , .suivant = NULL };
  milc m1 = { .valeur = 1 , .suivant = &m0 };
  milc m2 = { .valeur = 2 , .suivant = &m1 };
  milc m3 = { .valeur = 3 , .suivant = &m2 };
  ilc l = { .debut = &m3 };
  // travailler sur l

  return 0;
}
```

Mais pour retourner une liste chaînée dans une autre fonction, la gestion de la mémoire fait que cela change :

```
ilc* ilc012()
{
  milc* pm0 = malloc(sizeof(milc));
  pm0.valeur = 0;
  pm0.suivant = NULL;
  milc* pm1 = malloc(sizeof(milc));
  pm1.valeur = 1;
  pm1.suivant = pm0;
  milc* pm2 = malloc(sizeof(milc));
  pm2.valeur = 2;
  pm2.suivant = pm1;
  return { .debut = pm2 };
}
```

Concernant la libération, on imagine que la mémoire a été allouée pour chaque maillon, mais pas forcément pour la liste chaînée.

La raison est que la fonction qui appelle la fonction de libération connaît l'adresse de la liste chaînée et peut toujours faire la libération de l'éventuel pointeur en question elle-même par la suite si nécessaire, alors que libérer de la mémoire non allouée déclenche une erreur (idem si elle était déjà libérée, pour information).

Pour cette raison l'argument sera de type `ilc` et non pas `ilc*` (c'était cependant tout à fait envisageable).

```
void liberer(ilc l)
{
  milc* m = l.debut;
  milc* buff;
  while (m != NULL)
  {
    buff = m->suivant;
    // Sans buff mais en récursif, on peut écrire à la place :
    // liberer(&m->suivant);
    // Le while sera alors un if !
    // L'ordre des libérations restera correct.
    free(m);
    m = buff;
  }
}
```

Exercice 3 :

La correction sera en OCaml uniquement, à la suite de la recommandation (en C, on peut très bien faire un algorithme naïf ou commencer par implémenter une structure de table de hachage). La séquence choisie ici est le tableau.

```
let premier_plus_frequent tab = (* 'a array -> 'a *)
  let n = Array.length tab in assert (n > 0);
  let maxi = ref 1 in
  let xmax = ref tab.(0) in
  let th = Hashtbl.create 8 in
  for i = 0 to n - 1 do
    let x = tab.(i) in
    if Hashtbl.mem th x then
    begin
      let nocc = Hashtbl.find th x in
      if nocc >= !maxi then
      begin
        maxi := nocc + 1;
        xmax := x
      end;
      Hashtbl.replace th x (nocc + 1)
    end
    else Hashtbl.add th x 1
  done; !xmax;;
```

Exercice 4 :

La correction se fera en OCaml uniquement, mais en utilisant des tableaux pour les pièces et en renvoyant un tableau indiquant pour chaque pièce le nombre, éventuellement nul, d'occurrences à utiliser pour la pièce de même indice dans le tableau en argument. On pourrait faire un algorithme similaire en C (programmation dynamique *bottom-up*), à ceci près qu'il faut gérer correctement la mémoire annexe. Pour retourner un tableau de couples, on pourra créer une structure pour les couples en question.

```
let rendu_detail pieces s = (* int array -> int -> int array *)
  let n = Array.length pieces in (* pieces n'est pas forcément monotone, mais peu importe *)
  let optimum = Array.make (s+1) (-1) and info_supp = Array.make (s+1) (-1) in
  optimum.(0) <- 0;
  for i = 1 to s do
    for j = 0 to n-1 do
      if i >= pieces.(j) && optimum.(i-pieces.(j)) > -1 &&
        (optimum.(i) = -1 || optimum.(i) > optimum.(i-pieces.(j)) + 1) then
      begin
        optimum.(i) <- optimum.(i-pieces.(j)) + 1;
        info_supp.(i) <- j
      end
    done
  done;
  let a_rendre = Array.make n 0 and somme = ref s in
  while !somme > 0 do
    let j = info_supp.(!somme) in
    if j = -1 then failwith "Rendu impossible";
    a_rendre.(j) <- a_rendre.(j) + 1;
    somme := !somme - pieces.(j)
  done;
  a_rendre;;
```

Exercice 5 :

Pour changer, le programme sera fait en C. Il est tout aussi facile de faire la même chose en OCaml. Les tables de hachage ne sont pas nécessaire vu qu'on peut avoir une complexité linéaire en espace avec la méthode utilisée ici.

```
int min_absent_apres_min(int* tab, int taille)
{
  assert(taille > 0);
  int mini = tab[0];
  for (int i = 1 ; i < taille ; i += 1) { if (tab[i] < mini) mini = tab[i]; }
  bool* presents = malloc(taille * sizeof(bool));
  for (int i = 0 ; i < taille ; i += 1) presents[i] = false;
  for (int i = 0 ; i < taille ; i += 1) // Ne pas faire les deux boucles en même temps !
  {
    int tabidecale = tab[i] - mini;
    if (tabidecale < taille) presents[tabidecale] = true;
  }
  int indice = 0;
  while (indice < taille && presents[indice]) indice += 1;
  free(presents);
  return indice + mini;
}
```

Exercice 6 :

Envoyer (de droite à gauche, c'est mieux) les éléments vers la fin a une complexité de l'ordre du produit de k et n, ce qui est trop quand k n'est pas négligeable devant n. Une autre solution est de faire des échanges par décalages de k crans, mais la fin est à gérer aussi pour un coût total de l'ordre de $n+k(n \text{ modulo } k)$, dont la valeur est difficile à estimer mais qui est très intéressante si n est un multiple de k. Construire la réponse en recopiant d'abord le reste puis le bloc de taille k, puis tout recopier dans la liste de départ a une complexité linéaire en temps, mais en espace aussi, ce qui exclut cette option au vu de l'énoncé.

L'algorithme classique revient à procéder à trois « miroirs » en place, sur le bloc, sur le reste et sur l'ensemble. [1] Dans ce cas le bloc et le reste reviennent dans le bon sens et leurs positions sont échangées entre temps.

Une mise en œuvre en C :

```
void envoie_bloc(int tab[], int taille, int k)
{
  assert(k <= taille);
  int seuils_bas[3] = { 0 , k , 0 };
  int seuils_haut[3] = { k - 1 , taille - 1 , taille - 1 };
  for (int i = 0 ; i < 3 ; i += 1)
  {
    int fin = seuils_haut[i];
    int debut = seuils_bas[i];
    while (debut < fin)
    {
      int buffer = tab[debut];
      tab[debut] = tab[fin];
      tab[fin] = buffer;
      debut += 1; fin -= 1;
    }
  }
}
```

1. Pour information, cette technique fait partie d'une optimisation du tri fusion pour qu'il soit en place et en temps $\mathcal{O}(n \log n)$.

Exercice 7 :

Pour le plaisir de le faire une première fois, ce sera en OCaml avec un dictionnaire réalisé par une pseudo-référence d'ABR dont les étiquettes seront des couples formés par une clé de type quelconque et le nombre d'occurrences qui sera une référence d'entiers.

Sans optimisation de la structure d'ABR, la complexité est partout (sauf la création...) en $\mathcal{O}(n)$ où n est le nombre de valeurs distinctes à traiter. Si les ABR sont équilibrés, cela devient logarithmique.

On notera l'utilisation d'une référence de listes dans la récursion, c'est pour éviter d'utiliser l'opérateur de fusion qui empirerait la complexité.

```
type 'a abr = V | N of 'a abr * ('a * int ref) * 'a abr;;

type 'a table_comptage = { mutable arbre : 'a abr };;

let creer_table_comptage () = { arbre = V };;

let incr_table_comptage tc x =
  let rec aux a = match a with
  | V -> N(V, (x, ref 1), V)
  | N(g, (y, r), d) when y = x -> incr r; a
  | N(g, (y, r), d) when y < x -> N(g, (y, r), aux d)
  | N(g, (y, r), d) -> N(aux g, (y, r), d)
  in tc.arbre <- aux tc.arbre;;

let acces_table_comptage tc x =
  let rec aux a = match a with
  | V -> raise Not_found
  | N(g, (y, r), d) when y = x -> !r
  | N(g, (y, r), d) when y < x -> aux d
  | N(g, (y, r), d) -> aux g
  in aux tc.arbre;;

let liste_cles_table_comptage tc =
  let rec aux a = match a with
  | V -> ()
  | N(g, (y, _), d) -> buff := y::bugg; aux g ; aux d
  in let buff = ref [] in aux tc.arbre; !buff;;
```

Exercice 8 :

Le type `t` défini dans cet exercice peut être assimilé au type des arbres binaires, avec un ordre alternatif pour les constructeurs. La fonction calcule le minimum des clés d'un arbre binaire, avec la valeur `max_int` si l'arbre est vide au lieu d'une erreur (cela simplifie les filtrages).

On remarquera une complexité catastrophique en raison de l'absence de stockage des résultats des appels récursifs dans une variable. Dans le pire des cas (le minimum n'est jamais à la racine, par exemple si l'arbre est un tas-max), il y a un appel sur chacun des fils pour déterminer si on doit entrer dans le deuxième cas du filtrage, puis qu'on y aille ou qu'on aille dans le troisième il y a un nouvel appel sur l'un des fils, et si la valeur obtenue est inférieure à la racine un quatrième appel le récupère de nouveau. En admettant la structure équilibrée, chaque fils est de taille la moitié de la taille de l'arbre, ce qui donne une formule $c_n = 4c_{\frac{n}{2}} + \mathcal{O}(1)$, soit un temps quadratique. Ceci étant, on peut très bien imaginer un peigne décroissant où à chaque degré de profondeur il faut descendre une fois sur le fils vide et trois fois sur l'autre, donnant cette fois-ci en raisonnant sur la hauteur $c_k = 3c_{k-1}$ d'où une complexité de l'ordre de trois à la puissance de la hauteur... qui s'avère aussi être la taille.

Problème 1

```
type 'a elem_deque = { valeur : 'a; mutable suivant : 'a elem_deque option;
    mutable precedent : 'a elem_deque option };;

type 'a deque = { mutable vide : bool; mutable debut : 'a elem_deque option;
    mutable fin : 'a elem_deque option };;

let creer_deque () = { vide = true; debut = None; fin = None };;

let ajouter_debut_deque l e =
  let ebis = { valeur = e; suivant = None; precedent = None } in
  if not l.vide then
  ( ebis.suivant <- l.debut;
      match l.debut with
      | Some el -> el.precedent <- Some ebis
      | _ -> failwith "Ce cas ne se produit pas" )
  else
  ( l.vide <- false; l.fin <- Some ebis );
  l.debut <- Some ebis;;

let ajouter_fin_deque l e =
  let ebis = { valeur = e; suivant = None; precedent = None } in
  if not l.vide then
  ( ebis.precedent <- l.fin;
      match l.fin with
      | Some el -> el.suivant <- Some ebis
      | _ -> failwith "Ce cas ne se produit pas" )
  else
  ( l.vide <- false; l.debut <- Some ebis; );
  l.fin <- Some ebis;;

let retirer_debut_deque l =
  match l.vide, l.debut with
  | true, _ -> failwith "Ldc vide"
  | false, None -> failwith "Cas impossible"
  | false, Some ebis -> let e = ebis.valeur in
      l.debut <- ebis.suivant;
      (match l.debut with
      | None -> l.vide <- true; l.fin <- None;
      | Some deb -> deb.precedent <- None)
  e;;

let retirer_fin_deque l =
  match l.vide, l.fin with
  | true, _ -> failwith "Ldc vide"
  | false, None -> failwith "Cas impossible"
  | false, Some ebis -> let e = ebis.valeur in
      l.fin <- ebis.precedent;
      (match l.fin with
      | None -> l.vide <- true; l.debut <- None;
      | Some fi -> fi.suivant <- None)
  e;;
```

Problème 2

Question P1 :

Puisque les entiers tiennent sur 32 bits avec les compilateurs C actuels, les données sont quatre fois trop grosses dans la version utilisée par rapport à ce que l'on pourrait avoir.

Question P2 :

Il y a un problème sur l'opération de reste dans la division euclidienne en C (et en OCaml d'ailleurs), et en particulier -1 modulo 256 donnera -1.

Le mieux est de glisser un test pour les valeurs négatives ou d'additionner 255 au lieu de retirer un, avant de faire la division euclidienne.

Dans tous les cas, on trouve ici un avantage à utiliser des entiers sur plus d'un octet, pour ne pas dépendre de la spécification en cas de dépassement arithmétique. Mais un type représentant les entiers non signés sur 8 bits avec la garantie que les calculs soient exacts modulo 256 serait idéal.

Question P3 :

Comme pour l'exercice de la liste demandant de calculer le minimum et le maximum d'une liste, on créera un pointeur d'entiers pour renvoyer les deux entiers attendus.

```
int* minmax(char* programme)
{
  int min = 0;
  int max = 0;
  int actuel = 0;
  for (int i = 0 ; programme[i] != '\0' ; i += 1)
  {
    if (programme[i] == '<')
    {
      if (actuel == min) min = actuel - 1;
      actuel -= 1;
    }
    else if (programme[i] == '>')
    {
      if (actuel == max) max = actuel + 1;
      actuel += 1;
    }
  }
  int* rep = malloc(2 * sizeof(int));
  rep[0] = min;
  rep[1] = max;
  return rep;
}
```

Question P4 :

Le nombre de caractères lus ne peut pas être anticipé puisqu'il peut dépendre des caractères lus. L'exemple le plus simple est le programme `,[,]` qui lit toute l'entrée et ne fait rien d'autre. Le nombre de caractères lus est le nombre de caractères tout court. Parfois c'est plus compliqué : `+[,[.----------------]<]` imprime tous les caractères jusqu'à rencontrer une espace ou la fin du flux d'entrée.

Concernant les positions extrémales, en changeant le petit programme précédent on peut aussi les faire dépendre du flux d'entrée : `,[>,]` écrit sur les données tout le contenu du flux d'entrée.

Sans utiliser le flux d'entrée, les positions extrémales peuvent se calculer en exécutant le programme, mais leur connaissance nécessite de connaître l'état des données, vu qu'une boucle du genre `[>]` avance jusqu'à la prochaine valeur nulle, qui n'est pas évidente à déduire après quelques milliers d'instructions...

Question P5 :

Éventuellement, on mettra le programme dans un autre argument de `argv`, mais ce n'est pas la consigne. Encore mieux : les deux sont des liens vers un fichier, ou alors on laisse le choix entre un lien ou du code brut suivant une option qui précise au compilateur ce qu'il en est.

```
int main(int argc, char** argv)
{
  assert(argc == 2) // Juste pour le warning
  char programme[] = ",[..,]";
  interpreteur(programme, argv[1]);

  return 0;
}
```

Question P6 :

```
bool syntaxe_correcte(char* programme)
{
  int imbrication = 0;
  for (int i = 0 ; programme[i] != '\0' ; i += 1)
  {
    if (programme[i] == '[') imbrication += 1;
    else if (programme[i] == ']')
    {
      if (imbrication == 0) return false;
      imbrication -= 1;
    }
  }
  return imbrication == 0;
}
```

Question P7 :

On va retourner un tableau aplati dont le premier élément sera le nombre de couples de parenthèses et les suivants seront les appariements en alternant les indices (impairs : parenthèse ouvrante, pairs : la parenthèse fermante correspondant à l'ouvrante à l'indice précédent).

Comme dans l'exercice correspondant du TP 8, qui n'était à faire qu'en OCaml, on va utiliser une pile pour stocker les crochets en attente d'appariement.

La structure de pile sera implémentée en amont, on utilisera des piles bornées et la capacité maximale sera calculée par un premier passage dans le programme pour compter le nombre de crochets ouvrants.

Ici, on remarque que les appariements sont triés par position croissante des crochets fermants.

En anticipant ce qui suit, cela permettra une recherche logarithmique du crochet ouvrant correspondant au crochet fermant à un certain indice.

On propose alors à la suite une deuxième version pour avoir en plus la même information triée par position décroissante des crochets ouvrants, sans avoir besoin de faire de tri.

```
int* creer_pileb(int capacite)
{
  int* p = malloc((capacite+1) * sizeof(int));
  p[0] = 0;
  return p;
}

void empiler_pileb(int* p, int capacite, int x)
{
  assert(p[0] < capacite);
  p[0] += 1;
  p[p[0]] = x;
}

int depiler_pileb(int* p)
{
  assert(p[0] > 0);
  int reponse = p[p[0]];
  p[0] -= 1;
  return reponse;
}

// Fonction non utilisée ici, on fait plutôt l'assertion initiale.
bool est_vide_pileb(int* p)
{
  return p[0] == 0;
}

int* couples_crochets(char* programme)
{
  assert (syntaxe_correcte(programme));
  int nombre_crochets = 0;
  for (int i = 0 ; programme[i] != '\0' ; i += 1)
  {
    if (programme[i] == '[') nombre_crochets += 1;
  }
  int* pile = creer_pileb(nombre_crochets);
  int* reponse = malloc((2 * nombre_crochets + 1) * sizeof(int));
  reponse[0] = nombre_crochets;
  int indice_reponse = 1;
  for (int i = 0 ; programme[i] != '\0' ; i += 1)
  {
    if (programme[i] == '[') empiler_pileb(pile, nombre_crochets, i);
    else if (programme[i] == ']')
    // Ici aussi plus besoin de vérifier quoi que ce soit
    {
      int j = depiler_pileb(pile);
      reponse[indice_reponse] = j;
      reponse[indice_reponse+1] = i;
      indice_reponse += 2;
    }
  }
  free(pile);
  return reponse;
}
```

```
int* couples_crochets_autre_tri(char* programme)
{
  assert (syntaxe_correcte(programme));
  int nombre_crochets = 0;
  int fin = strlen(programme);
  for (int i = 0 ; i < fin ; i += 1)
  {
    if (programme[i] == '[') nombre_crochets += 1;
  }
  int* pile = creer_pileb(nombre_crochets);
  int* reponse = malloc((2 * nombre_crochets + 1) * sizeof(int));
  reponse[0] = nombre_crochets;
  int indice_reponse = 1;
  for (int i = fin-1 ; i >= 0 ; i -= 1)
  {
    if (programme[i] == ']') empiler_pileb(pile, nombre_crochets, i);
    else if (programme[i] == '[')
    {
      int j = depiler_pileb(pile);
      reponse[indice_reponse] = i;
      reponse[indice_reponse+1] = j;
      indice_reponse += 2;
    }
  }
  free(pile);
  return reponse;
}
```

Question P8 :

La structure qu'on va utiliser est le tableau redimensionnable dans les deux sens. Une possibilité est de doubler la taille par la gauche ou par la droite suivant le dépassement qui menace, sachant qu'un recopiage est nécessaire quoi qu'il arrive et qu'en réfléchissant bien on peut faire correspondre les indices lors de ce recopiage, une autre est d'utiliser deux supports sous forme de tableau : un pour les indices positifs et un pour les indices strictement négatifs. Cette dernière solution apporte un léger avantage si on décide d'utiliser la fonction `realloc` (certes hors-programme), qui peut optimiser l'ajout par la gauche en plus d'optimiser l'ajout par la droite.

L'interface de cette structure doit de toute manière changer, car elle sera toujours associée à un pointeur indiquant la position courante, et la taille effective sera toujours associée à la capacité.

```
struct t_r_i_b { int position; int capacite_pos ; int capacite_neg ;
  int* donnees_pos ; int* donnees_neg ; };
typedef struct t_r_i_b itrb;

itrb creer_itrb(int n)
{
  itrb t = { .position = 0, .capacite_pos = n, .capacite_neg = n ,
    .donnees_pos = malloc(n * sizeof(int)) ,
    .donnees_neg = malloc(n * sizeof(int)) };
  for (int i = 0 ; i < n ; i += 1)
  {
    t.donnees_pos[i] = 0;
    t.donnees_neg[i] = 0;
  }
  return t;
}
```

```
int acces_courant_itrb(itrb t)
{
  if (t.position >= 0) return t.donnees_pos[t.position];
  else return t.donnees_neg[-1-t.position];
}

void modif_courant_itrb(itrb t, int x)
{
  if (t.position >= 0) t.donnees_pos[t.position] = x;
  else t.donnees_neg[-1-t.position] = x;
}

void redimensionne_itrb(itrb* t)
{
  // redimensionner à droite
  if (t->position > 0)
  {
    int nv_capacite = t->capacite_pos * 2;
    int* nv_tab = malloc(nv_capacite * sizeof(int));
    for (int i = 0 ; i < t->capacite_pos ; i += 1)
    {
      nv_tab[i] = t->donnees_pos[i];
    }
    t->capacite_pos = nv_capacite;
    free(t->donnees_pos);
    t->donnees_pos = nv_tab;
  }
  // redimensionner à gauche
  else
  {
    int nv_capacite = t->capacite_neg * 2;
    int* nv_tab = malloc(nv_capacite * sizeof(int));
    for (int i = 0 ; i < t->capacite_neg ; i += 1)
    {
      nv_tab[i] = t->donnees_neg[i];
    }
    t->capacite_neg = nv_capacite;
    free(t->donnees_neg);
    t->donnees_neg = nv_tab;
  }
}

void decale_indice_itrb(itrb* t, int decalage) // +1 ou -1
{
  int nv_pos = t->position + decalage;
  if (nv_pos == t->capacite_pos || nv_pos == t->capacite_neg - 1) redimensionne_itrb(t);
  t->position = nv_pos;
}
```

Question P9 :

Cette fonction sera utilisée dans les deux sens en pratique, il suffit de changer l'ordre des arguments.

C'est ici qu'on observe la nécessité d'avoir l'argument `fermants` trié, pour éviter que la dichotomie n'échoue.

```
int ouvrant_correspondant(int indice, int* ouvrants, int* fermants, int taille)
{
  int debut = 0;
  int fin = taille - 1;
  while (debut <= fin)
  {
    int milieu = (debut + fin) / 2;
    if (fermants[milieu] == indice) return ouvrants[milieu];
    if (fermants[milieu] > indice) fin = milieu - 1;
    else debut = milieu + 1;
  }
  assert(false);
}
```

Question P10 :

La fonction ne tenant pas sur une page, on la coupe entre toutes les lignes d'initialisation et le corps de la boucle principale.

Pour information, un compilateur minimaliste tient sur environ 300 octets. Le nombre de caractères dans ce programme n'est pas compétitif, il faut croire...

```
void interpreteur(char* programme, char* flux)
{
  itrb t = creer_itrb(64); // 30 000 et tableau circulaire non redimensionnable à l'origine
  int pos_programme = 0;
  int* crochets = couples_crochets(programme);
  int nb_crochets = crochets[0];
  int* ouvrants = malloc(nb_crochets * sizeof(int));
  int* fermants = malloc(nb_crochets * sizeof(int));
  for (int i = 0 ; i < nb_crochets ; i += 1)
  {
    ouvrants[i] = crochets[2*i+1];
    fermants[i] = crochets[2*i+2];
  }
  free(crochets);
  int* crochetsbis = couples_crochets_autre_tri(programme);
  int* ouvrantsbis = malloc(nb_crochets * sizeof(int));
  int* fermantsbis = malloc(nb_crochets * sizeof(int));
  for (int i = 0 ; i < nb_crochets ; i += 1)
  {
    ouvrantsbis[nb_crochets-1-i] = crochetsbis[2*i+1];
    fermantsbis[nb_crochets-1-i] = crochetsbis[2*i+2];
  }
  free(crochetsbis);
  int pos_flux = 0;
  bool explosion = false;
```

```
  while(programme[pos_programme] != '\0')
  {
    if (programme[pos_programme] == '+')
    {
      int x = acces_courant_itrb(t);
      x = (x + 1) % 256;
      modif_courant_itrb(t, x);
    }
    else if (programme[pos_programme] == '-')
    {
      int x = acces_courant_itrb(t);
      x = (x + 255) % 256;
      modif_courant_itrb(t, x);
    }
    else if (programme[pos_programme] == '>')
    {
      decale_indice_itrb(&t, 1);
    }
    else if (programme[pos_programme] == '<')
    {
      decale_indice_itrb(&t, -1);
    }
    else if (programme[pos_programme] == ',')
    {
      if(explosion)
      {
        free(t.donnees_neg); free(t.donnees_pos);
        free(ouvrants); free(fermants); free(ouvrantsbis); free(fermantsbis);
        assert(false);
      }
      int caractere_suivant = (int) flux[pos_flux];
      if (caractere_suivant == '\0') explosion = true; // La prochaine fois cela explose !
      modif_courant_itrb(t, caractere_suivant);
      pos_flux += 1;
    }
    else if (programme[pos_programme] == '.')
    {
      printf("%c", (char) acces_courant_itrb(t));
    }
    else if (programme[pos_programme] == '[')
    {
      if (acces_courant_itrb(t) == 0)
        pos_programme = ouvrant_correspondant(pos_programme, fermantsbis, ouvrantsbis, nb_crochets);
    }
    else if (programme[pos_programme] == ']')
    {
      pos_programme = ouvrant_correspondant(pos_programme, ouvrants, fermants, nb_crochets) - 1;
    }
    pos_programme += 1; // Dans tous les cas, on décale, les blocs ont été écrits pour correspondre...
  }
  free(t.donnees_neg); free(t.donnees_pos);
  free(ouvrants); free(fermants); free(ouvrantsbis); free(fermantsbis);
}
```