1 Pied à l'étrier

R. 1-1 Écrire une fonction récursive retournant le dernier élément d'une liste d'un type quelconque. Cette fonction lèvera une exception `Invalid_argument` dans le cas où la liste est vide. On rappelle que l'exception `Invalid_argument` est prédéfinie en OCaml, elle doit être accompagnée d'une chaîne de caractères. Par exemple : l'évaluation de l'expression `raise (Invalid_argument("ceci est un message d'erreur"))` lève l'exception `Invalid_argument("ceci est un message d'erreur")`.

Solution

0-list-last-0

```
let rec last (l: 'a list): 'a =
  (* Retourne le dernier élément de l. Lève l'exception Invalid_argument
     dans le cas où la liste est vide. *)
  match l with
  | x :: [] -> x
  | _ :: l' -> last l'
  | _       -> raise (Invalid_argument "liste vide")
```

R. 1-2 Écrire une fonction récursive retournant le dernier élément d'une liste d'un type quelconque. Cette fonction doit retourner `None` si la liste est vide et `Some(x)` si la liste contient au moins un élément et que le dernier élément est `x`.

Solution

0-list-last-1

```
let rec last (l: 'a list): 'a option =
  (* Retourne le dernier élément de l. Retourne None si la liste est vide
     et Some(x) si la liste contient au moins un élément et que le dernier
     est x. *)
  match l with
  | x :: [] -> Some(x)
  | _ :: l' -> last l'
  | _       -> None
```

R. 1-3 Écrire une fonction permettant de calculer le miroir d'une liste. Une attention toute particulière sera accordée au fait de ne pas fournir une implémentation en $\mathcal{O}(n^2)$ due à une malencontreuse utilisation de `@`. On n'utilisera pas de fonctionnelle d'itération.

Solution

0-list-rev-0

```
let rev (l: 'a list): 'a list =
  (* Calcule le miroir de la liste l. *)
  let rec aux (l: 'a list) (res: 'a list) =
    match l with
    | []     -> res
    | p :: q -> aux q (p :: res)
  in aux l []
```

R. 1-4 Écrire une fonction permettant de calculer le miroir d'une liste. Une attention toute particulière sera accordée au fait de ne pas fournir une implémentation en $\mathcal{O}(n^2)$ due à une malencontreuse utilisation de `@`. On s'efforcera d'utiliser la fonctionnelle `List.fold_left`.

Solution

0-list-rev-1

```
let rev (l: 'a list): 'a list =
  (* Calcule le miroir de la liste l. *)
  List.fold_left (fun acc x -> x :: acc) [] l
```

R. 1-5 Écrire une fonction permettant de calculer la concaténation de deux listes. On n'utilisera pas de fonctionnelle d'itération.

Solution

0-list-concat-0

```
let concat (lg: 'a list) (ld: 'a list): 'a list =
  (* Calcule la concaténation des listes lg et ld (dans ce sens). *)
  let rec verse (todo: 'a list) (lres : 'a list): 'a list =
    (* Calcule la concaténation du miroir de todo et de lres. *)
    match todo with
    | []   -> lres
    | a::q -> verse q (a :: lres)
  in verse (List.rev lg) ld
```

R. 1-6 Écrire une fonction permettant de calculer la concaténation de deux listes. On s'efforcera d'utiliser la fonctionnelle `List.fold_left`.

Solution

0-list-concat-1

```
let concat (lg: 'a list) (ld: 'a list): 'a list =
  (* Calcule la concaténation des listes lg et ld (dans ce sens). *)
  List.fold_left (fun accu elem -> elem::accu) ld (List.rev lg)
```

R. 1-7 Écrire une fonction prenant en argument une liste `l` et un élément `x` et calculant la liste `l` privée de toutes les occurrences de l'élément `x`.

Solution

0-list-prive

```
let prive_de (l: 'a list) (x: int) : 'a list =
  (* Calcule une liste contenant les éléments de l qui ne sont pas x. *)
  let rec aux (ll: 'a list) (lres: 'a list) : 'a list =
    (* Calcule ll privée de x concaténée au miroir de lres. *)
    match ll with
    | []   -> List.rev lres
    | a::q -> if a <> x then aux q (a :: lres) else aux q lres
  in aux l []
```

R. 1-8 Écrire une fonction prenant en argument une liste `l` et un indice `i` et retournant deux listes : la sous-liste des éléments de `l` d'indice inférieurs stricts à `i` et la sous-liste des éléments de `l` d'indices supérieurs à `i`. Cette fonction doit faire appel à une fonction auxiliaire récursive terminale.

Solution

0-list-decoupe-0

```
let decoupe (l: 'a list) (i: int) : 'a list * 'a list =
  (* Calcule un couple de listes : la liste des éléments de l d'indice
     strictement inférieurs à i, la liste des éléments de l d'indice
```

```
    supérieurs à i. *)
  let rec aux (g: 'a list) (d: 'a list) (i: int) =
    if i = 0 then List.rev g, d
    else
      match d with
      | []     -> List.rev g, d
      | p :: q -> aux (p :: g) q (i-1) in
  aux [] l i
```

R. 1-9 Écrire une fonction prenant en argument une liste `l` et un indice `i` et retournant deux listes : la sous-liste des éléments de `l` d'indice inférieurs stricts à `i` et la sous-liste des éléments de `l` d'indices supérieurs à `i`. Cette fonction doit être récursive et ne pas faire appel à des fonctions récursives auxiliaires.

Solution

0-list-decoupe-1

```
let rec decoupe (l: 'a list) (i: int) : 'a list * 'a list =
  (* Calcule un couple de listes : la liste des éléments de l d'indice
     strictement inférieurs à i, la liste des éléments de l d'indice
     supérieurs à i. *)
  if i = 0 then [], l
  else match l with
    | []     -> [], []
    | p :: q ->
      let g, d = decoupe q (i-1) in
      (p :: g, d)
```

R. 1-10 Écrire une fonction prenant en argument une liste `l` et la découpant en deux listes, dans la première on rangera les éléments d'indices pairs dans `l`, dans la seconde on rangera les éléments d'indices impairs dans `l`. L'ordre des éléments dans les listes résultats doit être celui de la liste d'entrée.

Solution

0-list-unsurdeux

```
let un_sur_deux (l: 'a list) : ('a list * 'a list) =
  (* Retourne un couple de listes : les éléments de l d'indice pair, les
     éléments de l d'indice impair. *)
  let rec aux (lat: 'a list) (lres1: 'a list) (lres2: 'a list)
    : 'a list * 'a list =
    (* lat est la liste restant à traitée, lres1 est la liste des éléments
       d'indice pairs déjà traités, lres2 est la liste des éléments
       d'indice impairs déjà traités. lres1 et lres2 sont à l'envers. *)
    match lat with
    | []          -> (lres1     , lres2)
    | a::[]       -> (a :: lres1, lres2)
    | a::b::suite -> aux suite (a :: lres1) (b :: lres2)
  in
  let (lr1, lr2) = aux l [] [] in
  (List.rev lr1, List.rev lr2)
```

2 Listes de listes

R. 1-11 Écrire une fonction prenant en argument une liste de listes `l` et retournant la liste des éléments se trouvant dans une des listes de `l`. Les éléments devront être rangés dans le même ordre que dans la liste initiale. Par exemple sur la liste `[[1;2]; [3;4;5]; []; [6]]` on obtiendra `[1; 2; 3; 4; 5; 6]`. On utilisera la fonction de concaténation de listes `@`, en prenant garde à ne pas obtenir une complexité quadratique.

Solution

0-list-flatten-0

```
let flatten_1 (l: 'a list list): 'a list =
  (* Retourne la liste l aplatie. *)
  let rec aux (ll: 'a list list) (lres : 'a list) : 'a list =
    (* ll est la liste des listes encore à traiter, lres est la liste (à
       l'envers) des éléments déjà aplaties. *)
    match ll with
    | []   -> lres
    | a::q -> aux q ((List.rev a) @ lres)
    (* on ajoute à gauche : @ de coût linéaire en son opérande gauche *)
```

R. 1-12 Écrire une fonction prenant en argument une liste de listes `l` et retournant la liste des éléments se trouvant dans une des listes de `l`. Les éléments devront être rangés dans le même ordre que dans la liste initiale. Par exemple sur la liste `[[1;2]; [3;4;5]; []; [6]]` on obtiendra `[1; 2; 3; 4; 5; 6]`. On n'utilisera pas, et on ne redéfinira pas non plus, l'opération de concaténation de listes. On utilisera deux fonctions mutuellement récursives.

Solution

0-list-flatten-1

```
let flatten_mtr (l: 'a list list): 'a list =
  (* Retourne la liste l aplatie. *)
  let rec grand_pas (todo: 'a list list) (res: 'a list): 'a list =
    (* todo est la liste des éléments encore à traiter. res est la liste
       des éléments déjà aplatis. *)
    match todo with
    | []      -> List.rev res
    | x :: xs -> petit_pas x xs res
  and petit_pas (une_liste: 'a list) (todo: 'a list list) (res: 'a list)
    : 'a list =
    (* une_liste est une liste de la liste initiale l, que l'on transvase
       dans res. todo est la liste des listes encore à traiter. res est la
       liste des éléments déjà aplatis. *)
    match une_liste with
    | []      -> grand_pas todo res
    | y :: ys -> petit_pas ys todo (y :: res)
  in
  grand_pas l []
```

R. 1-13 Écrire une fonction prenant en argument une liste de listes `l` et retournant la liste des éléments se trouvant dans une des listes de `l`. Les éléments devront être rangés dans le même ordre que dans la liste initiale. Par exemple sur la liste `[[1;2]; [3;4;5]; []; [6]]` on obtiendra `[1; 2; 3; 4; 5; 6]`. On n'utilisera pas, et on ne redéfinira pas non plus, l'opération de concaténation de listes. On utilisera une unique fonction auxiliaire récursive terminale.

Solution

0-list-flatten-2

```
let flatten_tr (l: 'a list list): 'a list =
  (* Retourne la liste l aplatie. *)
  let rec aux (todo: 'a list list) (res: 'a list): 'a list =
    (* todo est la liste des éléments encore à traiter. res est la liste
       des éléments déjà aplatis. *)
    match todo with
    | []             -> List.rev res
    | ([]) :: xs     -> aux xs res
    | (y :: ys) :: xs -> aux (ys :: xs) (y :: res)
  in aux l []
```

R. 1-14 Écrire une fonction prenant en argument une liste de listes `l` et retournant la liste des éléments se trouvant dans une des listes de `l`. Les éléments devront être rangés dans le même ordre que dans la liste initiale. Par exemple sur la liste `[[1;2]; [3;4;5]; []; [6]]` on obtiendra `[1; 2; 3; 4; 5; 6]`. On n'utilisera pas, et on ne redéfinira pas non plus, l'opération de concaténation de listes, on se limitera à des itérations au moyen de la fonctionnelle `List.fold_left`.

Solution

0-list-flatten-3

```
let flatten (l: 'a list list): 'a list =
  (* Retourne la liste l aplatie. *)
  List.rev (
    List.fold_left (fun accu l ->
        List.fold_left (fun acc x ->
            x :: acc
          ) accu l
      ) [] l
  )
```

3 Listes et comparaisons

R. 1-15 Écrire une fonction récursive prenant en argument une liste et retournant un couple dont la première composante est le minimum de la liste et la seconde composante est le maximum. Cette fonction doit être récursive et ne pas utiliser de fonctions récursives auxiliaires. Dans le cas où la liste est vide on lèvera l'exception `Invalid_argument`.

Solution

0-list-minmax-0

```
let rec min_max (l: 'a list): 'a * 'a =
  (* Calcule le minimum et le maximum de la liste l. *)
  match l with
  | [] -> raise (Invalid_argument "liste vide")
  | x :: [] -> (x, x)
  | x :: q  ->
    let mi, ma = min_max q in
    (min x mi, max x ma)
```

R. 1-16 Écrire une fonction récursive prenant en argument une liste et retournant un couple dont la première composante est le minimum de la liste et la seconde composante est le maximum. Cette

fonction doit être récursive terminale et peut utiliser une fonction récursive auxiliaire. Dans le cas où la liste est vide on lèvera l'exception `Invalid_argument`.

Solution

O-list-minmax-1

```
let min_max (l: 'a list) : 'a * 'a =
  (* Calcule le minimum et le maximum de la liste l. *)
  let rec aux (ll: 'a list) (mi: 'a) (ma: 'a) =
    (* ll est la liste des éléments restants à traiter, mi (resp. ma) est
       le minimum (resp. maximum) croisé jusqu'à maintenant. *)
    match ll with
    | [] -> (mi, ma)
    | p :: l' -> aux l' (min p mi) (max p ma)
  in
  match l with
  | [] -> raise (Invalid_argument "liste vide")
  | p :: l -> aux l p p
```

R. 1-17 Écrire une fonction récursive prenant en argument une liste et retournant un couple dont la première composante est le minimum de la liste et la seconde composante est le maximum. Cette fonction doit utiliser la fonctionnelle `List.fold_left` comme mécanisme d'itération. Dans le cas où la liste est vide on lèvera l'exception `Invalid_argument`.

Solution

O-list-minmax-2

```
let min_max (l: 'a list) : 'a * 'a =
  (* Calcule le minimum et le maximum de la liste l. *)
  match l with
  | []   -> raise (Invalid_argument "liste vide")
  | p::l -> List.fold_left (fun (mi, ma) x -> (min x mi, max x ma)) (p, p) l
```

R. 1-18 Écrire une fonction permettant de découper une liste en sous-séquences (non vides) croissantes maximales pour l'extension à gauche. Par exemple la liste `[1; 4; 5; 8; 7; 5; 2; 3; 4; 9; 3; 3; 3; 5; 5; 7; 2]` est découpée en `[[1; 4; 5; 8]; [7]; [5]; [2; 3; 4; 9]; [3; 3; 3; 5; 5; 7]; [2]]`.

Solution

O-list-ssseqcroissante

```
let sous_sequences_croissantes (l: 'a list): ('a list list) =
  (* Décompose la liste l est une liste de sous-séquences croissantes. *)
  let rec aux (ll: 'a list) (ltemp: 'a list) (lres: 'a list list)
    : 'a list list =
    (* ll est la liste des éléments encore à traiter, ltemp est la
       sous-séquence croissante en cours de construction, lres est la liste
       des sous-séquences croissantes déjà construites avec le début de la
       liste l. *)
    match ll, ltemp with
    | [], []     -> List.rev lres
    | [], _      -> List.rev (List.rev ltemp::lres)
    | a::q, []   -> aux q (a::[]) lres
    | a::q, b::r ->
      if a >= b
      then aux q (a :: ltemp) lres
      else aux q (a :: []) ((List.rev ltemp) :: lres)
  in aux l [] []
```

R. 1-19 Écrire une fonction permettant de découper une liste en sous-séquences (non vides) monotones maximales pour l'extension à gauche. Par exemple la liste `[1; 4; 5; 8; 7; 2; 3; 4; 9; 3; 3; 3; 5; 5; 7; 2]` est découpée en `[[1; 4; 5; 8]; [7; 5; 2]; [3; 4; 9]; [3; 3; 3; 5; 5; 7]; [2]]`

Solution

0-list-ssseqmonotone

```
type signe = bool option
(* Some(true) indique une séquence croissante, Some(false) une décroiss. et
   None une suite ni strictement croiss, ni strict décroiss, ie cst *)

let sous_sequences_monotones (l: 'a list): ('a list list) =
  (* Décompose la liste l est une liste de sous-séquences croissantes. *)
  let diff_compatible_avec_signe (s: signe) (a:'a) (b:'a) : bool =
    (* Teste si [a; b] est compatible avec la monotonie indiquée par s *)
    s = None || (a <= b && s = Some true) || (a >= b && s = Some false) in
  let raffine_avec_diff (s: signe) (a: 'a) (b: 'a) : signe =
    (* Raffine la monotonie s afin qu'elle tienne compte de [a; b] *)
    if a = b then s else if a < b then Some true else Some false in
  let rec aux
      (todo: 'a list) (lres: 'a list list)
      (accu: 'a list) (csigne: signe) : 'a list list =
    (* todo est la liste des éléments encore à traiter. lres est la liste
       des sous séquences monotones déjà construites avec le début de l.
       accu est la sous-séquence monotone en cours de construction, elle
       est de monotonie indiquée par csigne *)
    match todo, accu with
    | [], []                  -> []
    | [], _                   -> List.rev ((List.rev accu) :: lres)
    | next::todo', []         -> aux todo' lres (next::accu) None
    | next::todo', last::accu' ->
      if diff_compatible_avec_signe csigne last next
      then aux todo' lres (next::accu) (raffine_avec_diff csigne last next)
      else aux todo' ((List.rev accu) :: lres) [next] None
  in aux l [] [] None
```

R. 1-20 Écrire une fonction prenant en arguments une liste `(l: 'a list)`, un entier `k`, une fonction `f: 'a -> int` à valeurs dans $[\![0, k]\!]$ et retournant la liste des éléments de `l` triés par image par `f` croissante. On demande une implémentation en $\mathcal{O}(\max(n, k))$ où n est la taille de la liste `l`.

Solution

0-list-rangement

```
let rangement (l: 'a list) (k: int) (f: 'a -> int): 'a list =
  (* Calcule la liste des éléments de l, triée par image croissante par f.
     f est à valeurs dans [|0, k|] *)
  let aux = Array.make (k + 1) [] in
  List.iter (fun x -> let i = (f x) in aux.(i) <- x :: aux.(i)) l;
  let ll = Array.to_list aux in
  List.flatten ll
```

4 Quelques utilisations classiques des listes en algorithmique

R. 1-21 Écrire une fonction permettant de trier une liste au moyen de l'algorithme du tri fusion♣.

♣. https://en.wikipedia.org/wiki/Merge_sort

Solution

0-list-trifusion

```
let rec fusion (g: 'a list) (d: 'a list): 'a list =
  (* fusionne deux listes triées par ordre croissant en une liste triée par
     ordre croissant *)
  match g, d with
  | [], _  -> d
  | _ , [] -> g
  | xg :: gr, xd :: dr ->
    if xg < xd then xg :: (fusion gr d)
    else            xd :: (fusion g dr)

let rec split (l: 'a list): 'a list * 'a list =
  (* Retourne un couple de listes : les éléments de l d'indice pair, les
     éléments de l d'indice impair. *)
  match l with
  | []          -> [] , []
  | [x]         -> [x], []
  | x :: y :: l' -> let g, d = split l' in (x :: g, y :: d)

let rec tri_fusion (l: 'a list): 'a list =
  (* Trie la liste l au moyen du tri fusion. *)
  match l with
  | [] | _ :: [] -> l          (* listes de taille <= 1 *)
  | _ ->
    let g, d = split l in
    let gs, ds = tri_fusion g, tri_fusion d in
    fusion gs ds
```

R. 1-22 Fournir une implémentation du type de données abstrait file au moyen de deux listes. On assurera une complexité amortie en $\mathcal{O}(1)$ pour chaque opération.

Solution

0-list-file

```
(* File *)
type 'a file = 'a list * 'a list
(* une file est un couple de listes : [1; 2; 3], [6; 5; 4] représente
   la file contenant -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> dont le prochain
   élément à sortir est 6, le dernier inséré est 1 *)

(** Crée la file vide. *)
let file_vide: 'a file = ([], [])

(** Teste si la file [f] est vide. *)
let est_vide (f: 'a file) : bool = f = ([], [])

(** Enfile l'élément [e] dans la file [f]. *)
let enfile (e: 'a) (f: 'a file): 'a file =
  let (g, d) = f in (e :: g, d)

(** Défile l'élément en tête de la file [f]. La fonction retourne l'élément
    défilé et la nouvelle file. *)
let rec defile (f: 'a file): 'a * 'a file =
    match f with
  | [], []     -> failwith "file vide"
  | g, x :: d' -> x, (g, d')
  | g,  []     -> defile ([], List.rev g)
```

R. 1-23 Écrire une fonction prenant en argument une liste et retournant la liste des permutations de cette liste.

Solution

0-list-permutation

```
(** Calcule la liste de toutes les listes pouvant être obtenues par
    insertion de [x] à toutes les positions possibles dans [l]. *)
let toutes_insertions (x: 'a) (l: 'a list): 'a list list =
  let rec aux (debut: 'a list) (fin: 'a list) (acc: 'a list list) =
    (* debut est le début de la liste l considérée (à l'envers), fin est la
       fin de la liste l considérée, acc est la liste des listes fabriquées
       jusqu'à présent *)
    let nouveau = (List.rev_append (debut) (x :: fin)) in
    match fin with
    | []        -> nouveau :: acc
    | y :: fin' -> aux (y :: debut) fin' (nouveau :: acc)
  in aux [] l []

(** Calcule la liste obtenue en insérant [x] à toutes les positions
    possibles dans toutes les listes de [ll]. *)
let rec toutes_insertions_toutes_listes (x: 'a) (ll: 'a list list)
  : 'a list list =
  match ll with
  | [] -> []
  | l :: ll' -> (toutes_insertions x l)
                @ (toutes_insertions_toutes_listes x ll')

(** Calcule l'ensemble des permutations de la liste [l]. Si un même élément
    apparaît plusieurs fois dans [l], plusieurs permutations sont générés,
    comme si ces éléments étaient distincts. *)
let rec permutations (l: 'a list): 'a list list =
  match l with
  | []      -> [ [] ]
  | x :: l' -> toutes_insertions_toutes_listes x (permutations l')
```

R. 1-24 Écrire une fonction prenant en argument un entier naturel n et retournant la liste des 2^n listes contenant n booléens. Ainsi dans la liste résultat on devra pouvoir trouver chaque liste de n booléens.

Solution

0-list-enumbool

```
(** Calcule la liste des 2^n listes de [n] booléens. *)
let rec enumere_base_deux (n: int): bool list list =
  if n = 0 then [ [] ]
  else
    let ll = enumere_base_deux (n-1) in
    (* on ajoute en tête de ttes les listes de ll le booléen vrai *)
    (List.map (fun l -> true :: l) ll)
    (* on ajoute en tête de ttes les listes de ll le booléen faux *)
    @ (List.map (fun l -> false :: l) ll)
    (* et on concatène les deux listes *)
```

5 Produit de listes

R. 1-25 Écrire une fonction prenant en argument `l1` et `l2` deux listes de types respectifs `'a list` et `'b list`, ainsi qu'une fonction de type `'a -> 'b -> unit` qui appelle cette fonction sur tous les couples d'éléments `(a, b)` avec a∈l1 et b∈l2. Cette fonction ne doit utiliser aucune fonctionnelle d'itération dans cette fonction.

Solution

0-list-itereproduit-0

```
(** Appelle f sur tous les couples d'éléments de l1xl2 *)
let rec itere_sur_produit (l1 :'a list)(l2: 'b list)(f:'a->'b->unit): unit =
  (** Appelle f sur tous les couples (x1, x2) pour x2 dans l2 *)
  let rec itere_sur_liste (x1:'a) (ll2:'b list) : unit =
    match ll2 with
    |[] -> ()
    |x2::q2 -> (f x1 x2; itere_sur_liste x1 q2)
  in
  match l1 with
  |[] -> ()
  |x1::q1 -> (itere_sur_liste x1 l2; itere_sur_produit q1 l2 f)
```

R. 1-26 Écrire une fonction prenant en argument `l1` et `l2` deux listes de types respectifs `'a list` et `'b list`, ainsi qu'une fonction de type `'a -> 'b -> unit` qui appelle cette fonction sur tous les couples d'éléments `(a, b)` avec a∈l1 et b∈l2. Cette fonction doit utiliser la fonctionnelle d'itération `List.iter`.

Solution

0-list-itereproduit-1

Deux versions avec un seul `List.iter`

```
(** Appelle f sur tous les couples d'éléments de l1xl2 *)
let rec itere_sur_produit_bis (l1 :'a list)(l2: 'b list)(f:'a->'b->unit): unit =
  (** Appelle f sur tous les couples d'éléments de {x1}xll2 *)
  let itere_sur_liste (x1:'a) (ll2:'b list) : unit =
    List.iter (fun x2 -> f x1 x2) ll2
  in
  match l1 with
  |[] -> ()
  |x1::q1 -> (itere_sur_liste x1 l2; itere_sur_produit q1 l2 f)
```

```
(** Appelle f sur tous les couples d'éléments de l1xl2 *)
let rec itere_sur_produit_ter (l1 :'a list)(l2: 'b list)(f:'a->'b->unit): unit =
  match l1 with
  | []      -> ()
  | x1 :: q1 ->
     List.iter (fun x2 -> f x1 x2) l2;
     itere_sur_produit q1 l2 f

```

Une version avec deux `List.iter`

```
let itere_sur_produit_quater (l1 :'a list)(l2: 'b list)(f:'a->'b->unit): unit =
  List.iter (fun x1 -> List.iter (fun x2 -> f x1 x2) l2) l1

```

R. 1-27 Écrire une fonction prenant en argument `l1` et `l2` deux listes de types respectifs `'a list`

et `'b list`, et calculant une liste de type `('a * 'b) list` représentant le produit cartésien de `l1` et `l2`. Par exemple le produit de `[1; 2; 2]` et `['a'; 'b']` peut être représenté par `[(3, 'b'); (3, 'a'); (2, 'b'); (2, 'a'); (1, 'b'); (1, 'a')]` ou n'importe quelle permutation de cette liste. Cette fonction doit utiliser une référence et la fonctionnelle d'itération `List.iter`.

Solution

0-list-creeproduit-0

```
let cree_produit (l1 :'a list)(l2: 'b list): (('a*'b) list) =
  let res = ref [] in
  List.iter
    (fun x1 -> List.iter
      (fun x2 -> res:= (x1, x2)::!res)
      l2)
    l1;
  !res

```

R. 1-28 Écrire une fonction prenant en argument `l1` et `l2` deux listes de types respectifs `'a list` et `'b list`, et calculant une liste de type `('a * 'b) list` représentant le produit cartésien de `l1` et `l2`. Par exemple le produit de `[1; 2; 2]` et `['a'; 'b']` peut être représenté par `[(3, 'b'); (3, 'a'); (2, 'b'); (2, 'a'); (1, 'b'); (1, 'a')]` ou n'importe quelle permutation de cette liste. Cette fonction ne doit utiliser aucune fonctionnelle d'itération, en particulier ni `List.iter` ni `List.fold_left`.

Solution

0-list-creeproduit-1

Version non récursive terminale

```
let cree_produit_bis (l1 :'a list)(l2: 'b list): (('a*'b) list) =
  (** calcule une liste contenant tous les couples de {x1}xl12 *)
  let rec prod_singleton_liste (x1:'a) (ll2: 'b list): (('a*'b) list) =
    match ll2 with
    | []      -> []
    | x2::q2  -> (x1, x2)::(prod_singleton_liste x1 q2)
  in
  (** calcule une liste contenant tous les couples de {ll1}xl2 *)
  let rec aux (ll1: 'a list) : (('a*'b) list) =
    match ll1 with
    | []     -> []
    | x1::q1 -> (prod_singleton_liste x1 l2) @ aux q1
                                                (* surtout pas l'inverse : on deverse
                                                ↪  la liste courte dans la longue *)
  in aux l1

```

Version récursive terminale

```
let cree_produit_ter (l1 :'a list)(l2: 'b list): (('a*'b) list) =
  (** calcule la concaténation des couples de {x1}xll2 et accu *)
  let rec prod_singleton_liste (x1:'a) (ll2: 'b list) (accu:('a*'b) list): (('a*'b) list)
  ↪  =
    match ll2 with
    | []     -> accu
    | x2::q2 -> prod_singleton_liste x1 q2 ((x1, x2)::accu)
  in
  (** calcule la concaténation des couples de ll1xl2 et accu *)
  let rec aux (ll1: 'a list) (accu:('a*'b) list): (('a*'b) list) =
     match ll1 with
```

```
11      | []     -> accu
12      | x1::q1 -> aux q1 (prod_singleton_liste x1 l2 accu)
13    in aux l1 []
```

1 Pied à l'étrier

R. 2-1 Écrire une fonction calculant l'indice du premier `0` dans un tableau d'entiers parcouru par boucle `while`. La fonction devra retourner `-1` si le tableau ne contient aucun `0`.

Solution

0-tableau-findfirst-0

```
(** Calcule l'indice du premier 0 du tableau [t], retourne -1 si un tel 0
    n'a pu être trouvé. *)
let find_indice_first_0 (t: int array): int =
  let n = Array.length t in
  let i = ref 0 in
  while (!i < n && t.(!i) <> 0) do
    incr i;
  done;
  if !i = n then -1 else !i
```

R. 2-2 Écrire une fonction calculant l'indice du premier `0` dans un tableau d'entiers parcouru par boucle `for`, arrêtée au moyen d'une levée d'exception. La fonction devra retourner `-1` si le tableau ne contient aucun `0`.

Solution

0-tableau-findfirst-1

```
(** Calcule l'indice du premier 0 du tableau [t], retourne -1 si un tel 0
    n'a pu être trouvé. *)
exception Found of int
let find_indice_first_0_bis (t: int array): int =
  let n = Array.length t in
  try
    for i = 0 to (n-1) do
      if t.(i) = 0 then raise (Found(i))
    done; -1
  with
  | Found(i) -> i
```

R. 2-3 Écrire une fonction calculant l'indice du *dernier* `0` dans un tableau d'entiers parcouru par boucle `while`. La fonction devra retourner `-1` si le tableau ne contient aucun `0`.

Solution

0-tableau-findlast-0

```
(** Calcule l'indice du premier 0 du tableau [t], retourne -1 si un tel 0
    n'a pu être trouvé. *)
let find_indice_last_0 (t: int array): int =
  let n = Array.length t in
  let i = ref (n-1) in
  while (!i >=0 && t.(!i) <> 0) do
    i := !i-1;
  done;
  !i (* if !i = -1 then -1 else !i *)
```

R. 2-4 Écrire une fonction testant l'existence d'un `0` dans un tableau d'entiers avec une boucle `while`, arrêtée dès que possible sans utiliser d'exception. La fonction doit retourner `true` s'il est possible de trouver la valeur 0 dans le tableau et `false` sinon.

Solution

O-tableau-existe0-0

```
(** Calcule si le tableau [t] contient un 0. *)
let existe_0 (t: int array): bool =
  let res = ref false in
  let i = ref 0 in
  while (!i < Array.length t && not !res) do
    if t.(!i) = 0 then res := true else ();
    i := !i+1;
  done;
  !res
```

R. 2-5 Écrire une fonction testant l'existence d'un `0` dans un tableau d'entiers avec une fonctionnelle `Array.iter`, arrêtée par une levée d'exception. La fonction doit retourner `true` s'il est possible de trouver la valeur 0 dans le tableau et `false` sinon.

Solution

O-tableau-existe0-1

```
(** Calcule si le tableau [t] contient un 0. *)
exception Found
let existe_0_bis (t: int array): bool =
  try  Array.iter (fun x -> if x = 0 then raise Found) t; false
  with | Found -> true
```

R. 2-6 Écrire une fonction prenant en argument un tableau d'entiers et calculant la somme des éléments du tableau. On fournira une implémentation au moyen d'une boucle `for`.

Solution

O-tableau-somme-0

```
(** Calcule la somme des éléments du tableau [t]. *)
let somme_tab (t: int array) : int =
  let res = ref 0 in
  let n = Array.length t in
  for i = 0 to (n-1) do
    res := !res + t.(i)
  done;
  !res
```

R. 2-7 Écrire une fonction prenant en argument un tableau d'entiers et calculant la somme des éléments du tableau. On fournira une implémentation au moyen de la fonctionnelle `Array.fold_left`.

Solution

O-tableau-somme-1

```
(** Calcule la somme des éléments du tableau [t]. *)
let somme_tab2 (t: int array) : int =
  Array.fold_left (fun acc x -> acc + x) 0 t
```

R. 2-8 Écrire une fonction calculant l'indice du minimum dans un tableau d'entiers parcouru par une boucle `for`. Cette fonction déclenchera l'exception `Invalid_argument` dans le cas où le tableau

est de taille `0`.

Solution

O-tableau-indicemin

```
(** Calcule l'indice du plus petit élément du tableau d'entiers [t]. Lève
    l'exception [Invalid_argument] si le tableau est vide. *)
let indice_minimum (t: int array): int =
  let n = Array.length t in
  if n = 0 then raise (Invalid_argument "tableau de taille 0")
  else
    let i_min = ref 0 in
    let v_min = ref t.(0) in
    for i = 1 to (n-1) do
      if t.(i) < !v_min then
        begin
          i_min := i;
          v_min := t.(i)
        end
    done;
    !i_min
```

R. 2-9 Écrire une fonction calculant l'indice du maximum dans un tableau d'entiers positifs parcouru par une fonctionnelle `Array.fold_left`. Cette fonction déclenchera l'exception `Invalid_argument` dans le cas où le tableau est de taille `0`.

Solution

O-tableau-indicemax

```
(** Calcule l'indice du plus petit élément du tableau d'entiers [t]. Lève
    l'exception [Invalid_argument] si le tableau est vide. *)
let indice_maximum (t: int array): int =
  let aux = (fun (i_max, v_max, i) _ ->
      if t.(i) > v_max then (i, t.(i), i+1) else (i_max, v_max, i+1))
  in let i_max, _, _ = Array.fold_left aux (-1, -1, 0)  t
  in i_max
```

R. 2-10 Écrire une fonction prenant en argument un tableau d'entiers `t` et calculant le tableau des sommes cumulées du tableau `t`. Ainsi la case d'indice `i` du tableau résultat doit contenir `t.(0) + t.(1) + ... + t.(i)`. On fournira une implémentation en $\mathcal{O}(n)$.

Solution

O-tableau-sommecum

```
(** Calcule le tableau des sommes cumulées de [t]. *)
let tab_somme_cumulee (t: int array): int array =
  let n = Array.length t in
  if n = 0 then [||]
  else
    begin
      let rep = Array.make n t.(0) in
      for i = 1 to (n-1) do
        rep.(i) <- t.(i) + rep.(i-1)
      done;
      rep
    end
```

R. 2-11 Définir une fonction prenant en argument un tableau de listes d'entiers `t` et le modifiant

en ajoutant à chaque liste l'entier indice de la case du tableau dans laquelle se trouve la liste. On itérera sur le tableau au moyen d'une boucle `for`

Solution

0-tableau-ajouteindliste-0

```
(** Modifie le tableau [t] de sorte que les listes qu'il contient soit
    préfixée par l'indice de la case du tableau dans laquelle elle se
    trouve. *)
let ajoute_indice_listes_bis (t: int list array): unit =
  let n = Array.length t in
  for i = 0 to (n-1) do
    t.(i) <- i :: t.(i)
  done
```

R. 2-12 Définir une fonction prenant en argument un tableau de listes d'entiers `t` et un entier `x` et retournant un tableau de listes d'entiers obtenu par ajout de l'entier en tête de chacune des listes se trouvant dans le tableau. On itérera sur le tableau au moyen d'un `Array.map`.

Solution

0-tableau-ajouteindliste-1

```
(** Modifie le tableau [t] de sorte à préfixer chacune des listes qu'il
    contient par l'entier [x]. *)
let ajoute_indice_listes (t: int list array) (x: int): int list array =
  Array.map (fun l -> x :: l) t
```

R. 2-13 Écrire une fonction prenant en argument un tableau d'entiers t, contenant des valeurs entières dans un intervalle $[\![0, m-1]\!]$ où m vous est passé en argument, et calculant l'entier de $[\![0, m-1]\!]$ ayant le plus d'occurrences dans le tableau. On s'efforcera d'itérer sur les tableaux aux moyens de fonctionnelles et non de boucle `for`/`while`. On fournira une implémentation en $\mathcal{O}(n+m)$.

Solution

0-tableau-elemmajoritaire

```
(** Calcule l'entier du tableau [t] ayant le nombre maximal d'occurrence
    dans [t].

    Hypothèse : les entiers contenus dans le tableau [t] sont tous dans
    l'intervalle entier \{0, 1, ... m-1\} *)
let element_majoritaire (t: int array) (m: int): int =
  let tab_occurs = Array.make m 0 in
  Array.iter (fun x -> tab_occurs.(x) <- tab_occurs.(x) + 1) t;
  let aux = (fun (i_max, v_max, i) x ->
      if x > v_max then (i, x, i+1)
      else (i_max, v_max, i+1)) in
  let i_max, _, _  =  Array.fold_left aux (-1, -1, 0) tab_occurs in
  i_max
```

2 Tris

R. 2-14 Écrire une fonction permettant de trier un tableau au moyen de l'algorithme du tri à bulles♣.

♣. https://en.wikipedia.org/wiki/Bubble_sort

Solution

O-tableau-tribulle

```
(** Échange le contenu des cases d'indices [i] et [j] du tableau [t]. *)
let echange (t: 'a array) (i: int) (j: int): unit =
  let tmp = t.(i) in
  t.(i) <- t.(j);
  t.(j) <- tmp

(** Trie le tableau [t] au moyen d'un tri bulle. *)
let tri_bulle (t: 'a array): unit =
  let n = Array.length t in
  (*inv : les i derniers élém. de t sont les i + grands et sont triés*)
  for i = 0 to (n-1) do
    (*inv : t[j] est le max des t[k] pour k in [0..j] *)
    for j = 0 to (n-2-i) do
      if t.(j) > t.(j+1) then echange t j (j+1)
    done
  done
```

R. 2-15 Écrire une fonction qui prend en argument un tableau `t` et deux indices `g` et `d` valides dans `t`, et qui décale d'une case vers la droite, tous les éléments d'indices dans $[\![\mathtt{g}, \mathtt{d}-1]\!]$. Le contenu de la case d'indice `d` est écrasé par cette opréation

Solution

O-tableau-decale

```
(** Décale d'une case vers la droite les cases d'indices \[|g, d-1|\], le
    contenu de la case d est donc écrasé *)
let decale (t: 'a array) (g: int) (d: int): unit =
  for i = d downto g+1 do
    t.(i) <- t.(i-1)
  done
```

R. 2-16 Écrire une fonction permettant de trier un tableau au moyen de l'algorithme du tri par insertion♣.

Solution

O-tableau-triinsertion

```
(** On suppose que le tableau t\[0..i-1\] est trié, on insère [t.(i)] dans
    le sous-tableau \[t\[0..i\] de sorte à propager la propriété triée *)
let insertion (t: 'a array) (i: int): unit =
  let a_inserer = t.(i) in
  let j = ref i in
  while !j >= 1 && t.(!j-1) > a_inserer do
    j := !j - 1;
  done;                          (* on a trouvé la position d'insertion *)
  decale t (!j) i;               (* on fait de la place en décalant *)
  t.(!j) <- a_inserer            (* on insère *)

(** Trie le tableau [t] au moyen d'un tri par insertion. *)
let tri_insertion (t: 'a array): unit =
  let n = Array.length t in
  (* inv : les i premiers élem. de t sont triés *)
  for i = 0 to (n-1) do
    insertion t i
```

♣. https://en.wikipedia.org/wiki/Insertion_sort

```
  done
```

NB : L'utilisation de la fonction `decale` dans la fonction `insertion` permet de rendre le code plus facile à lire, mais n'est pas nécessaire : en se déplaçant vers la gauche pour chercher la position d'insertion, on pourrait décaler les éléments que l'on croise.

R. 2-17 Écrire une fonction permettant de trier un tableau au moyen de l'algorithme du tri par sélection♣.

Solution

0-tableau-triselection

```
(** Calcule l'indice d'un minimum du tableau t\[g, ...\]. *)
let find_idx_min (t: 'a array) (g: int): int =
  let n = Array.length t in
  let idx_min = ref     g   in
  let val_min = ref (t.(g)) in
  for i = g+1 to (n-1) do
     if t.(i) < !val_min then
       begin
         val_min := t.(i) ;
         idx_min := i
       end
  done;
 !idx_min

(** Trie le tableau [t] au moyen d'un tri par sélection. *)
let tri_selection (t: 'a array): unit =
  let n = Array.length t in
  (* inv : les i premiers élem. de t sont les i + petits et sont triés *)
  for i = 0 to (n-1) do
    let j = find_idx_min t i in
    echange t i j
  done
```

R. 2-18 Écrire une fonction partition prenant en argument un tableau `t`, deux indices `d` (début) et `f` (fin) et `ip` un indice du sous-tableau $\texttt{t}[\texttt{d}, \texttt{f}]$ et qui permute $\texttt{t}[\texttt{d}, \texttt{f}]$ de manière à placer d'abord les éléments inférieurs à la valeur pivot `t.(ip)`, puis cette valeur pivot, à l'indice `q` qu'elle retournera, et enfin ceux strictement supérieurs à cette valeur.

Solution

0-tableau-partitionne

```
(** Permute les valeurs de [t[d..f]] autour de la valeur pivot [v = t.(p)]
    pour que [v] soit finalement enregistrée à l'indice [j] retourné et que
    [t.(i) <= v] pour [i] dans [\[d, j\[] et [not t.(i) <= v] pour [i] dans
    [\]j, f\]] *)
let partitionne (t: 'a array) (d: int) (f: int) (p: int): int =
  echange t p f;             (* on place le pivot à la fin du tableau    *)
  (* inv : [d..j[ = {k \in [d..i[ | t[k] <= t[f] } *)
  let j = ref d in
  for i = d to f-1 do
    if t.(i) < t.(f) then
      (echange t (!j) i; incr j)
  done;
  echange t (!j) f;
  !j
```

♣. https://en.wikipedia.org/wiki/Selection_sort

R. 2-19 Écrire une fonction permettant de trier un tableau au moyen de l'algorithme du tri rapide.

Solution

O-tableau-trirapide

On propose ici le tri rapide probabiliste. Une implémentation où le pivot est choisi de manière déterministe et arbitraire (par exemple dans la première case) conenait aussi vu l'énoncé.

```
(** Trie le tableau [t] entre les indices [d] et [f]. *)
let rec tri_rapide_aux (t: 'a array) (d: int) (f: int): unit =
  if d < f then
    let p = d + Random.int (f - d + 1) in (* on tire au hasard p *)
    let g = partitionne t d f p in        (* on partitionne pour p *)
    tri_rapide_aux t d (g-1);             (* on trie rec. à gauche *)
    tri_rapide_aux t (g+1) f              (* on trie rec. à droite *)

(** Trie le tableau [t] par l'algorithme du tri rapide.  *)
let tri_rapide (t: 'a array) : unit =
  tri_rapide_aux t 0 (Array.length t)
```

R. 2-20 Écrire une fonction permettant de calculer la médiane d'un tableau d'entiers sans d'abord trier ce tableau. On s'attend à une complexité en $\mathcal{O}(n^2)$.

Solution

O-tableau-mediane

```
(** Calcule la médiane du tableau [t]. *)
let mediane_bis (t: 'a array): 'a =
  let n = Array.length t in
  let est_mediane (x: 'a): bool =
    (* compte le nombre d'éléments strictement inférieurs à x *)
    let res_inf = ref 0 in
    (* compte le nombre d'éléments égaux à x *)
    let res_ega = ref 0 in
    for i = 0 to (n-1) do
      if t.(i) < x then res_inf := !res_inf + 1;
      if t.(i) = x then res_ega := !res_ega + 1
    done;
    !res_inf <= n/2 && !res_inf + !res_ega > n/2
  in
  let i    = ref 0     in
  while not (est_mediane t.(!i)) do
    incr i
  done;
  t.(!i)
```

3 Chaînes de caractères

R. 2-21 Écrire une fonction `string_depuis_char_list : char list -> string` prenant en paramètre une liste de caractères et calculant la chaîne de caractères constituées des caractères de cette liste, dans le même ordre. Par exemple `(string_depuis_char_list ['a'; 'b'; 'c'] = "abc")`.

Solution

O-tableau-stringofcharlist

```
(** Transforme la liste de caractères l en une chaîne de caractères. *)
let rec string_depuis_char_list (l: char list): string =
```

```
  match l with
  | []      -> ""
  | c :: l' -> (String.make 1 c) ^ (string_depuis_char_list l')
```

R. 2-22 Écrire une fonction `char_list_depuis_string : string -> char list` prenant en paramètre une chaîne de caractères et calculant la liste des caractères la constituant, dans le même ordre. Par exemple `(string_depuis_char_list "abc" = ['a'; 'b'; 'c'])`.

Solution

0-tableau-charlistofstring

```
(** Transforme la liste de caractères l en une chaîne de caractères. *)
let char_list_depuis_string (s: string): char list =
  let n = String.length s in
  (** Retourne la liste des caractères de la chaîne de caractères
      s[i..n-1], le résultat est accumulé, à l'envers, dans acc. *)
  let rec aux (i: int) (acc: char list): char list =
    if i = n then List.rev acc
    else aux (i + 1) (s.[i] :: acc)
  in aux 0 []
```

R. 2-23 Écrire une fonction `int_depuis_string : string -> int option` prenant en paramètre une chaîne de caractères et calculant, s'il existe, l'entier représenté par cette chaîne de caractères. Par exemple `(int_depuis_string "0123" = Some 123)` mais `(int_depuis_string "01a23" = None)`.

Solution

0-tableau-intofstring

```
(** Retourne l'entier représenté par la chaîne de caractère s, s'il
    existe. *)
let int_depuis_string (s: string): int option =
  let n = String.length s in
  (** Retourne l'entier associé à la chaîne de caractères s[i..n-1], le
      résultat est accumulé, dans acc. *)
  let rec aux (i: int) (acc: int): int option =
    if i = n then Some(acc)
    else if '0' <= s.[i] && s.[i] <= '9' then
      aux (i + 1) (acc * 10 + (int_of_char s.[i]) - (int_of_char '0'))
    else None
  in aux 0 0
```

4 Autres utilisations classiques des tableaux en algorithmique

R. 2-24 Écrire une fonction prenant en argument un tableau d'entiers, triés par ordre croissant, de taille n, un entier p et testant si l'entier p apparaît dans le tableau. L'algorithme devra être récursif et ne pas manipuler de boucles `while`/`for`. On assurera une complexité en $\mathcal{O}(\log(n))$.

Solution

0-tableau-dichotomierec

```
(** Teste l'appartenance de [p] dans le tableau [t], supposé trié. *)
let dichotomie_rec (t: int array) (p: int): bool =
  let rec aux (t: int array) (g: int) (d: int) : bool = (* [|g, d|] *)
    (g <= d) &&
```

```
    (let m = (g + d) / 2 in
     (t.(m) = p) ||
     ((t.(m) > p) && aux t g (m-1)) ||
     ((t.(m) < p) && aux t (m+1) d)
    )
  in aux t 0 ((Array.length t) -1)
```

R. 2-25 Écrire une fonction prenant en argument un tableau d'entiers, triés par ordre croissant, de taille n, un entier p et testant si l'entier p apparaît dans le tableau. L'algorithme utilisé ne devra pas être récursif. On assurera une complexité en $\mathcal{O}(\log(n))$.

Solution

0-tableau-dichotomiewhile

```
(** Teste l'appartenance de [p] dans le tableau [t], supposé trié. *)
let dichotomie_while (t: int array) (p: int): bool =
  let n = Array.length t in
  let g = ref 0 in
  let d = ref (n-1) in
  let found = ref false in
  while (!g <= !d && not (!found)) do
    let m = (!g + !d) / 2 in
    if t.(m) = p then found := true
    else if t.(m) > p then d := m-1
    else g := m + 1
  done;
  !found
```

R. 2-26 Écrire une fonction prenant en argument un tableau de booléens représentant un entier écrit en base 2 (les bits de poids faibles sont à droite) et modifiant le tableau pour qu'il représente l'entier suivant. On lèvera l'exception `Invalid_argument "dernier"` s'il n'est pas possible d'incrémenter l'entier.

Solution

0-tableau-next

```
(** Modifie le tableau [t] de sorte que si [t] avant l'appel représente la
    décomposition en binaire d'un entier n, alors [t] après l'appel
    représente la décomposition en binaire de l'entier n+1. Si ce n'est pas
    possible, pour des raisons de retenue, la fonction lève l'exception
    [Invalid_argument] *)
let next (t: bool array): unit=
  let n   = Array.length t in
  let idx = ref (n-1) in
  let ret = ref true in
  while (!ret && !idx >= 0) do
    if not t.(!idx) then ret := false;
    t.(!idx) <- not t.(!idx);
    idx := !idx - 1
  done;
  if !ret then raise (Invalid_argument "dernier")
```

R. 2-27 On représente une permutation de $[\![0, n-1]\!]$ par un tableau d'entiers t tel que l'image de tout entier i de $[\![0, n-1]\!]$ par la permutation est `t.(i)`. On représente un cycle (au sens des permutations) par la liste des éléments de son support, dans l'ordre du cycle, ainsi `[0; 1; 2]` est la permutation qui envoie 0 sur 1, 1 sur 2 et 2 sur 0. On rappelle qu'une permutation se décompose de manière unique en un produit de cycles à supports disjoints. On représente un produit de cycles

à supports disjoints comme un liste des cycles qui le composent. Leurs supports étant disjoints, ces cycles commutent, ainsi l'ordre dans la liste n'importe pas. Donner une fonction calculant la décomposition d'une permutation en produit de cycles à supports disjoints.

Solution

0-tableau-decomppdtcycles

```
(** Calcule la décomposition en produit de cycles à supports disjoints de
    la permutation [p]. *)
let decomposition_produit_cycles (p: int array): int list list =
  let n = Array.length p in
  let visited = Array.make n false in
  let rec trouve_cycle (p: int array) (i: int) (cur: int) =
    visited.(cur) <- true ;
    if p.(cur) = i then [cur]
    else cur :: (trouve_cycle p i p.(cur))
  in
  let rec aux (i: int) (res: int list list): int list list =
    if i >= n then res
    else if visited.(i) then aux (i+1) res
    else aux (i+1) ((trouve_cycle p i i) :: res)
  in
  aux 0 []
```

R. 2-28 Écrire une fonction prenant en argument un tableau `t` de taille n et une valeur par défaut `x` de même type que celui des éléments du tableau et retournant un tableau de taille $2n$ dont les n premiers éléments sont ceux de `t` et les n derniers contiennent la valeur par défaut.

Solution

0-tableau-double

```
(** Retourne un nouveau tableau de taille [2n] (où [n] est la taille du
    tableau [t] passé en argument) dont les [n] premières cases sont celles
    du tableau [t] et les [n] suivantes sont complétées avec la valeur
    [default]. *)
let double (t: 'a array) (default: 'a): 'a array =
  let n = Array.length t in
  let rep = Array.make (2 * n) default in
  for i = 0 to (n-1) do
    rep.(i) <- t.(i)
  done;
  rep
```

R. 2-29 Implémenter une structure de table dynamique, c'est-à-dire une structure de tableau telle que l'ajout et la suppression d'éléments en fin de tableau ont un coût amorti constant.

Solution

0-tableau-tabdynamique

```
(************************************************)
(* Définition d'une structure de tableau dynamique *)
(************************************************)
type 'a tab_dyn = {
  mutable contenus : 'a array ; (* le tableau contenant les éléments *)
  mutable nb_elems : int ;      (* le nombre d'éléments dans contenus *)
}

(** Initialisation d'une structure de tableau dynamique. *)
let init () : 'a tab_dyn =
```

```
  {contenus = [||]; nb_elems = 0}

(** Ajoute l'élément [x] dans le tableau dynamique [t]. *)
let ajout (t: 'a tab_dyn) (x: 'a): unit =
  if t.nb_elems = 0 then
    begin
    t.contenus <- [|x|];
  end
  else if Array.length t.contenus = t.nb_elems then
    begin
      let new_t = double t.contenus x in
      t.contenus <- new_t
    end;
  t.contenus.(t.nb_elems) <- x;
  t.nb_elems <- t.nb_elems + 1

(** Supprime l'élément en fin de tableau dynamique [t]. Lève l'exception
    [Invalid_argument] si cela n'est pas possible. *)
let suppression (t: 'a tab_dyn): 'a =
  if t.nb_elems = 0 then
    raise (Invalid_argument "suppression dans table vide")
  else begin
    t.nb_elems <- t.nb_elems - 1;
    t.contenus.(t.nb_elems)
  end
```

Dans toute cette feuille de révision on suppose définis les types OCaml suivants, permettant respectivement la représentation d'arbres binaires non vides, d'arbres généraux non vides.

```
type 'a btree =
  | E                              (* arbre vide *)
  | N of 'a * 'a btree * 'a btree (* noeud *)

type 'a gtree =
  | GN of 'a * 'a gtree list
```

1 Pied à l'étrier

R. 3-1 Définir une fonction permettant de tester si un arbre binaire est vide.

Solution

0-arbre-estvide

```
(** Teste la vacuité de l'arbre [b]. *)
let est_vide (b: 'a btree): bool =
  match b with
  | N(_, _, _) -> false
  | E          -> true
```

R. 3-2 Définir une fonction calculant la hauteur d'un arbre binaire. On rappelle que la hauteur de l'arbre vide est -1.

Solution

0-arbre-hauteur

```
(** Retourne la hauteur de l'arbre [b]. *)
let rec hauteur (b: 'a btree): int =
  match b with
  | N(x, g, d) -> 1 + (max (hauteur g) (hauteur d))
  | E          -> -1
```

R. 3-3 Définir une fonction calculant la taille (*i.e.* le nombre de nœuds) d'un arbre général. On itérera sur l'arbre général au moyen de deux fonctions mutuellement récursives.

Solution

0-arbre-taille

```
(** Calcule le nombre de nœuds dans l'arbre général [a]. *)
let rec taille_ag (a: 'a gtree): int =
  match a with
  | GN(x, l) -> 1 + (taille_ag_list l)
(** Calcule la somme des tailles des arbres dans [l]. *)
and taille_ag_list (l: 'a gtree list): int =
  match l with
  | []      -> 0
  | x :: xl -> (taille_ag x) + (taille_ag_list xl)
```

R. 3-4 Définir une fonction calculant la taille (*i.e.* le nombre de nœuds) d'un arbre général. On itérera dans l'arbre général au moyen d'une unique fonction récursive et d'un `List.fold_left`.

Solution

O-arbre-tailleag-0

```
(** Calcule le nombre de nœuds dans l'arbre général [a]. *)
let rec taille_ag_bis (a: 'a gtree): int =
  let GN(_, l) = a in
  List.fold_left (fun accu elem -> accu + (taille_ag_bis elem)) 1 l
```

R. 3-5 Définir une fonction calculant la taille (*i.e.* le nombre de nœuds) d'un arbre général. On fournira une implémentation récursive terminale.

Solution

O-arbre-tailleag-1

```
(** Calcule le nombre de nœuds dans l'arbre général [a]. *)
let taille_ag (a: 'a gtree): int =
  let rec aux (todo : 'a gtree list) (res: int): int =
    match todo with
    | []                -> res
    | GN(x, gl) :: todo' -> aux (gl @ todo') (1 + res)
  in aux [a] 0
```

R. 3-6 Définir une fonction permettant de tester si une étiquette apparaît dans un arbre binaire.

Solution

O-arbre-mem

```
(** Teste l'appartenance de l'élément [x] à l'arbre [b]. *)
let rec mem (x: 'a) (b: 'a btree): bool =
  match b with
  | E          -> false
  | N(y, g, d) -> (x = y) || (mem x g) || (mem x d)
```

R. 3-7 Définir une fonction prenant en paramètre un arbre binaire étiqueté par des entiers et retournant `None` si l'arbre est vide et `Some(mi, ma)` où `mi` est l'étiquette minimale de l'arbre et `ma` l'étiquette maximale sinon. Cette fonction sera récursive et ne devra pas faire usage d'une fonction récursive auxiliaire.

Solution

O-arbre-minmax-0

```
(** Retourne le plus petit et le plus grand élément de l'arbre [b].
    Retourne [None] si l'arbre est vide. *)
let rec min_max (b: int btree): (int * int) option =
  match b with
  | E          -> None
  | N(y, g, d) ->
    begin
      match min_max g, min_max d with
      | None, None
        -> Some(y, y)
      | None, Some(mi, ma)
        -> Some (min mi y, max ma y)
      | Some(mi, ma), None
        -> Some (min mi y, max ma y)
      | Some(mi1, ma1), Some(mi2, ma2)
        -> Some(min mi1 (min y mi2), max ma1 (max y ma2))
    end
```

R. 3-8 Définir une fonction prenant en paramètre un arbre binaire étiqueté par des entiers et retournant `None` si l'arbre est vide et `Some(mi, ma)` où `mi` est l'étiquette minimale de l'arbre et `ma` l'étiquette maximale sinon. Cette fonction ne sera pas récursive, elle devra faire appel à une fonction récursive auxiliaire qui sera récursive terminale.

Solution

O-arbre-minmax-1

```
(** Retourne le plus petit et le plus grand élément de l'arbre [b].
    Retourne [None] si l'arbre est vide. *)
let min_max_bis (b: 'a btree): ('a * 'a) option =
  let rec aux (todos: 'a btree list) (mi: 'a) (ma: 'a): 'a * 'a =
    match todos with
    | []                -> (mi, ma)
    | N(y, g, d)::todos2 -> aux (g::d::todos2) (min y mi) (max y ma)
    | E :: todos2       -> aux todos2 mi ma
  in
  match b with
  | E -> None
  | N(y, g, d) -> let mi, ma = aux [g; d] y y in Some(mi, ma)
```

R. 3-9 Définir une fonction prenant en paramètres deux arbres binaires et testant si ceux-ci ont la même forme, c'est-à-dire si les deux arbres sont égaux lorsqu'on omet les valeurs des étiquettes.

Solution

O-arbre-memeforme

```
(** Teste si les arbres [b1] et [b2] ont même forme. *)
let rec meme_forme (b1: 'a btree) (b2: 'a btree): bool =
  match b1, b2 with
  | E, E                       -> true
  | N(_, g1, d1), N(_, g2, d2) -> (meme_forme g1 g2) && (meme_forme d1 d2)
  | _                          -> false
```

R. 3-10 Définir une fonction prenant en paramètre un arbre binaire étiqueté par des entiers et retournant `Some(p)` où `p` est la profondeur d'un nœud d'étiquette paire s'il en existe et `None` sinon. On s'efforcera d'interrompre la recherche dès que cela est possible, en utilisant un mécanisme d'exception.

Solution

O-arbre-rechprofetiqpaire

```
(** Retourne la profondeur d'un nœud d'étiquette paire. *)
let recherche_profondeur_etiquette_paire (b: int btree): int option =
  let exception Found of int in
  let rec aux (bb: int btree) (p: int): unit =
    match bb with
    | N(x, _, _) when x mod 2 = 0 -> raise (Found(p))
    | N(x, g, d)                  -> (aux g (p+1); aux d (p+1))
    | E                           -> ()
  in
  try (aux b 0; None) with
  | Found(p) -> Some(p)
```

2 Parcours d'arbres

R. 3-11 Définir une fonction calculant le parcours infixe d'un arbre binaire. On pourra fournir une implémentation de complexité quadratique.

Solution

0-arbre-infixe-0

```
(** Calcule le parcours infixe d'un arbre binaire. *)
let rec parcours_infixe (b: 'a btree): 'a list =
  match b with
  | E          -> []
  | N(x, g, d) -> (parcours_infixe g) @ [x] @ (parcours_infixe d)
```

R. 3-12 Définir une fonction calculant le parcours infixe d'un arbre binaire. On fournira une implémentation de complexité linéaire.

Solution

0-arbre-infixe-1

```
(** Calcule la parcours infixe d'un arbre binaire. La complexité de
    l'algorithme est linéaire. *)
let parcours_infixe_bis (b: 'a btree): 'a list =
  (** [aux todo res] parcours la liste d'arbres [todo], en parcourant
      chaque arbre de [todo] de manière infixe. [res] est l'accumulateur,
      en cours de fabrication. *)
  let rec aux (todo: 'a btree list) (res:'a list): 'a list =
    match todo with
    | []                -> List.rev res
    | E::reste          -> aux reste res
    | N(x, E, E)::reste -> aux reste (x::res)
    | N(x, g, d)::reste -> aux (g::(N(x, E, E))::d::reste) res
  in aux [b] []
```

R. 3-13 Définir une fonction prenant en paramètre un arbre binaire a de type `'a btree` et deux éléments x et y de type `'a` et calculant l'arbre binaire obtenu en remplaçant, dans a, toutes les occurences x par y.

Solution

0-arbre-remplace

```
(** Calcule l'arbre obtenu en remplaçant dans a toutes les occurences
    de x par y *)
let rec remplace (a: 'a btree) (x:'a) (y:'a) : 'a btree =
  match a with
  | E -> E
  | N(r, g, d) ->
      N((if r = x then y else r), remplace g x y, remplace d x y)
```

R. 3-14 Définir une fonction calculant la bijection vue en première année entre l'ensemble des listes d'arbres généralisés (`'a gtree list`) et les arbres binaires (`'a btree`). Définir ensuite sa réciproque.

Solution

0-arbre-bijection

```
(** Calcule la bijection canonique entre une forêt d'arbres généraux et un
```

```
    arbre binaire. *)
let rec bijection_1 (gl: 'a gtree list): 'a btree =
  match gl with
  | []                -> E
  | GN(x, gl2) :: gl3 -> N(x, bijection_1 gl2, bijection_1 gl3)

let rec bijection_2 (b: 'a btree): 'a gtree list =
  match b with
  | E          -> []
  | N(x, g, d) -> (GN(x, bijection_2 g)):: (bijection_2 d)
```

R. 3-15 Définir une fonction retournant un parcours en profondeur d'un arbre binaire. On doit fournir une implémentation récursive n'utilisant pas de fonction récursive auxiliaire. Cette implémentation peut être non récursive terminale, de complexité quadratique.

Solution

0-arbre-profondeur-0

```
(** Calcule un parcours en profondeur de l'arbre [b]. *)
let rec profondeur (b: 'a btree): 'a list =
  match b with
  | E          -> []
  | N(x, g, d) -> [x] @ (profondeur g) @ (profondeur d)
```

R. 3-16 Définir une fonction retournant un parcours en profondeur d'un arbre binaire. On doit fournir une implémentation récursive terminale de complexité linéaire.

Solution

0-arbre-profondeur-1

```
(** Calcule un parcours en profondeur de l'arbre [b]. *)
let profondeur2 (b: 'a btree): 'a list =
  let rec aux (todos: 'a btree list) (res: 'a list) : 'a list =
    match todos with
    | []                  -> List.rev res
    | E :: todos2         -> aux todos2 res
    | N(x, g, d) :: todos2 -> aux (g :: d :: todos2) (x :: res)
  in aux [b] []
```

R. 3-17 Définir une fonction retournant un parcours en profondeur d'un arbre binaire. On doit fournir une implémentation récursive terminale de complexité linéaire en utilisant le module `Stack`.

Solution

0-arbre-profondeur-2

```
(** Calcule un parcours en profondeur de l'arbre [b]. *)
let profondeur3 (b: 'a btree): 'a list =
  let todos = Stack.create () in
  Stack.push b todos;
  (** [aux res] est le coeur récursif du parcours en profondeur. Cette
      fonction utilise et modifie la pile mutable [todos] comme structure
      de mise en attente des sommets à traiter. *)
  let rec aux (res: 'a list) : 'a list =
    if Stack.is_empty todos then List.rev res
    else
      match Stack.pop todos with
      | E          -> aux res
      | N(x, g, d) -> ( Stack.push g todos; Stack.push d todos; aux (x :: res) )
```

```
  in aux []
```

R. 3-18 Définir une fonction retournant un parcours en largeur d'un arbre binaire (la liste de ses étiquettes, triées par profondeur, et à profondeur équivalente de gauche à droite). On doit fournir une implémentation récursive terminale de complexité linéaire en utilisant le module `Queue`.

Solution

O-arbre-largeur-0

```
(** Calcule un parcours en largeur de l'arbre [b]. *)
let largeur (b: 'a btree): 'a list =
  let todos = Queue.create () in
  Queue.push b todos;
  (** [aux res] est le coeur récursif du parcours en largeur. Cette
      fonction utilise et modifie la file mutable [todos] comme structure
      de mise en attente des sommets à traiter. *)
  let rec aux (res: 'a list) : 'a list =
    if Queue.is_empty todos then List.rev res
    else
      match Queue.pop todos with
      | E          -> aux res
      | N(x, g, d) ->
          Queue.push g todos; Queue.push d todos; aux (x :: res)
  in aux []
```

R. 3-19 Définir une fonction retournant un parcours en largeur d'un arbre binaire. On doit fournir une implémentation récursive terminale de complexité linéaire en utilisant deux listes : celle du niveau courant, celle du niveau suivant.

Solution

O-arbre-largeur-1

```
(** Calcule un parcours en largeur de l'arbre [b]. *)
let largeur2 (b: 'a btree): 'a list =
  (** [aux niveau_courant niveau_suivant res] est le coeur récursif du
      parcours en largeur.
      - [niveau_courant] est la liste des sous-arbres non encore traités
        qui se trouvent au niveau en cours de traitement.
      - [niveau_suivant] est la liste des sous-arbres du niveau suivant le
        niveau en cours de traitement.
      - [res] est la liste résultat du parcours, en cours de construction.
  *)
  let rec aux
      (niveau_courant: 'a btree list)
      (niveau_suivant: 'a btree list)
      (res: 'a list): 'a list =
    match niveau_courant with
    | [] when niveau_suivant = []   ->
      List.rev res
    | []                            ->
      aux (List.rev niveau_suivant) [] res
    | E :: niveau_courant2          ->
      aux niveau_courant2 niveau_suivant res
    | N(x, g, d) :: niveau_courant2 ->
      aux niveau_courant2 (d :: g :: niveau_suivant) (x :: res)
  in aux [b] [] []
```

3 Utilisations classiques en algorithmique

R. 3-20 Définir une fonction permettant de tester l'appartenance d'un élément dans un arbre binaire de recherche. On fournira une implémentation ayant une complexité linéaire en la hauteur de l'arbre considéré.

Solution

0-arbre-memabr

```
(** [mem_abr b x] Teste la présence de l'élément [x] dans l'ABR [b]. *)
let rec mem_abr (b: 'a btree) (x: 'a): bool =
  match b with
  | E          -> false
  | N(y, g, d) ->
    (x = y)
    || (y < x && (mem_abr d x))
    || (y > x && (mem_abr g x))
```

R. 3-21 Définir une fonction prenant en paramètre un arbre binaire étiqueté par des entiers et testant si l'arbre est de recherche, à savoir si pour tout nœud de l'arbre d'étiquette e, toute étiquette de son fils gauche est inférieure stricte à e, et toute étiquette de son fils droit est supérieur à e. On fournira une implémentation ayant une complexité en $\mathcal{O}(n^2)$.

Solution

0-arbre-estabr-0

```
(** [forall f b] teste si tous les nœuds de l'arbre [b] vérifient le
    prédicat [f]. *)
let rec forall (f: 'a -> bool) (b: 'a btree): bool =
  match b with
  | E          -> true
  | N(x, g, d) -> (f x) && (forall f g) && (forall f d)

(** [est_abr b] teste si l'arbre [b] est un arbre binaire de recherche. *)
let rec est_abr (b: 'a btree) =
  match b with
  | E -> true
  | N(x, g, d) ->
    forall (fun y -> y < x) g
    && forall (fun y -> y >= x) d
```

R. 3-22 Définir une fonction prenant en paramètre un arbre binaire étiqueté par des entiers et testant si l'arbre est de recherche, à savoir si pour tout nœud de l'arbre d'étiquette e, toute étiquette de son fils gauche est inférieure à e, et toute étiquette de son fils droit est supérieur à e. On fournira une implémentation ayant une complexité en $\mathcal{O}(n)$.

Solution

0-arbre-estabr-1

Une première version.

```
(** [est_abr b] Teste si l'arbre [b] est un ABR. *)
let est_abr (b: int btree): bool =
  (** [parcours b inf sup] parcours l'arbre [b] et vérifie que c'est un ABR
      dont les étiquettes sont comprises entre [inf] et [sup]. *)
  let rec parcours (b: int btree) (inf: int) (sup: int): bool =
    match b with
    | N(y, g, d) -> (inf <= y) && (y <= sup)
```

```
                    && (parcours g inf y) && (parcours d y sup)
    | E          -> true
  in parcours b min_int max_int
```

Une seconde version plus compliquée.

```
exception PasABR
(** [est_abr b] Teste si l'arbre [b] est un ABR. *)
let est_abr (b: 'a btree): bool =
  (** [aux b] vérifie que l'arbre [b] est un ABR et retourne :
      - [None] si l'arbre [b] est vide.
      - [Some(mi, ma)] si l'arbre [b] est non vide et [mi] est la plus
      petite étiquette de l'arbre [b] et [ma] est la plus grande étiquette
      de l'arbre [b].
      Si [b] n'est pas un ABR, [aux b] lève l'exception [PasABR]
  *)
  let rec aux (b: 'a btree): ('a * 'a) option =
    match b with
    | E          -> None
    | N(y, g, d) ->
      begin
        match aux g, aux d with
        | None, None                    ->
          Some(y, y)
        | None, Some(mi, ma)            ->
          if y > mi then raise PasABR
          else Some (min mi y, max ma y)
        | Some(mi, ma), None            ->
          if y <= ma then raise PasABR
          else Some (min mi y, max ma y)
        | Some(mi1, ma1), Some(mi2, ma2) ->
          if y <= ma1 || y > mi2 then raise PasABR
          else Some(min mi1 (min y mi2), max ma1 (max y ma2))
      end
  in
  try let _ = aux b in true
  with
  | PasABR -> false
```

R. 3-23 Définir une fonction prenant en paramètre un arbre binaire et testant si celui-ci est parfait c'est à dire que tout nœud a soit deux fils vides (autrement dit est une feuille), soit deux fils non vides et que toutes les feuilles de l'arbre ont même profondeur.

Solution

0-arbre-estparfait-0

```
(** [est_parfait b] teste si l'arbre [b] est parfait. *)
let rec est_parfait (b: 'a btree): bool =
  match b with
  | E          -> true
  | N(_, g, d) ->
    est_parfait g && est_parfait d
    && hauteur g = hauteur d
```

R. 3-24 Définir une fonction prenant en paramètre un arbre binaire et testant si celui-ci est parfait c'est à dire que tout nœud a soit deux fils vides (autrement dit est une feuille), soit deux fils non vides et que toutes les feuilles de l'arbre ont même profondeur. On fournira une implémentation ayant une complexité en $\mathcal{O}(n)$.

Solution

0-arbre-estparfait-1

```
(** [est_parfait b] teste si l'arbre [b] est parfait. *)
let est_parfait (b: 'a btree) : bool =
  (** [aux b] retourne [None] si l'arbre [b] n'est pas un arbre parfait et
      retourne [Some(h)] si [b] est un arbre parfait de hauteur [h]. *)
  let rec aux (b: 'a btree) : int option =
    match b with
    | E          -> Some(-1)
    | N(x, g, d) ->
      match aux g, aux d with
      | Some(hg), Some(hd) ->
        if (hg <> hd) then None
        else Some(hg + 1)
      | _ -> None
  in
  match aux b with
  | None -> false
  | _    -> true
```

R. 3-25 Implémenter une fonction de tri par arbre binaire de recherche : pour trier une liste d'éléments, on insère successivement ces éléments dans un arbre binaire de recherche, un parcours infixe de l'arbre binaire de recherche fourni alors un tri de la liste initiale.

Solution

0-arbre-triabr

```
(** [insertion_abr x t] ajoute l'étiquette [x] à l'arbre binaire de
    recherche [t]. *)
let rec insertion_abr (x: 'a) (t: 'a btree): 'a btree =
  match t with
  | E          -> N(x, E, E)
  | N(y, g, d) ->
    if x < y then N(y, insertion_abr x g, d)
    else          N(y, g, insertion_abr x d)

(** [tri l] trie la liste [l] *)
let tri (l: 'a list) =
  (** [insertion_liste t l] ajoute les étiquettes de la liste [l] à l'arbre
      [t]. *)
  let rec insertion_liste (t: 'a btree) (l: 'a list): 'a btree =
    match l with
    | []      -> t
    | x :: l' -> insertion_liste (insertion_abr x t) l'
  in
  parcours_infixe (insertion_liste E l)
```

R. 3-26 Implémenter une fonction de tri par tas : pour trier un tableau d'éléments, on insère successivement ces éléments dans une file de priorité implémentée par un tournoi complet, des extractions successives du maximum permettent alors d'obtenir un tri de la liste initiale.

Solution

0-arbre-tritas

```
(* Les fonctions suivantes calculent respectivement les indices du fils
   gauche et du fils droit du sommet d'indice i dans un arbre tournoi
   complet. *)
let fg (i: int): int   = (2 * (i + 1)) - 1
```

```
let fd (i: int): int   = (2 * (i + 1) + 1) - 1

(** [lit t taille i] retourne l'étiquette du nœud d'indice [i] dans l'arbre
    se trouvant dans les [taille] premières cases du tableau [t]. Si [i]
    n'est pas un nœud valide, la fonction retourne [None]. *)
let lit (t: 'a array) (taille: int) (i: int): 'a option =
  if 0 <= i && i < taille then Some t.(i)
  else None

(** Échange le contenu des cases d'indice [i] et [j] du tableau [t]. *)
let echange (t: 'a array) (i: int) (j: int): unit =
  let tmp = t.(i) in
  t.(i) <- t.(j);
  t.(j) <- tmp


(** [descend t taille i] descend l'étiquette du nœud se trouvant se
    trouvant en case [i]. Cette fonction suppose que le fils gauche et le
    fils droit du nœud d'indice [i] sont bien des arbres tournois
    parfaits.*)
let rec descend (t: 'a array) (taille: int) (i: int): unit =
  match lit t taille (fg i), lit t taille (fd i) with
  | None, Some x when t.(i) < x            ->
    echange t i (fd i); descend t taille (fd i)
  | Some x, None when t.(i) < x            ->
    echange t i (fg i); descend t taille (fg i)
  | Some x, Some y when x >= y && t.(i) < x ->
    echange t i (fg i); descend t taille (fg i)
  | Some x, Some y when y >= x && t.(i) < y ->
    echange t i (fd i); descend t taille (fd i)
  | _ -> ()


(** [extrait_et_supprime_max t taille] supprime la maximum de l'arbre
    tournoi se trouvant dans les [taille] premières cases du tableau [t].
    La fonction retourne la valeur du maximum ainsi supprimé. *)
let extrait_et_supprime_max (t: 'a array) (taille: int): int =
  let min = t.(0) in
  echange t 0 (taille - 1);
  descend t (taille - 1) 0;
  min


(** [initialise_tas t] initialise, en place dans le tableau [t], un arbre
    tournoi complet contenant les valeurs du tableau [t]. *)
let initialise_tas (t: 'a array): unit =
  let n = Array.length t in
  for i = (n - 1) downto 0 do
    descend t n i
  done


(** [tri_par_tas t] trie le tableau [t], en place. *)
let tri_par_tas (t: 'a array): unit =
  let n = Array.length t in
  initialise_tas t;
  for taille = n downto 1 do
    let mi = extrait_et_supprime_max t taille in
    t.(taille - 1) <- mi;
  done
```

1 Création de tableaux

Dans toutes les questions on utilise les types suivants.

```c
// struct pour les tableaux d'entiers munis de leur taille
struct tableau_s {
  int  taille ;
  int* tab    ;
};
typedef struct tableau_s tableau;
```

```c
// struct pour les tableaux de tableaux d'entiers munis de leur taille
struct tabtab_s {
  int      taille ;
  tableau* tab    ;
};
typedef struct tabtab_s tabtab;
```

```c
// struct pour les tableaux de booléens munis de leur taille
struct btableau_s {
  int   taille ;
  bool* tab    ;
};
typedef struct btableau_s btableau;
```

R. 4-1 Écrire une fonction prenant en paramètre un entier naturel non nul n et retournant un tableau de n booléens rempli de `false` à l'aide d'une boucle **`while`**.

Solution

C-tableau-tabfalse

```c
btableau genere_tab_false(int n) {
  /* Alloue et initialise un tableau de booléens de taille n contenant
     uniquement des false. */
  bool* tab = malloc(n * sizeof(bool));
  int  i    = 0 ;
  while (i < n) {
    tab[i] = false ;
    i = i + 1 ;
  }
  return (btableau) {.taille = n, .tab = tab} ;
}
```

R. 4-2 Écrire une fonction prenant en paramètre un entier naturel non nul n et retournant un tableau contenant les entiers de 1 à n à l'aide d'une boucle **`for`**.

Solution

C-tableau-tabn

```c
tableau genere_tab_n(int n) {
  /* Alloue et initialise un tableau d'entiers de taille n contenant les
     entiers de 1 à n */
```

```
  int* tab = malloc (n * sizeof(int));
  for (int i = 0 ; i < n ; i++) {tab[i] = i + 1;}
  return (tableau) {.taille = n, .tab = tab} ;
}
```

R. 4-3 Écrire une fonction prenant en paramètre un entier naturel non nul n, et deux entiers relatifs a et b tels que $a < b$ et retournant un tableau de n entiers aléatoires tirés uniformément dans $[\![a, b[\![$.

Solution

C-tableau-tabnalea

```
tableau genere_tab_n_alea(int n, int a, int b) {
  /* hyp : n > 0 && a < b.
   * Alloue et initialise un tableau d'entiers choisis au hasard
   * uniformément dans l'intervalle entiers [|a, b - 1|]. Initialiser la
   * graine de rand avant usage ! */
  int* tab = malloc (n * sizeof(int));
  int l = b - a;
  for (int i = 0 ; i < n ; i++) {
    tab[i] = rand() % l + a;
  }
  return (tableau) {.taille = n, .tab = tab} ;
}
```

2 Agrégation

R. 4-4 Écrire une fonction testant l'existence d'un `0` dans un tableau d'entiers parcouru par boucle `while`. La fonction doit retourner `true` s'il est possible de trouver la valeur `0` dans le tableau et `false` sinon.

Solution

C-tableau-existe0

```
bool existe0(tableau t) {
  /* Teste la présence d'un 0 dans le tableau t. */
  int i = 0;
  while (t.tab[i] != 0) {
    i = i + 1;
  }
  return (i < t.taille);
}
```

R. 4-5 Écrire une fonction calculant l'indice du premier `0` dans un tableau d'entiers parcouru par boucle `while` arrêtée dès que possible, sans utiliser de `break`, ni de `return` dans la boucle. La fonction devra retourner `-1` si le tableau ne contient aucun `0`.

Solution

C-tableau-find0-0

```
int find0_v0(tableau t) {
  /* Calcule l'indice du premier 0 dans le tableau t. */
  int i = 0;
  while (t.tab[i] != 0) {
    i = i + 1;
  }
```

```
  if (i == t.taille) {
    return -1;
  } else {
    return i;
  }
}
```

R. 4-6 Écrire une fonction calculant l'indice du premier `0` dans un tableau d'entiers parcouru par boucle `while` arrêtée dès que possible grâce à une instruction `return` dans la boucle. La fonction devra retourner `-1` si le tableau ne contient aucun `0`.

Solution

C-tableau-find0-1

```
int find0_v1(tableau t) {
  /* Calcule l'indice du premier 0 dans le tableau t. */
  int i = 0;
  while (i < t.taille) {
    if (t.tab[i] == 0) { return i; }
    i = i + 1;
  }
  return -1;
}
```

R. 4-7 Écrire une fonction calculant l'indice du premier `0` dans un tableau d'entiers parcouru par boucle `for` arrêtée dès que possible. La fonction devra retourner `-1` si le tableau ne contient aucun `0`.

Solution

C-tableau-find0-2

```
int find0_v2(tableau t) {
  /* Calcule l'indice du premier 0 dans le tableau t. */
  for (int i = 0 ; i < t.taille ; i ++) {
    if (t.tab[i] == 0) { return i; }
  }
  return -1;
}
```

R. 4-8 Écrire une fonction calculant l'indice du dernier `0` dans un tableau d'entiers parcouru par boucle `while` arrêtée dès que possible grâce à une instruction `return` dans la boucle. La fonction devra retourner `-1` si le tableau ne contient aucun `0`.

Solution

C-tableau-find0-3

```
int find0_v3(tableau t) {
  /* Calcule l'indice du dernier 0 dans le tableau t. */
  int i = t.taille-1;
  while (i >= 0) {
    if (t.tab[i] == 0) { return i; }
    i = i - 1;
  }
  return -1;
}
```

R. 4-9 Écrire une fonction prenant en paramètre un tableau d'entiers et calculant la somme des éléments du tableau. On fournira une implémentation au moyen d'une boucle `for`.

Solution

C-tableau-somme

```c
int somme(tableau t) {
  /* Calcule la somme des éléments du tableau t. */
  int res = 0 ;
  for (int i = 0 ; i < t.taille ; i ++) {
    res = res + t.tab[i];
  }
  return res;
}
```

R. 4-10 Écrire une fonction calculant l'indice du minimum dans un tableau d'entiers non vide parcouru par une boucle `for`. En cas d'ambiguïté, on retournera le plus petit indice du minimum.

Solution

C-tableau-argmin

```c
int argminimum(tableau t) {
  /* Calcule l'indice du minimum du tableau t. */
  if (t.taille == 0) {return -1;}
  else {
    int vmin = t.tab[0] ;
    int imin = 0 ;
    for (int i = 1 ; i < t.taille ; i ++) {
      if (t.tab[i] < vmin) {
        vmin = t.tab[i];
        imin = i;
      }
    }
    return imin;
  }
}
```

R. 4-11 Écrire une fonction calculant la conjonction d'un tableau de booléens parcouru par une boucle `while`.

Solution

C-tableau-conjonction

```c
bool conjonction(btableau t) {
  /* Calcule la conjonction des cases du tableau t. */
  bool res = true;
  for (int i = 0 ; i < t.taille ; i ++) {
    res = res && t.tab[i];
  }
  return res;
}
```

R. 4-12 Écrire une fonction prenant en paramètre un tableau d'entiers à valeurs dans $[\![0, m-1]\!]$ où m est donné en paramètre, et calculant l'entier de $[\![0, m-1]\!]$ ayant le plus d'occurrences dans le tableau. On fournira une implémentation en $\mathcal{O}(n+m)$.

Solution

C-tableau-maxoccur

```c
int max_occurrences(tableau t, int m) {
  /* Retourne l'entier de [|0, m-1|] ayant le plus d'occurrences dans le
   * tableau t d'entiers à valeurs dans [|0, m-1|]. */
  int* nb_occur = malloc(m * sizeof(int));
  for (int i = 0 ; i < m ; i ++) {
    nb_occur[i] = 0;
  }
  for (int i = 0 ; i < t.taille ; i ++) {
    nb_occur[t.tab[i]] = nb_occur[t.tab[i]] + 1;
  }
  int vmax = nb_occur[0] ;
  int imax = 0 ;
  for (int i = 1 ; i < m ; i ++) {
    if (nb_occur[i] > vmax) {
      vmax = nb_occur[i];
      imax = i;
    }
  }
  return imax;
}
```

3 Tableaux à partir d'un tableau

R. 4-13 Écrire une fonction qui copie un tableau d'entiers.

Solution

C-tableau-copie

```c
tableau copie_tableau (tableau t){
  /* Retourne une copie du tableau t. */
  int* nv_tab = malloc(t.taille * sizeof(int));
  for(int i = 0 ; i < t.taille ; i++){
    nv_tab[i] = t.tab[i];
  }
  return (tableau) {.taille = t.taille, .tab = nv_tab};
}
```

R. 4-14 Écrire une fonction prenant en paramètre `t` un tableau d'entiers de taille n et un entier par défaut `x` et retournant un tableau de taille $2n$ dont les n premiers éléments sont ceux de `t` et les n derniers contiennent la valeur par défaut.

Solution

C-tableau-copiedouble

```c
tableau copie_double_tableau(tableau t, int val){
  /* Renvoie un tableau de taille double de t, rempli par les élements de t
   * puis par val. */
  tableau res = (tableau) {
    .taille = 2 * t.taille,
    .tab    = malloc(res.taille * sizeof(int))};
  for(int i = 0; i < t.taille; i++){
    res.tab[i] = t.tab[i];
  }
  for(int i = t.taille; i < res.taille; i++){
```

```c
    res.tab[i] = val;
  }
  return res;
}
```

R. 4-15 Écrire une fonction prenant en paramètre un tableau d'entiers `t` et retournant le tableau miroir de `t`.

Solution

C-tableau-miroir

```c
tableau miroir(tableau t){
  /* Renvoie le miroir de t. */
  tableau res = (tableau) {
    .taille = t.taille,
    .tab    = malloc (res.taille * sizeof(int))
  };
  int i = 0;
  int j = t.taille - 1;
  while (i < t.taille) {
    res.tab[i] = t.tab[j];
    i++; j--;
  }
  return res;
}
```

R. 4-16 Écrire une fonction prenant en paramètre un tableau d'entiers `tailles` et retournant `t` un tableau de tableaux d'entiers initialisé à 0, de même taille que le tableau `tailles`, et tel que pour tout indice `i` valide pour ces tableaux, `t[i]` est de taille `tailles[i]`.

Solution

C-tableau-tabtabvide

```c
tabtab tabtab_vide(tableau sizes){
  /* Crée un tableau de tableaux de taille indiquées dans sizes. */
  tabtab res;
  res.taille = sizes.taille;
  res.tab = malloc(res.taille * sizeof(tableau));
  for(int i = 0; i < res.taille; i++){
    res.tab[i].taille = sizes.tab[i];
    res.tab[i].tab = malloc (sizes.tab[i] * sizeof(int));
    for(int j = 0; j < res.tab[i].taille; j++){
      res.tab[i].tab[j] = 0;
    }
  }
  return res;
}
```

R. 4-17 Écrire une fonction prenant en paramètre un tableau d'entiers `t` et retournant le tableau des sommes cumulées du tableau `t`. Ainsi la case d'indice `i` du tableau résultat doit contenir `t[0] + t[1] + ... + t[i]`. On fournira une implémentation de complexité linéaire en la taille du tableau.

Solution

C-tableau-sommecumulee

```c
tableau sommes_cumulees(tableau t){
  /* Renvoie le tableau des sommes cumulées de t */
```

```c
  tableau res;
  res.taille = t.taille;
  res.tab = malloc (res.taille * sizeof(int));
  res.tab[0] = t.tab[0];
  for(int i = 1; i < t.taille; i++){
    res.tab[i] = res.tab[i-1] + t.tab[i];
  }
  return res;
}
```

R. 4-18 Écrire une fonction qui prend en paramètres deux tableaux d'entiers triés et qui crée le tableau trié obtenu en interclassant les éléments des deux tableaux. Le nombre d'occurrence de chaque élément dans le tableau résultat est la somme de ceux dans chacun des tableaux en paramètre. Cette fonction devra être en complexité linéaire, c'est-à-dire en $\mathcal{O}(n_1 + n_2)$ où n_1 et n_2 sont respectivement les tailles des deux tableaux pris en paramètre.

Solution

C-tableau-interclassement

```c
tableau interclassement(tableau t1, tableau t2){
  /* hyp : t1 et t2 sont triés */
  /* Renvoie le tableau trié obtenu par fusion de t1 et t2 */

  tableau res = {
    .taille = t1.taille + t2.taille,
    .tab    = malloc((t1.taille + t2.taille) * sizeof(int))
  };
  int i = 0, j = 0, k = 0; //curseurs dans t1,t2, res respectivement
  while(i < t1.taille && j < t2.taille){ //inv : k=i+j
    if(t1.tab[i]<= t2.tab[j]) {
      res.tab[k] = t1.tab[i];
      i++; k++;
    } else {
      res.tab[k] = t2.tab[j];
      j++; k++;
    }
  }
  while(i < t1.taille){
    res.tab[k] = t1.tab[i];
    i++; k++;
  }
  while(j < t2.taille){
    res.tab[k] = t2.tab[j];
    j++; k++;
  }
  return res;
}
```

R. 4-19 Écrire une fonction permettant de distinguer les sous-séquences (non vides) croissantes maximales pour l'extension à gauche d'un tableau d'entiers non vide en indiquant où elles commencent, c'est-à-dire retournant le tableau trié des indices où commencent ces sous-séquences, muni de sa taille, *i.e.* le nombre de telles sous-séquences. Par exemple pour **{1, 4, 5, 8, 7, 5, 2, 3, 4, 9, 3, 3, 3, 5, 5, 7, 2}**, le tableau des indices de découpage est **{0, 4, 5, 6, 10, 16}** correspondant aux 6 sous-séquences **{1, 4, 5, 8}**, **{7}**, **{5}**, **{2, 3, 4, 9}**, **{3, 3, 3, 5, 5, 7}** et **{2}**. Cette fonction pourra allouer un tableau temporaire, mais ne devra parcourir qu'une fois de tableau pris en paramètre.

Solution

C-tableau-sssseqcroissante-0

Version avec un seul parcours du tableau pris en entrée.

```
tableau sous_sequences_croissantes(tableau t) {
  /* renvoie le tableau des indices où commence une sous-séquence
     croissante maximale dans tab */
  int* temp = (int*) malloc(t.taille * sizeof(int));
  temp[0] = 0;
  int cpt = 1;
  int i = 1;
  while (i < t.taille){
    if(t.tab[i] < t.tab[i-1]){//t[i] ne prolonge pas la séquence en cours
      temp[cpt] = i;          //donc une nouvelle séquence commence en i
      cpt++;
    }
    i++;
  }
  tableau res;
  res.taille = cpt;
  res.tab = (int*) malloc(cpt * sizeof(int));
  for(int j = 0; j < cpt; j++){
    res.tab[j] = temp[j];
  }
  free(temp);
  return res;
}
```

R. 4-20 Écrire une fonction permettant de distinguer les sous-séquences (non vides) croissantes maximales pour l'extension à gauche d'un tableau d'entiers non vide en indiquant où elles commencent, c'est-à-dire retournant le tableau trié des indices où commencent ces sous-séquences, muni de sa taille, *i.e.* le nombre de telles sous-séquences. Par exemple pour **{1, 4, 5, 8, 7, 5, 2, 3, 4, 9, 3, 3, 3, 5, 5, 7, 2}**, le tableau des indices de découpage est **{0, 4, 5, 6, 10, 16}** correspondant aux 6 sous-séquences **{1, 4, 5, 8}**, **{7}**, **{5}**, **{2, 3, 4, 9}**, **{3, 3, 3, 5, 5, 7}** et **{2}**. Cette fonction pourra parcourir plusieurs fois de tableau pris en paramètre, mais ne devra pas allouer d'espace mémoire en dehors de celui nécessaire pour enregistrer le tableau résultat.

Solution

C-tableau-sssseqcroissante-1

Version sans allocation de tableau temporaire.

```
tableau sous_sequences_croissantes_bis (tableau t){
  /* renvoie le tableau des indices où commence une sous-séquence
     croissante maximale dans tab */
  int cpt = 1; //on compte 1 pour la séquence qui commence à t[0]
  int i = 1;   //on regarde les nouvelles valeurs, à partir de t[1]
  while (i < t.taille){
    if(t.tab[i] < t.tab[i-1]){//t[i] ne prolonge pas la séquence en cours
      cpt++;                  //donc une nouvelle séquence commence en i
    }
    i++;
  }
  tableau res;
  res.taille = cpt;
  res.tab = (int*) malloc(cpt * sizeof(int));
  res.tab[0] = 0;
  cpt = 1;
```

```
  i = 1;
  while (i < t.taille){
    if(t.tab[i] < t.tab[i-1]){//t[i] ne prolonge pas la séquence en cours
      res.tab[cpt] = i;       //donc une nouvelle séquence commence en i
      cpt++;
    }
    i++;
  }
  return res;
}
```

R. 4-21 Écrire une fonction permettant de découper un tableau d'entiers en sous-séquences non vides de somme inférieure à 10 maximales pour l'extension à gauche. Par exemple **{1, 2, 8, 2, 3, 4, 1}** est découpé en **{{1, 2}, {8, 2}, {2, 3, 4, 1}}**.

Solution

C-tableau-ssseqbornee

```
tabtab bin_packing_glouton (tableau t){
  /* Renvoie le tableau des sous-séquence de t de somme <= 10 maximales */
  int* tab_indices = (int*) malloc((t.taille + 1) * sizeof(int));
  tab_indices[0] = 0;
  int s = t.tab[0];
  int cpt = 1;              // nombre de sous-sequences

  int i = 1;
  while (i < t.taille){
    s = s + t.tab[i];
    if(s > 10){             // t[i] ne prolonge pas la séquence en cours
      tab_indices[cpt] = i; // donc une nouvelle séquence commence en i
      s = t.tab[i];
      cpt++;
    }
    i++;
  }
  tab_indices[cpt] = t.taille; //séquence factice après le dernier elem

  tabtab res;
  res.taille = cpt;
  res.tab = (tableau*) malloc(cpt*sizeof(tableau));
  i = 0;
  for(int j = 0; j < cpt; j++){
    int nj = tab_indices[j+1]-tab_indices[j];
    res.tab[j].taille = nj;
    res.tab[j].tab = (int*) malloc(nj*sizeof(int));
    for(int k = 0; k < nj; k++){
      res.tab[j].tab[k] = t.tab[i];
      i++;
    }
  }
  free(tab_indices);
  return res;
}
```

4 Tableaux représentant des matrices

Dans toutes les questions on utilise les types suivants.

```c
//struct pour les matrices dans un tableau unidimensionnel
struct matrice_unidimensionel_s {
  int nbl;  //nombre de lignes
  int nbc;  //nombre de colonnes
  int* tab; //tableau des coeffs de taille nbl*nbc
};
typedef struct matrice_unidimensionel_s matrice_u;
```

```c
//struct pour les matrices dans un tableau bidimensionnel
struct matrice_bidimensionel_s {
  int nbl;  //nombre de lignes
  int nbc;  //nombre de colonnes
  int** tab;//tableau de nbl tableaux de nbc coeffs
};
typedef struct matrice_bidimensionel_s matrice_b;
```

R. 4-22 En utilisant la représentation des matrices par tableau unidimensionnel qui consiste à mettre les lignes bout à bout, écrire une fonction prenant en paramètre une matrice M et les indices i et j d'un coefficient de M, et qui renvoie le coefficient $M_{i,j}$.

Solution

C-tableau-coeff-0

```c
int coeff (matrice_u m, int i, int j){
  /* hyp: 0<=i && i< m.nbl && 0<=j && j < m.nbc */
  return m.tab[i * m.nbl + j];
};
```

R. 4-23 En utilisant la représentation des matrices par tableau unidimensionnel qui consiste à mettre les lignes bout à bout, écrire une fonction prenant en paramètres une matrice M et les indices i et j d'un coefficient de M, et une valeur entière x, et qui donne au coefficient d'indice (i, j) dans cette matrice la valeur x.

Solution

C-tableau-setcoeff-0

```c
void set_coeff(matrice_u m, int i, int j, int val){
  /* hyp: 0<=i && i< m.nbl && 0<=j && j < m.nbc */
  m.tab[i*m.nbl + j] = val;
}
```

R. 4-24 En utilisant la représentation des matrices par tableau unidimensionnel qui consiste à mettre les lignes bout à bout, écrire une fonction prenant en paramètre un entier naturel non nul n et retournant un tableau d'entiers unidimensionnel représentant I_n, la matrice identité de dimensions $n \times n$.

Solution

C-tableauid-0

```c
matrice_u id (int n){
  /* Retourne la matrice identité. */
  matrice_u res;
  res.nbl = n;
  res.nbc = n;
  res.tab = malloc (sizeof(int) * res.nbl * res.nbc );
```

```
  for(int i=0; i < res.nbl; i++){
    for(int j=0; j < res.nbc; j++){
      set_coeff(res, i, j, 0);
    }
    set_coeff(res, i, i, 1);
  }
  return res;
}
```

R. 4-25 En utilisant la représentation des matrices par tableau de tableaux, écrire une fonction prenant en paramètre une matrice M et les indices i et j d'un coefficient de M, et qui renvoie le coefficient $M_{i,j}$.

Solution

C-tableau-coeff-1

```
int coeff_bis (matrice_b m, int i, int j){
  /* hyp: 0<=i && i< m.nbl && 0<=j && j < m.nbc */
  return m.tab[i][j];
};
```

R. 4-26 En utilisant la représentation des matrices par tableau de tableaux, écrire une fonction prenant en paramètres une matrice M et les indices i et j d'un coefficient de M, et une valeur entière x, et qui donne au coefficient d'indice (i, j) dans cette matrice la valeur x.

Solution

C-tableau-setcoeff-1

```
void set_coeff_bis(matrice_b m, int i, int j, int val){
  /* hyp: 0<=i && i< m.nbl && 0<=j && j < m.nbc */
  m.tab[i][j] = val;
}
```

R. 4-27 En utilisant la représentation des matrices par tableau de tableaux, écrire une fonction prenant en paramètre un entier naturel non nul n et retournant un tableau d'entiers unidimensionnel représentant I_n, la matrice identité de dimensions $n \times n$.

Solution

C-tableauid-1

```
matrice_b id_bis(int n) {
  /* Renvoie la matrice identité. */
  matrice_b res;
  res.nbl = n;
  res.nbc = n;
  res.tab = malloc (sizeof(int*) * res.nbl);
  for(int i = 0; i < res.nbl; i++){
    res.tab[i] = malloc (sizeof(int) * res.nbc);
    for(int j = 0; j < res.nbc; j++){
      set_coeff_bis(res, i, j, 0);
    }
    set_coeff_bis(res, i, i, 1);
  }
  return res;
}
```

5 Tris

Solution

Dans les 4 fonctions suivantes on utilise la fonction d'échange suivante.

```c
void echange(tableau t, int i, int j) {
  /* Échange, par effet de bord, le contenu des cases d'indice i et j du
   * tableau t. */
  int tmp = t.tab[i];
  t.tab[i] = t.tab[j];
  t.tab[j] = tmp;
  return ;
}
```

R. 4-28 Écrire une fonction permettant de trier un tableau au moyen de l'algorithme du tri à bulles♣.

Solution

C-tableau-tribulle

```c
void tri_bulle(tableau t) {
  /* Trie le tableau t au moyen du tri bulle. */
  for (int i = 0; i < t.taille; i ++) {
    for (int j = 0; j < t.taille - i - 1; j ++) {
      if (t.tab[j] > t.tab[j + 1]) {
        echange(t, j, j + 1);
      }
    }
  }
  return ;
}
```

R. 4-29 Écrire une fonction permettant de trier un tableau au moyen de l'algorithme du tri par insertion♡.

Solution

C-tableau-triinsertion

```c
void insere_trie(tableau t, int i) {
  /* Hyp : le tableau t[0 ... i-1] est trié par ordre croissant. */
  /* On insère l'élément t[i] au sous-tableau t[0 ... i-1], de manière
     à ce que le tableau résultat t[0 ... i] soit trié. */
  while (i - 1 >= 0 && t.tab[i] < t.tab[i - 1]) {
    echange(t, i, i - 1);
    i --;
  }
}

void tri_insertion(tableau t) {
  /* Trie le tableau t au moyen du tri insertion. */
  for (int i = 0; i < t.taille; i ++) {
    insere_trie(t, i);
  }
  return ;
}
```

♣. https://en.wikipedia.org/wiki/Bubble_sort
♡. https://en.wikipedia.org/wiki/Insertion_sort

R. 4-30 Écrire une fonction permettant de trier un tableau au moyen de l'algorithme du tri par sélection♣.

Solution

C-tableau-triselection

```c
int trouve_idx_min(tableau t, int i) {
  /* Trouve l'indice du minimum du sous-tableau t[i ... t.taille - 1]. */
  int id_min  = i;
  for (int j = i + 1 ; j < t.taille; j ++) {
    if (t.tab[j] < t.tab[id_min]) {
      id_min = j;
    }
  }
  return id_min;
}

void tri_selection(tableau t) {
  /* Trie le tableau t au moyen du tri par sélection. */
  for (int i = 0 ; i < t.taille ; i ++) {
    int j = trouve_idx_min(t, i);
    echange(t, i, j);
  }
}
```

R. 4-31 Écrire une fonction partition prenant en arguments un tableau `t` et `ip` un indice de `t` et qui modifie `t` en place de manière à placer d'abord les éléments inférieurs à la valeur pivot `t[ip]`, puis cette valeur pivot, à un indice q qu'elle retournera, et enfin ceux strictement supérieurs à cette valeur. En déduire une fonction permettant de trier un tableau au moyen de l'algorithme du tri rapide.

Solution

C-tableau-trirapide

```c
int partitionne(tableau t, int d, int f, int p) {
  /* Permute les valeurs de t[d..f] autour de la valeur pivot v = t.(p)
     pour que v soit finalement enregistrée à l'indice j retourné et que
     t.(i) <= v pour i dans [d, j[ et t.(i) > v pour i dans ]j, f] */
  echange(t, p, f);
  int j = d;
  for (int i = d; i < f; i ++) {
    if (t.tab[i] < t.tab[f]) {
      echange(t, j, i);
      j ++;
    }
  }
  echange(t, j, f);
  return j;
}

void tri_rapide_aux(tableau t, int d, int f) {
  /* Trie, par effet de bord, le sous-tableau t[d..f]. */
  if (d < f) {
    /* Choix du pivot, on pourrait choisir aléatoirement dans [d, f] */
    int p = d;
    int g = partitionne(t, d, f, p);
    tri_rapide_aux(t, d, g - 1);
```

♣. https://en.wikipedia.org/wiki/Selection_sort

```c
    tri_rapide_aux(t, g + 1, f);
  }
}

void tri_rapide(tableau t) {
  /* Trie, par effet de bord, le tableau t. */
  tri_rapide_aux(t, 0, t.taille - 1);
}
```

6 Autres utilisations classiques des tableaux en algorithmique

R. 4-32 Écrire une fonction prenant en paramètre un tableau d'entiers, triés par ordre croissant, de taille n, un entier p et testant si l'entier p apparaît dans le tableau. Cette fonction devra être récursive et ne pas manipuler de boucles **while**/**for**. On assurera une complexité en $\mathcal{O}(\log(n))$.

Solution

C-tableau-dichotomierec

```c
bool dichotomie_rec_aux(tableau t, int d, int f, int e) {
  /* Retourne si oui ou non e apparaît dans le sous-tableau t[d..f],
   * supposé croissant. */
  if (d <= f) {
    /* L'indice du milieu entre d et f. */
    int m = d + (f - d) / 2;
    if (t.tab[m] == e) { return true ; }
    else if (t.tab[m] < e) { return dichotomie_rec_aux(t, m + 1, f, e); }
    else { return dichotomie_rec_aux(t, d, m - 1, e); }
  } else {
    return false;
  }
}

bool dichotomie_rec(tableau t, int e) {
  /* Retourne si oui ou non e apparaît dans le tableau t supposé
   * croissant. */
  return dichotomie_rec_aux(t, 0, t.taille - 1, e);
}
```

R. 4-33 Écrire une fonction prenant en paramètre un tableau d'entiers, triés par ordre croissant, de taille n, un entier p et testant si l'entier p apparaît dans le tableau. Cette fonction ne devra pas être récursive. On assurera une complexité en $\mathcal{O}(\log(n))$.

Solution

C-tableau-dichotomiewhile

```c
bool dichotomie(tableau t, int e) {
  /* Retourne si oui ou non e apparaît dans le tableau t supposé
   * croissant. */
  int d = 0;
  int f = t.taille - 1;
  bool trouve = false;
  while (d <= f && !trouve) {
    int m = d + (f - d) / 2;
    if (t.tab[m] == e) { trouve = true ; }
    else if (t.tab[m] < e) { d = m + 1 ; }
    else { f = m - 1 ; }
  }
```

```
13    return trouve;
14  }
```

R. 4-34 Écrire une fonction prenant en paramètre un tableau de booléens représentant un entier écrit en base 2 (les bits de poids faibles sont à droite) et modifiant le tableau pour qu'il représente l'entier suivant. Cette fonction renverra un booléen indiquant si l'incrémentation a bien été faite, de sorte qu'on renverra `false` s'il n'est pas possible d'incrémenter l'entier, en pareil cas le tableau sera modifié pour ne contenir que des `false`.

Solution

C-tableau-incrementationbinaire

```c
bool incrementation(btableau bt) {
  /* Incrémente le tableau de booléens bt, vu comme l'écriture en base 2
   * d'un entier. Le booléen renvoyé indique si l'incrémentation a eu
   * lieu. */
  bool retenue = true;
  int  i = bt.taille - 1;
  while (i >= 0 && retenue) {
    bt.tab[i] = !bt.tab[i];
    if (bt.tab[i]) {           /* false avant */
      retenue = false;
    }
    i --;
  }
  return !retenue;
}
```

R. 4-35 Écrire une fonction permettant de calculer la médiane d'un tableau d'entiers deux à deux distincts sans trier ce tableau. On attend une fonction de complexité pire cas quadratique.

Solution

C-tableau-mediane

```c
int mediane(tableau t) {
  /* Calcule la médiane d'un tableau d'entiers t, dont les éléments sont
   * supposés deux à deux distincts. */
  int objectif = t.taille / 2 ;
  for (int i = 0 ; i < t.taille ; i ++) {
    if (compte_plus_petit(t, t.tab[i]) == objectif) {
      return t.tab[i];
    }
  }
  printf("Cas impossible");
  exit(1);
}
```

1 Différents types de listes chaînées

R. 5-1 Définir une structure de cellule pour les listes *simplement* chaînées d'éléments constitués d'un entier seul. Définir ensuite un nouveau type pour de telles listes.

Solution

0-liste-type-0

```c
typedef struct cell1_s cell1;
struct cell1_s {
  int val;
  cell1* next; //adresse de la cellule suivante ou NULL pr la dernière
};

typedef cell1* liste_c1;
//adresse de la 1ere cellule ou NULL pr la liste vide
```

R. 5-2 Définir une structure de cellule pour les listes *doublement* chaînées d'éléments constitués d'un entier seul. Définir ensuite un nouveau type pour de telles listes, qui doivent pouvoir être parcourues depuis la tête ou depuis la queue.

Solution

0-liste-type-1

Il ne faut pas seulement un pointeurs enregistrant l'adresse de la première cellule, mais aussi un enregistrant la dernière. C'est pourquoi on définit ici aussi une structure pour les listes.

```c
typedef struct cell2_s cell2;
struct cell2_s {
  int val;
  cell2* next; //adresse de la cellule suivante ou NULL pr la dernière
  cell2* prev; //adresse de la cellule précedente ou NULL pr la 1ère
};

struct liste_c2_s {
  cell2* debut; //adresse de la 1ere cellule ou NULL pr la liste vide
  cell2* fin;
};
typedef struct liste_c2_s liste_c2;

```

R. 5-3 Définir une structure de cellule pour les listes *simplement* chaînées d'éléments constitués d'un entier *et* d'une chaîne de caractères. Définir ensuite un nouveau type pour les listes chaînées de tels éléments.

Solution

0-liste-type-2

```c
typedef struct cell3_s cell3;
struct cell3_s {
  int entier;
  char* chaine;
  cell3* next; //adresse de la cellule suivante ou NULL pr la derniere
};

```

```c
typedef cell3* liste_c3;
  //adresse de la 1ere cellule ou NULL pr la liste vide
```

R. 5-4 Définir une structure de cellule pour les listes *simplement* chaînées d'éléments constitués d'un entier et d'un tableau. On peut imaginer par exemple qu'un élément est un sommet muni du tableau de ses voisins dans un graphe. Définir ensuite un nouveau type pour les listes chaînées de tels éléments.

Solution

O-liste-type-3

Il faut penser à stocker dans chaque cellule un entier supplémentaire pour la taille du tableau.

```c
typedef struct cell4_s cell4;
struct cell4_s {
  int n;
  int* tab; //tableau de taille n
  cell4* next; //adresse de la cellule suivante ou NULL pr la derniere
};

typedef cell4* liste_c4;
  //adresse de la 1ere cellule ou NULL pr la liste vide
```

2 Listes d'entiers simplement chaînées

Dans les questions suivantes on travaille avec le type ci-dessous.

```c
typedef struct s_cell cell;
struct s_cell {
  int val ;
  cell* next ; /* adresse de la cell. suivante ou NULL en fin de liste */
};

/* adresse de la 1ere cellule ou NULL pr la liste vide */
typedef cell* liste_c;
```

R. 5-5 Définir une fonction qui crée une liste chaînée d'entiers réduite à une cellule. L'entier à enregistrer dans cette cellule sera pris en paramètre.

Solution

C-liste-creation

```c
cell* create_cell(int elem){
  //retourne une liste réduite à une cellule contenant elem
  //!! attention allocation de mémoire dynamique --> penser au free !!
  cell* res = (cell*) malloc(sizeof(cell));
  res->val = elem;
  res->next = NULL;
  return res;
}
```

R. 5-6 Définir une fonction qui affiche une liste chaînée d'entiers. Cette fonction doit pouvoir s'exécuter sur une liste vide, et fournir alors un affichage adéquat.

Solution

C-liste-affiche

```c
void affiche_liste(liste_c l){
  //affiche la liste l
  if(l == NULL){
    printf("La liste est vide.\n----\n");
  }
  else{
    cell* p = l; // p joue le rôle de curseur
    while(p != NULL){
      printf("%d --",p->val);
      p = p->next;
    }
    printf("\n----\n");
  }
};
```

R. 5-7 Définir une fonction qui prend en paramètre un entier $n \in \mathbb{N}$ et qui renvoie une liste chaînée contenant les n premiers entiers naturels. L'itération de cette fonction devra être réalisée avec une boucle **for**.

Solution

C-liste-creationentiers

```c
liste_c cree_liste_premiers_entiers(int n) {
  //hyp : n >= 0
  //crée puis retourne une liste des n premiers entiers
  //!! attention allocation de mémoire dynamique --> penser au free !!
  if (n == 0){return  NULL;}
  liste_c res = create_cell(0);
  cell * p = res;
  for(int i = 1; i < n; i++){
    p->next = create_cell(i);
    p = p->next;
  }
  return res;
}
```

R. 5-8 Définir une fonction qui libère l'espace mémoire alloué pour une liste chaînée d'entiers.

Solution

C-liste-free

```c
void free_liste_c(liste_c l){
  //libère la mémoire occupée par les cells de l
  cell* p = l;
  while(p != NULL){
    cell* temp = p->next;
    free(p);
    p = temp;
  }
}
```

R. 5-9 Définir une fonction qui crée une liste chaînée d'entiers à partir d'un tableau d'entiers.

Solution

C-liste-listedetab

```c
liste_c create_from_tab_alt(int* tab, int lg){
  // hyp : tab est de taille lg
  // retourne une liste chaînée contenant les éléments de tab
  // !! attention allocation dynamique de mémoire -> penser au free !!
  cell* res = NULL;
  for (int i = lg-1; i >= 0 ; i --) {
    cell* n = create_cell(tab[i]);
    n->next = res;
    res = n;
  }
  return res;
}
```

R. 5-10 Définir une fonction qui crée un tableau d'entiers à partir d'une liste chaînée d'entiers. L'itération de cette fonction devra être réalisée avec une boucle `for`.

Solution

C-liste-tabdeliste

On définit d'abord la fonction qui renvoie la longueur d'une liste afin de pouvoir réserver la bonne taille d'espace mémoire.

```c
int nb_elem(liste_c l){
  //hyp : l n'est pas NULL
  //retourne le nombre d'élém de l
  cell* p = l;
  int res = 0;
  while(p != NULL){
    res = res + 1;
    p = p->next;
  }
  return res;
}
```

Puis la création du tableau à partir d'une liste

```c
int* tab_from_liste(liste_c l){
  //hyp : l n'est pas NULL
  // retourne un tableau contenant les éléments de l
  // !! attention allocation dynamique de mémoire -> penser au free !!
  int n = nb_elem(l);
  int* tab = malloc(sizeof(int)*n);
  cell* p = l;
  for(int i = 0; i < n; i++){
    tab[i] = p->val;
    p = p->next;
  }
  return tab;
}
```

R. 5-11 Définir une fonction qui ajoute un à un les éléments d'une liste `l1` dans une autre liste `l2`. Les listes `l1` et `l2` ne doivent pas être dégradées par cette procédure. On renverra la nouvelle liste, dans laquelle les éléments de `l1` apparaissent avant ceux de `l2`, en ordre inverse. Par exemple pour `l1` la liste $1/2/3$ et `l2` la liste $4/5/6$ le résultat attendu est la liste $3/2/1/4/5/6$.

Solution

C-liste-deverse

```c
liste_c rev_append(liste_c l1, liste_c l2){
  //ajoute un à un les éléments de l1 en tete de l2 et retourne
  cell* p1 = l1; //curseur dans l1
  cell* res = l2; //tete de l2 avec les élém ajoutés
  while(p1 != NULL){
    cell* new = create_cell(p1->val);
    new->next = res;
    res = new;
    p1 = p1->next;
  }
  return res;
}
```

R. 5-12 Définir une fonction qui calcule le miroir d'une liste `l1` sans la dégrader.

Solution

C-liste-miroir-0

```c
liste_c miroir(liste_c l1){
  //crée le miroir de l1 sans dégrader l1
  return rev_append(l1,NULL);
}
```

R. 5-13 Définir une fonction qui crée une copie d'une liste `l1` sans la dégrader.

Solution

C-liste-copy

```c
liste_c copie(liste_c l1){
  //crée une copie de l1 sans dégrader l1
  liste_c temp = miroir(l1);
  liste_c res = miroir(temp);
  free_liste_c(temp);
  return res;
}
```

R. 5-14 Définir une fonction qui prend en paramètre une liste et un entier naturel k inférieur à la taille de la liste, et qui supprime les éléments de cette liste au delà des k premiers. On veillera à désallouer l'espace mémoire des cellules supprimées de la liste.

Solution

C-liste-tronque-0

```c
void tronque_liste(liste_c* pl1,int k){
  //hyp : 0<= k <= la longueur de la liste *pl1
  // supprime les éléments de *pl1 au delà du k-ieme
  int cpt = 0;
  liste_c* p = pl1;
  //inv : à l'adresse p est enregistrée l'adresse de la (cpt+1)-ième cell.
  while(cpt < k){
    p = &((*p)->next);
    cpt ++;
  } //ici cpt =k
  cell* to_delete = *p; //adresse de la cellule à supprimer
  *p =  NULL; //on rompt le lien vers le k+1ème élém
  while(to_delete != NULL){
```

```
    cell* temp = to_delete -> next;
    //printf("....on supprime l'elem %d \n",to_delete->val);
    free(to_delete);
    to_delete = temp;
  }
}
```

R. 5-15 Définir une fonction qui prend en paramètres une liste et un entier naturel k, et qui supprime les éléments de cette liste au delà des k premiers, s'il y en a. On veillera à désallouer l'espace mémoire des cellules supprimées de la liste.

Solution

C-liste-tronque-1

```
void tronque_liste_robuste(liste_c* pl1,int k){
  // hyp : 0<= k
  // supprime les éléments de *pl1 au delà du k-ieme
  int cpt = 0;
  liste_c* p = pl1;
  //inv : à l'adresse p est enregistrée l'adresse de la (cpt+1)-ième cell.
  while( (cpt < k) && ( *p != NULL) ){
    p = &((*p)->next);
    cpt ++;
  } //ici cpt =k
  if(*p != NULL){
    cell* to_delete = *p; //adresse de la cellule à supprimer
    *p =  NULL; //on rompt le lien vers le k+1ème élém
    while(to_delete != NULL){
      cell* temp = to_delete -> next;
      printf("....on supprime l'elem %d \n",to_delete->val);
      free(to_delete);
      to_delete = temp;
    }
  }
}
```

R. 5-16 Définir une fonction qui prend en paramètre une liste passée par référence et la transforme en place en la liste miroir. Cette fonction ne devra pas allouer d'espace mémoire supplémentaire, mais pourra en revanche faire usage de la pile d'appels.

Solution

C-liste-miroir-1

```
liste_c aux(liste_c todo, liste_c res){
  //renverse les liens de la liste todo, met à la suite res
  //et renvoie la tête de cette nouvelle liste
  if(todo == NULL){return res;}
  else{
    cell* reste = todo->next;
    todo->next = res;
    return aux(reste, todo);
  }
}

void miroir_en_place (liste_c* pl1){
  *pl1 = aux(*pl1, NULL);
}
```

Dans toute cette feuille de révision on suppose définis les types OCaml suivants, permettant respectivement la représentation de graphes (orientés ou non) par table de listes d'adjacences, par matrice d'adjacence.

```
type graphe_adj =  (* graphes par table de listes d'adjacence *)
  int list array

type graphe_mat =  (* graphes par matrice d'adjacence *)
  bool array array
```

1 Pied à l'étrier

R. 6-1 Définir une fonction OCaml prenant en paramètres un graphe `g` (représenté par matrice d'adjacence), deux sommets entiers `i` et `j` et retournant si l'arête $\{i, j\}$ est une arête du graphe `g`.

Solution

O-graphe-memarete-0

```
let est_arete_mat (g: graphe_mat) (i: int) (j: int): bool =
  g.(i).(j)
```

R. 6-2 Définir une fonction OCaml prenant en paramètres un graphe `g` (représenté par table de listes d'adjacence), deux sommets entiers `i` et `j` et retournant si l'arête $\{i, j\}$ est une arête du graphe `g`.

Solution

O-graphe-memarete-1

```
let est_arete (g: graphe_adj) (i: int) (j: int): bool =
  List.mem j g.(i)
```

R. 6-3 Définir une fonction OCaml prenant en paramètres un entier `n` et une liste `l` de couples d'entiers et retournant le graphe orienté (représenté par table de listes d'adjacence) dont les sommets sont les entiers de l'intervalle $[\![0, n-1]\!]$ et les arcs sont les éléments de `l`. On peut supposer que `l` est sans doublons. On déclenchera une erreur si `l` contient un couple qui ne peut être un arc possible de ce graphe.

Solution

O-graphe-fabriqueo

```
let fabrique_graphe_o (n: int) (l: (int * int) list): graphe_adj =
  let rep = Array.make n [] in
  List.iter (fun (i, j) ->
    if ( i < 0 || i >= n || j < 0 || j >= n || i = j )
    then let cij= "("^(string_of_int i)^", "^(string_of_int j)^")"
      in failwith (cij ^ " n'est pas un arc possible")
    else rep.(i) <- j :: rep.(i)
  ) l;
  rep
```

R. 6-4 Définir une fonction OCaml prenant en paramètres un entier `n` et une liste `l` de couples d'entiers et retournant le graphe non orienté (représenté par table de listes d'adjacence) dont les

sommets sont les entiers de l'intervalle $[\![0, n-1]\!]$ et les arêtes sont données par les couples de `l`. On peut supposer que les couples de `l` représentent deux à deux des arêtes différentes. On déclenchera une erreur si `l` contient un couple qui ne correspond pas à une arête possible de ce graphe.

Solution

O-graphe-fabriqueno

```
let fabrique_graphe_no (n: int) (l: (int * int) list): graphe_adj =
  let rep = Array.make n [] in
  List.iter (fun (i, j) ->
    if ( i < 0 || i >= n || j < 0 || j >= n || i = j )
    then let cij= "("^(string_of_int i)^", "^(string_of_int j)^")"
      in failwith (cij ^ " n'est pas une arête possible")
    else (rep.(i) <- j :: rep.(i) ; rep.(j) <- i :: rep.(j))
  ) l;
  rep
```

R. 6-5 Définir une fonction OCaml prenant en paramètres un entier `n` et une liste `l` de couples d'entiers et retournant le graphe orienté (représenté par matrice d'adjacence) dont les sommets sont les entiers de l'intervalle $[\![0, n-1]\!]$ et les arcs sont données par les éléments de `l`.

Solution

O-graphe-fabriqueomat

```
let fabrique_graphe_o_mat (n: int) (l: (int * int) list): graphe_mat =
  let mat = Array.make_matrix n n false in
  List.iter (fun (i, j) ->
    if ( i < 0 || i >= n || j < 0 || j >= n || i = j )
      then let cij= "("^(string_of_int i)^", "^(string_of_int j)^")"
        in failwith (cij ^ " n'est pas un arc possible")
      else mat.(i).(j) <- true
    ) l;
  mat
```

R. 6-6 Définir une fonction OCaml prenant en paramètre un graphe orienté (représenté par table de listes d'adjacence) et retournant la liste de ses arcs.

Solution

O-graphe-listearcs-0

```
let liste_arcs (g: graphe_adj): (int * int) list =
  let x, _ = Array.fold_left (fun (accu, i) vois_i ->
      let new_acc = List.fold_left (fun acc j ->
          (i, j) :: acc
        ) accu vois_i in
      (new_acc, i+1)
    ) ([], 0) g in
  x
```

R. 6-7 Définir une fonction OCaml prenant en paramètre un graphe orienté (représenté par matrice d'adjacence) et retournant la liste de ses arcs.

Solution

O-graphe-listearcs-1

```
let liste_arcs_mat (g: graphe_mat): (int * int) list =
  let x, _ = Array.fold_left (fun (acc, i) t ->
      let acc3, _ = Array.fold_left (fun (acc, j) b ->
```

```
          let acc2 = if b then (i, j) :: acc else acc in
          (acc2, j+1)
        ) (acc, 0) t in
      (acc3, i+1)
    ) ([], 0) g
```

R. 6-8 La matrice d'accessibilité d'un graphe g non orienté à n sommets est une matrice de booléens de dimension $n \times n$ contenant `true` dans la case d'indice (i, j) si et seulement il existe un chemin du sommet i au sommet j dans le graphe. Définir une fonction prenant en paramètres un graphe (représenté par matrice d'adjacence) et retournant sa matrice d'accessibilité. On s'autorise une complexité en $\mathcal{O}(n^4)$.

Solution

0-graphe-accessibilite

```
let produit_matriciel (t1: bool array array) (t2: bool array array): bool array array =
  let n = Array.length t1 in
  let res = Array.make_matrix n n false in
  for i = 0 to (n-1) do
    for j = 0 to (n-1) do
      for k = 0 to (n-1) do
        if t1.(i).(k) && t2.(k).(j) then res.(i).(j) <- true
      done
    done
  done; res

let somme_matricielle (t1: bool array array) (t2: bool array array): bool array array =
  let n = Array.length t1 in
  let res = Array.make_matrix n n false in
  for i = 0 to (n-1) do
    for j = 0 to (n-1) do
      res.(i).(j) <- t1.(i).(j) || t2.(i).(j)
    done
  done; res

let accessibilite (g: graphe_mat): bool array array =
  let n = Array.length g in
  let res = ref g in
  for i = 1 to n do
    res := somme_matricielle !res (produit_matriciel !res g)
  done;
  !res
```

R. 6-9 La matrice d'accessibilité d'un graphe g non orienté à n sommets est une matrice de booléens de dimension $n \times n$ contenant `true` dans la case d'indice (i, j) si et seulement s'il existe un chemin du sommet i au sommet j dans le graphe. Définir une fonction prenant en paramètres un graphe (représenté par matrice d'adjacence) et retournant sa matrice d'accessibilité. On s'autorise une complexité en $\mathcal{O}(n^3)$, on pourra pour cela s'aider de l'algorithme de Roy-Warshall.

Solution

0-graphe-accessibiliterw

```
let roy_warshall (g: graphe_mat): bool array array =
  let n = Array.length g in
  let old = ref (Array.map Array.copy g) in
    for i = 0 to (n-1) do !old.(i).(i) <- true done;
  (*inv : pr ts i,j old.(i).(j) indique s'il existe une chaîne entre
    i et j ds g dont les sommets intermédiaires sont dans [0..k[ *)
```

```
  let cur = ref (Array.make_matrix n n false) in
  for k = 0 to n-1 do
    for i = 0 to n-1 do
      for j = 0 to n-1 do
        !cur.(i).(j) <- !old.(i).(j) || ( !old.(i).(k) && !old.(k).(j) )
      done
    done;
    let tmp = !old in
    old := !cur ;
```

R. 6-10 On considère un graphe g orienté, à n sommets dont les arcs sont *pondérés* par des longueurs qui sont des entiers positifs. Un tel graphe est représenté en mémoire au moyen d'une matrice d'entiers de dimension $n \times n$ contenant dans la case d'indice (i, j) la pondération de l'arc (i, j) ou -1 si un tel arc n'existe pas. Définir une fonction `floyd_warshall : int array array -> int array array` appliquant l'algorithme de Floyd-Warshall au graphe qui lui est passé en paramètre et retournant donc la matrice des plus courts chemins. Ainsi La matrice résultat devra contenir dans sa case d'indice (i, j) ou bien un entier positif indiquant la longueur du plus court chemin de i à j dans le graphe g, ou bien -1 si un tel chemin n'existe pas.

Solution

O-graphe-fw

```
(** Calcule l'addition de x et y lorsque -1 représente l'infini *)
let add_ (x: int) (y: int) = if x = -1 || y = -1 then -1 else x + y
(** Calcule le minimum de x et y lorsque -1 représente l'infini *)
let min_ (x: int) (y: int) = if x = -1 then y else if y = -1 then x else min x y

let floyd_warshall (g: int array array): int array array =
  let n = Array.length g in
  let old = ref (Array.map Array.copy g) in
  for i = 0 to (n-1) do !old.(i).(i) <- 0 done;
  (* inv : pr ts i,j old.(i).(j) indique la longueur d'un plus court chemin
     de i à j dans g dont les sommets intermédiaires sont dans [0..k[ *)
  let cur = ref (Array.make_matrix n n (-1)) in
  for k = 0 to n-1 do
    for i = 0 to n-1 do
      for j = 0 to n-1 do
        !cur.(i).(j) <- min_ !old.(i).(j) (add_ !old.(i).(k) !old.(k).(j))
      done
    done;
    let tmp = !old in
    old := !cur ;
    cur := tmp
  done;
  !old
```

2 Parcours de graphes

Dans cette partie les graphes seront tous représentés par table de listes d'adjacence (*i.e.* avec le type `graphe_adj`).

R. 6-11 Définir une fonction OCaml calculant un parcours en profondeur (une permutation des sommets, de type `int list`) du graphe passé en argument. On s'efforcera d'utiliser la pile des appels récursifs, pour stocker les éléments encore à traiter. La complexité de l'implémentation devra être en $\mathcal{O}(n + m)$ où n est le nombre de sommets du graphe et m est le nombre d'arêtes.

Solution

0-graphe-parcoursprof-0

```
let parcours_prof_rec (g: graphe_adj): int list =
  let n       = Array.length g in
  let visites = Array.make n false in
  let res = ref [] in
  let rec explore_descendants (s: int): unit =
  (* explore les descendants de s dans g non encore visités met à jour le
     tableau visités et complete res de sorte que List.rev !res soit un parcours
     en prof du sous-graphe de g induit par les sommets visités *)
    if not (visites.(s)) then
      begin
        visites.(s) <- true;
        res := s::!res;
        List.iter explore_descendants g.(s)
      end
  in
  for s = 0 to n - 1 do
    explore_descendants s
  done;
  List.rev !res
```

R. 6-12 Définir une fonction OCaml calculant un parcours en profondeur (une permutation des sommets, de type `int list`) du graphe passé en argument. On s'efforcera d'utiliser le module `Stack`, pour stocker les éléments encore à traiter. La complexité de l'implémentation devra être en $\mathcal{O}(n+m)$ où n est le nombre de sommets du graphe et m est le nombre d'arêtes.

Solution

0-graphe-parcoursprof-1

```
let parcours_prof_avec_todo (g: graphe_adj): int list =
  (* nombre de sommets dans le graphe *)
  let n       = Array.length g in
  (* parcours que nous sommes en train de fabriquer *)
  let l       = ref [] in
  (* ensemble des visités, représenté par un tableau indicateur :
     i est visité, ssi visité.(i) = true *)
  let visites = Array.make n false in
  let explore_descendants (s: int) =
    (* todo pour stocker les sommets à traiter *)
    let todo = Stack.create () in
    (* on ajoute s dans la pile des choses à traiter *)
    Stack.push s todo;
    while (not (Stack.is_empty todo)) do
      (* on extrait un élément dans la pile des choses à traiter *)
      let v = Stack.pop todo in
      (* on teste si v a déjà été visité *)
      if not (visites.(v)) then
        (* si ce n'est pas le cas on le visite *)
        begin
          (* on l'ajoute aux visités pour ne pas le revisiter plus tard *)
          visites.(v) <- true;
          (* on ajoute les successeurs de v dans todo *)
          List.iter (fun y -> Stack.push y todo) g.(v);
          (* on ajoute v au parcours *)
          l := v :: !l
        end
    done
```

```
  in
  for s = 0 to n - 1 do
    if not visites.(s) then explore_descendants s
  done;
  List.rev !l
```

R. 6-13 Définir une fonction OCaml calculant un parcours en largeur (une permutation des sommets, de type `int list`) du graphe passé en argument. On s'efforcera d'utiliser le module `Queue`, pour stocker les éléments encore à traiter. La complexité de l'implémentation devra être en $\mathcal{O}(n+m)$ où n est le nombre de sommets du graphe et m est le nombre d'arêtes.

Solution

O-graphe-parcourslarg

```
let parcours_largeur_avec_todo (g: graphe_adj): int list =
  (* nombre de sommets dans le graphe *)
  let n       = Array.length g in
  (* parcours que nous sommes en train de fabriquer *)
  let l       = ref [] in
  (* ensemble des visités, représenté par un tableau indicateur :
     i est visité, ssi visité.(i) = true *)
  let visites = Array.make n false in
  let explore_descendants (s: int) =
    (* todo pour stocker les sommets à traiter *)
    let todo = Queue.create () in
    (* on ajoute s dans la file des choses à traiter *)
    Queue.push s todo;
    while (not (Queue.is_empty todo)) do
      (* on extrait un élément dans la file des choses à traiter *)
      let v = Queue.pop todo in
      (* on teste si v a déjà été visité *)
      if not (visites.(v)) then
        (* si ce n'est pas le cas on le visite *)
        begin
          (* on l'ajoute aux visités pour ne pas le revisiter plus tard *)
          visites.(v) <- true;
          (* on ajoute les successeurs de v dans todo *)
          List.iter (fun y -> Queue.push y todo) g.(v);
          (* on ajoute v au parcours *)
          l := v :: !l
        end
    done
  in
  for s = 0 to n - 1 do
    if not visites.(s) then explore_descendants s
  done;
  List.rev !l
```

3 Application des parcours

Dans cette partie les graphes seront représentés par table de listes d'adjacence (ou table de listes de successeurs pour les graphes orientés), *i.e.* par le type OCaml `graphe_adj` défini plus haut.

R. 6-14 Définir une fonction OCaml prenant en paramètre un graphe non orienté et deux sommets de ce graphe, et testant si l'un est accessible depuis l'autre.

Solution

0-graphe-accessibilite

```
let sont_connectes (g: graphe_adj) (u: int) (v: int) : bool =
  let n = Array.length g in
  assert( 0 <= u && u < n && 0 <= v && v < n );
  let visites = Array.make n false in
  (** Marque à true dans visités les sommets accessibles depuis s dans g *)
  let rec explore_descendants (s: int): unit =
    if not (visites.(s)) then
      begin
        visites.(s) <- true;
        List.iter explore_descendants g.(s)
      end
  in
  explore_descendants u;
  visites.(v)
```

R. 6-15 Définir une fonction OCaml prenant en paramètre un graphe non orienté et testant si celui-ci est connexe.

Solution

0-graphe-connexe

```
let est_connexe (g: graphe_adj): bool =
  let n       = Array.length g in
  let visites = Array.make n false in
  (** Marque à true dans visités les sommets accessibles depuis s dans g *)
  let rec explore_descendants (s: int): unit =
    if not (visites.(s)) then
      begin
        visites.(s) <- true;
        List.iter explore_descendants g.(s)
      end
  in
  (n = 0) || (explore_descendants 0 ; Array.for_all (fun x -> x) visites)
```

R. 6-16 Définir une fonction OCaml prenant en paramètre un graphe non orienté et testant si celui-ci est un arbre.

Solution

0-graphe-estarbre

On rappelle qu'un arbre peut être défini comme un graphe connexe ayant une arête de moins que de sommets.

```
let est_arbre (g: graphe_adj): bool =
  let n = Array.length g in
  let deuxm = (* on compte deux fois le nombre d'arêtes de g *)
    (Array.fold_left (fun acc l -> acc + (List.length l)) 0 g) in
  (est_connexe g) && (deuxm = 2 * n - 2) (*m = n-1 ssi 2m = 2n-2*)
```

R. 6-17 Définir une fonction OCaml prenant en paramètre un graphe non orienté et retournant sa décomposition en composantes connexes sous la forme d'un tableau associant à chaque sommet un identifiant unique de composante.

Solution

0-graphe-decompcc

```
let decomposition_cc (g: graphe_adj): int array =
```

```
  let n       = Array.length g in
  let num_visites = Array.make n (-1) in
  (* on indique dans num_visite.(s) le numéro de l'exploration au *)
  (* cours de laquelle s a été visité, -1 s'il ne l'est pas encore*)
  let rec explore_descendants (num: int) (s: int): unit =
    (* marque à num dans num_visites tous les sommets accessibles*)
    (* depuis s dans g et non encore visités selon num_visites   *)
    if num_visites.(s) = -1 then
      begin
        num_visites.(s) <- num ;
        List.iter (explore_descendants num) g.(s)
      end
  in
  let num = ref (-1) in
  for i = 0 to n - 1 do
    if num_visites.(i) = - 1 (* nouvelle exploration depuis i *)
    then (num := !num + 1;  explore_descendants (!num) i )
  done;
  num_visites
```

R. 6-18 Définir une fonction OCaml prenant en paramètre un graphe non orienté et retournant si ce graphe est biparti.

Solution

0-graphe-estbiparti

Un graphe est biparti ssi il est 2-coloriable. L'algorithme ci-dessous est un algorithme de coloration glouton, qui prend soin de colorier les sommmets dans l'ordre d'un parcours.

```
exception NotBiparti
let est_biparti (g: graphe_adj): bool =
  let n = Array.length g in
  (* coloration booléenne de chaque sommet *)
  (* sert aussi d'ensemble des visités : visité = colorié *)
  let coloration : bool option array = Array.make n None in
  let rec colorie_descendants (c: bool) (s: int) : unit =
    (* colorie s et ses descendants en donnant la couleur c à s, *)
    (* et en respectant coloration, si c'est possible,           *)
    (* lève l'exception NotBiparti sinon                         *)
    match coloration.(s) with
    | Some (b) -> if b <> c then raise NotBiparti
    | None -> (* si s n'est pas encore colorié = pas visité *)
      begin
        (* on le colorie en c *)
        coloration.(s) <- Some(c);
        (* on essaye de colorier ses successeurs de l'autre couleur *)
        List.iter (fun x -> colorie_descendants (not c) x) (g.(s));
        (* on ajoute s au dessus de ses successeurs dans la liste *)
      end
  in try
    for i = 0 to n - 1 do
      (* on colorie les points de régénération en false, arbitrairement *)
      if coloration.(i) = None
      then colorie_descendants false i
    done;
    true
  with
  | NotBiparti -> false

```

R. 6-19 Définir une fonction OCaml prenant en paramètres un graphe non orienté, deux sommets u et v et retournant la distance, en nombre d'arêtes séparant u et v.

Solution

O-graphe-distancenbarcs

On utilise un algorithme de parcours en largeur, utilisant le module `Queue` pour implémenter la file de sommets à traiter.

```
exception Dist of int
let distance_nb_arc (g: graphe_adj) (x: int) (y: int): int =
  let n       = Array.length g in
  let visites = Array.make n false in
  let explore_descendants (s: int) =
    let todo = Queue.create () in
    (* Dans la file, on ajoute des couples (u, d) où u est un sommet à
       visiter et d la distance d'un chemin de s à u. *)
    Queue.push (s, 0) todo;
    while (not (Queue.is_empty todo)) do
      let (u, d) = Queue.pop todo in
      if not (visites.(u)) then
        begin
          if u = y then raise (Dist d)
          else
            begin
              visites.(u) <- true;
              List.iter (fun v -> Queue.push (v, d+1) todo) g.(u);
            end
        end
    done
  in
  try (explore_descendants x ; failwith "pas de chemin")
  with | Dist d -> d
```

R. 6-20 Définir une fonction OCaml prenant en paramètre un graphe orienté supposé sans circuit et retournant, *à l'aide d'un parcours* un tri topologique sous la forme d'une liste de sommets.

Solution

O-graphe-tritopo

```
let tri_topologique (g: graphe_adj): int list =
  (* calcule un tri topologique de g *)
  let n = Array.length g in
  let tri_topo = ref [] in
  let visites = Array.make n false in
  let rec explore_descendants (s: int) : unit =
    (* explore les descendants non visités de s dans g et       *)
    (* complète tri_topo de sorte qu'il soit un tri topo du     *)
    (* sous-graphe de g induit par les sommets visités          *)
    if not visites.(s) then
      begin
        visites.(s) <- true;
        List.iter (fun x -> explore_descendants x) (g.(s));
        tri_topo := s :: !tri_topo
      end
  in
  for s = 0 to n - 1 do
    explore_descendants s
  done;
  !tri_topo
```

R. 6-21 Définir une fonction OCaml prenant en paramètre un graphe orienté et retournant un booléen indiquant s'il admet un circuit.

Solution

O-graphe-admetcircuit

```
exception Circuit

type statut = | NonVu | Ouvert | Ferme

let admet_circuit (g: graphe_adj): bool =
  (* teste si g admet un circuit *)
  let n = Array.length g in
  let visites = Array.make n NonVu in
  let rec explore_descendants (s: int) : unit =
    (* explore les descendants de s dans g qui sont non fermés   *)
    (* selon visités et lève l'exception Circuit si l'un d'eux   *)
    (* est ouvert selon visites                                  *)
    if visites.(s) = Ouvert then raise Circuit;
    if visites.(s) = NonVu then
      begin
        visites.(s) <- Ouvert;
        List.iter (fun x -> explore_descendants x) (g.(s));
        visites.(s) <- Ferme;
      end
  in
  try
    for s = 0 to n - 1 do
      explore_descendants s
    done;
    false
  with
  | Circuit -> true
```

R. 6-22 Définir une fonction OCaml prenant en paramètre un graphe orienté et retournant un `int list option` qui donne un circuit si le graphe en admet un, et qui vaut None sinon.

Solution

O-graphe-circuit

```
(** Retourne un circuit du graphe [g] s'il en existe un, [None] sinon. *)
exception Found of int list
let trouve_circuit (g: graphe_adj): int list option =
  let n = Array.length g in
  let visites = Array.make n NonVu in
  (** Parcours les descendants de [s] non encore visités : retourne None si
      aucun circuit n'est détecté, Retourne [Some(x, chemin)] si un cycle a
      été détecté grâce au sommet [x]. [chemin] est un chemin menant de [s]
      à [x]. *)
  let rec explore_descendants (s: int): (int * int list) option =
    if visites.(s) = Ouvert then Some(s, [s])
    else if visites.(s) = NonVu then
      begin
        visites.(s) <- Ouvert;
        let rep = explore_fils g.(s) in
        visites.(s) <- Ferme;
        match rep with
        | Some(s', chemin) when s = s'      -> raise (Found(s :: chemin))
        | Some(s', chemin) (*when s <> s'*) -> Some(s', s :: chemin)
        | None                              -> None
```

```
      end
    else None
(** Parcours les sommets de [l] non encore visités : retourne None si
    aucun circuit n'est détecté, Retourne [Some(x, chemin)] si un cycle a
    été détecté grâce au sommet [x]. [chemin] est un chemin menant à
    [x]. *)
and explore_fils (l: int list): (int * int list) option =
  match l with
  | []      -> None
  | v :: l' ->
    match explore_descendants v with
    | None -> explore_fils l'
    | Some(s, chemin) -> Some(s, chemin)
in
try
  for i = 0 to (n-1) do
    match explore_descendants i
    with | None -> ()
         | Some _ -> failwith "cas impossible"
  done; None
with
| Found(chemin) -> Some(chemin)
```

Dans les questions suivantes on considère les types définis ci-dessous, permettant la représentation de graphes orientés et non orientés en C.

```c
struct list_s {
  int            elem ;         /* l'élément contenu dans le maillon */
  struct list_s* suivant ;      /* le maillon suivant */
};

/* Une liste est un pointeur vers son premier maillon */
typedef struct list_s* list;

struct graph_s {
  int   taille  ;               /* le nombre de sommets */
  list* contenu ;               /* le tableau des listes d'adjacence*/
};
typedef struct graph_s* graphe ;
```

1 Graphes en C

R. 7-1 Donner une fonction C, prenant en paramètre un graphe non orienté et affichant (dans le terminal) un parcours en profondeur de ce graphe. La gestion des sommets à visiter devra être faite par le mécanisme de récursivité.

Solution

C-graphe-parcoursprof-0

```c
/* Explore, en profondeur, les descendants de s dans le graphe g, qui ne
 * sont pas encore dans visites. */
void explore_descendants(bool* visites, graphe g, int s) {
  if (!visites[s]) {
    visites[s] = true;
    printf("%d, ", s); fflush(stdout);
    for (list c = g->contenu[s] ; c != NULL ; c = c->suivant) {
      explore_descendants(visites, g, c->elem);
    }
  }
}

/* Affiche un parcours en profondeur du graphe g. */
void parcours_profondeur(graphe g) {
  bool* visited = malloc(sizeof(bool) * g->taille);
  for (int i = 0 ; i < g->taille ; i ++) { visited[i] = false; }
  for (int i = 0 ; i < g->taille ; i ++) {
    explore_descendants(visited, g, i);
  }
}
```

R. 7-2 Donner une fonction C, prenant en paramètre un graphe non orienté et affichant (dans le terminal) un parcours en profondeur de ce graphe. La gestion des sommets à visiter devra être faite par une pile implémentée au moyen d'un tableau dynamique.

Solution

C-graphe-parcoursprof-1

```c
/*****************************************/
/* Implémentation d'une structure de pile */
/*****************************************/
struct pile_s {
  int* contenu;             /* les entiers dans la pile */
  int  taille_allouee;      /* la taille allouée du tableau contenu */
  int  nb_elems;            /* le nombre d'éléments stockés dans contenu */
};
typedef struct pile_s* pile;

/* Crée une pile vide. */
pile pile_cree_pile() {
  pile p = (pile) malloc(sizeof(struct pile_s));
  p->contenu = malloc(sizeof(int) * 1);
  p->taille_allouee = 1;
  p->nb_elems = 0;
  return p;
}

/* Ajoute l'élément x au sommet de la pile p. */
void pile_empile(pile p, int x) {
  if (p->nb_elems == p->taille_allouee) {
    int* n_contenu = (int*) malloc(2 * p->taille_allouee * sizeof(int));
    for (int i = 0 ; i < p->nb_elems ; i ++) {
      n_contenu[i] = p->contenu[i];
    }
    free(p->contenu);
    p->contenu = n_contenu;
    p->taille_allouee = 2 * p->taille_allouee;
  }
  p->contenu[p->nb_elems] = x;
  p->nb_elems ++;
}

/* Retire et retourne l'élément en sommet de pile p. */
int pile_depile(pile p) {
  if (p->nb_elems == 0) {printf("Dépile d'une pile vide\n"); exit(1);}
  p->nb_elems --;
  return p->contenu[p->nb_elems];
}

/* Teste si la pile p est vide. */
bool pile_est_vide(pile p) {
  return (p->nb_elems == 0);
}

/* Libère l'espace occupé par une pile. */
void pile_libere(pile p) {
  free(p->contenu);
  free(p);
}

/* Explore, en profondeur, les descendants de s dans le graphe g, qui ne
 * sont pas encore dans visites. */
void explore_descendants_2(bool* visited, graphe g, int s) {
  pile p = pile_cree_pile();
  pile_empile(p, s);
  while (!pile_est_vide(p)) {
```

```c
    int v = pile_depile(p);
    if (!visited[v]) {
      visited[v] = true;
      printf("%d, ", v); fflush(stdout);
      for (list c = g->contenu[v] ; c != NULL ; c = c->suivant) {
        pile_empile(p, c->elem);
      }
    }
  }
  pile_libere(p);
}

/* Affiche un parcours en profondeur du graphe g. */
void parcours_profondeur_2(graphe g) {
  bool* visited = malloc(sizeof(bool) * g->taille);
  for (int i = 0 ; i < g->taille ; i ++) { visited[i] = false; }
  for (int i = 0 ; i < g->taille ; i ++) {
    explore_descendants_2(visited, g, i);
  }
}
```

R. 7-3 Donner une fonction C, prenant en paramètre un graphe non orienté et affichant (dans le terminal) un parcours en largeur de ce graphe. La gestion des sommets à visiter devra être faite par une file implémentée au moyen de deux piles.

Solution

C-graphe-parcoursprof-2

```c
/* Transvase le contenu de la pile p1 dans la pile p2, en inversant l'ordre
 * des éléments. */
void transvase(pile p1, pile p2) {
  while (!(pile_est_vide(p1))) {
    int x = pile_depile(p1);
    pile_empile(p2, x);
  }
}

/* Explore, en largeur, les descendants de s dans le graphe g, qui ne
 * sont pas encore dans visites. */
void explore_descendants_3(bool* visited, graphe g, int s) {
  pile p1 = pile_cree_pile();
  pile p2 = pile_cree_pile();
  pile_empile(p1, s);
  while (!(pile_est_vide(p1) && pile_est_vide(p2))) {
    if (pile_est_vide(p2)) {transvase(p1, p2);}
    int v = pile_depile(p2);
    if (!visited[v]) {
      visited[v] = true;
      printf("%d, ", v); fflush(stdout);
      for (list c = g->contenu[v] ; c != NULL ; c = c->suivant) {
        pile_empile(p1, c->elem);
      }
    }
  }
  pile_libere(p1);
  pile_libere(p2);
  }

/* Affiche un parcours en largeur du graphe g. */
```

```c
void parcours_largeur(graphe g) {
  bool* visited = malloc(sizeof(bool) * g->taille);
  for (int i = 0 ; i < g->taille ; i ++) { visited[i] = false; }
  for (int i = 0 ; i < g->taille ; i ++) {
    explore_descendants_3(visited, g, i);
  }
}
```

R. 7-4 Donner une fonction C, prenant en paramètre un graphe non orienté et affichant (dans le terminal) un parcours en largeur de ce graphe. La gestion des sommets à visiter devra être faite par une file implémentée dans un tableau de taille n^2.

Solution

C-graphe-parcourslarg

```c
/*************************************************************/
/* Implémentation d'une file dans un tableau de taille statique */
/*************************************************************/
struct file_s {
  int* contenu;             /* les entiers dans la file */
  int  taille_allouee;      /* la taille allouée du tableau contenu */
  int  deb;                 /* le début de la file */
  int  fin;                 /* la fin de la file */
  bool est_vide;
};
typedef struct file_s* file;

/* Crée une file vide dont la taille ne dépassera par n. */
file file_cree_file(int n) {
  file f = (file) malloc(sizeof(struct file_s));
  f->contenu = malloc(sizeof(int) * n);
  f->taille_allouee = n;
  f->deb = 0;
  f->fin = 0;
  return f;
}

/* Ajoute l'élément x à la fin de la file f. */
void file_enfile(file f, int x) {
  f->contenu[f->fin] = x;
  f->fin ++;
}

/* Retire et retourne l'élément en début de file f. */
int file_defile(file f) {
  if (f->deb == f->fin) {printf("Défile d'une file vide\n"); exit(1);}
  f->deb ++;
  return f->contenu[f->deb-1];
}

/* Teste si la file p est vide. */
bool file_est_vide(file f) {
  return (f->deb == f->fin);
}

/* Libère l'espace occupé par une file. */
void file_libere(file f) {
  free(f->contenu);
  free(f);
```

```c
}

/* Explore, en largeur, les descendants de s dans le graphe g, qui ne
 * sont pas encore dans visites. */
void explore_descendants_4(bool* visited, graphe g, int s) {
  file f = file_cree_file(g->taille * g->taille);
  file_enfile(f, s);
  while (!file_est_vide(f)) {
    int v = file_defile(f);
    if (!visited[v]) {
      visited[v] = true;
      printf("%d, ", v); fflush(stdout);
      for (list c = g->contenu[v] ; c != NULL ; c = c->suivant) {
        file_enfile(f, c->elem);
      }
    }
  }
  file_libere(f);
}

/* Affiche un parcours en largeur du graphe g. */
void parcours_largeur_2(graphe g) {
  bool* visited = malloc(sizeof(bool) * g->taille);
  for (int i = 0 ; i < g->taille ; i ++) { visited[i] = false; }
  for (int i = 0 ; i < g->taille ; i ++) {
    explore_descendants_4(visited, g, i);
  }
}
```

R. 7-5 Donner une fonction C, prenant en paramètre un graphe orienté et testant si ce graphe contient un circuit.

Solution

C-graphe-circuit

```c
/* Retourne si oui ou non il existe un circuit dans les sommets accessibles
 * depuis le sommet s dans le graphe g. Le tableau visites contient : 0
 * pour les sommets non encore visités, 1 pour les sommets visités, 2 pour
 * les sommets en cours de visite. */
bool explore_descendants_5(int* visites, graphe g, int s) {
  if (visites[s] == 2) { return true; }
  else if (visites[s] == 0) {
    visites[s] = 2;
    for (list c = g->contenu[s] ; c != NULL ; c = c->suivant) {
      if (explore_descendants_5(visites, g, c->elem)) {return true;}
    }
    visites[s] = 1;
  }
  return false;
}

/* Retourne si oui ou non le graphe g contient un circuit. */
bool detecte_circuit(graphe g) {
  int* visites = malloc(sizeof(int) * g->taille);
  for (int i = 0 ; i < g->taille ; i ++) { visites[i] = 0; }
  for (int i = 0 ; i < g->taille ; i ++) {
    if (explore_descendants_5(visites, g, i)) {return true;}
  }
  return false;
}
```