TP 11 - En piste ! Speed coding

Règle du jeu : pour chaque exercice, on s'accordera C minutes pour coder; D minutes pour debrieffer. Les participants découvriront chaque exercice live. Ceux qui terminent avant la cloche vont aider les voisins...

1 Les yeux bandés : C = 15, D = 5

Écrire les fonctions suivantes, décrites par leur nom et leur typage :

1. `expo_naif : int -> int -> int;`
2. `expo_rapide : int -> int -> int;`
3. `reverse : 'a list -> 'a list` : temps linéaire obligatoire;
4. `swap : int -> int -> 'a array -> unit;`
5. `bubble_sort : 'a array -> unit.`
6. `do_list : ('a -> unit) -> 'a list -> unit`

2 Casser des listes : C = 10, D = 5

1. Écrire une fonction `split : ('a -> 'b -> bool) -> 'a -> 'b list -> 'b list *'b list` séparant une liste en deux en fonction des positions des membres de la liste par rapport à un objet donné (la fonction d'ordre est également donnée).
2. Écrire une fonction `quick : ('a -> 'a -> bool) -> 'a list -> 'a list` réalisant le tri rapide d'une liste.

3 Des énumérations : C = 15, D = 5

1. Écrire une fonction `my_map : ('a -> 'b) -> 'a list -> 'b list` faisant la même chose que `map` !
2. Écrire une fonction `parties : int -> int list list` calculant la liste des parties de $[\![1,n]\!]$.
3. Écrire une fonction `mid_adj : 'a -> int -> 'a list -> 'a list` réalisant l'adjonction d'un élément à l'intérieur d'une liste (dans une position donnée en paramètre).
4. Écrire une fonction `permutations : int -> int list list` calculant la liste des permutations de $[\![1,n]\!]$.

4 Écriture en base b : C = 10, D = 5

1. Écrire une fonction `decomp : int -> int -> int -> int list` décomposant un entier dans une base donnée, en imposant la taille de la décomposition. Exemple :

```
decomp 10 3 4;;
- : int list = [1; 0; 1; 0]
```

2. Écrire une fonction `recomp : int -> int list -> int` faisant le travail inverse.
3. Écrire une fonction `affiche : int -> int -> unit` affichant à l'écran tous les habitants de $[\![1,n]\!]^p$.


```
(*------------ Les yeux bandÃ©s ----------*)
let rec expo_naif x n=
  if n=0
  then 1
  else x*(expo_naif x (n-1));;

(* expo_naif : int -> int -> int = <fun>*)


let rec expo_rapide x n=
  if n=0
  then 1
  else
    let y=expo_rapide x (n/2)
    in
      if (n mod 2)=0
      then y*y
      else y*y*x;;

(* expo_rapide : int -> int -> int = <fun>*)


let rec revquadra =function
  | [] -> []
  | x::r -> (revquadra r)@[x];;

(*revquadra : 'a list -> 'a list = <fun>*)


let reverse l=
  let rec rev1 accu=function
  | []->accu
  | x::r-> rev1 (x::accu) r
  in rev1 [] l;;

(* reverse : 'a list -> 'a list = <fun> *)


let swap i j t=
  let x=t.(i)
  in
    t.(i)<-t.(j);
    t.(j)<-x;;

(* swap : int -> int -> 'a array -> unit = <fun> *)


let bubble_sort t=
  let n=Array.length t in
    for i=n-1 downto 1
    do
      for j=0 to i-1
      do
        if t.(j)>t.(j+1) then swap j (j+1) t
      done
    done;;

(* bubble_sort : 'a array -> unit = <fun> *)


let rec do_list (f:'a->unit)=function
  | []->()
  | x::r-> (f x);do_list f r;;

(* do_list : ('a -> unit) -> 'a list -> unit = <fun> *)


do_list (fun n->print_int n; print_string " ") [15;12;1];
print_string "\b";;
```

```
(*  15 12 1- : unit = ()*)


(*------------ quick sort ------------*)

let rec split order x=function
  | []->[],[]
  | y::l->
      let (l1,l2)=split order x l
      in
        if order x y
        then l1,y::l2
        else y::l1,l2;;

(*split : ('a -> 'b -> bool) -> 'a -> 'b list -> 'b list * 'b list = <fun>*)

let rec quick order=function
  | [] -> []
  | x::r->let (l1,l2)=split order x r in
      (quick order l1)@(x::(quick order l2));;

(*quick : ('a -> 'a -> bool) -> 'a list -> 'a list*)

quick (fun x y -> x-y>0) [3;-2;-42;42;23];;

(*- : int list = [42; 23; 3; -2; -42]*)



(*-------- ÃM-^InumÃ©rations ------------------*)

let rec my_map f=function
  |[] -> []
  |x::r->(f x)::(my_map f r);;

(* my_map : ('a -> 'b) -> 'a list -> 'b list = <fun> *)


let rec parties=function
  | 1->[[1];[]]
  | n->let p=parties (n-1) in
      (my_map (fun l->n::l) p)@p;;

(*parties : int -> int list list = <fun>*)

parties 3;;

(*- : int list list = [[3; 2; 1]; [3; 2]; [3; 1]; [3]; [2; 1]; [2]; [1]; []]*)


let rec mid_adj x pos l=match l with
  | []->[x]
  | y::r ->
      if pos=0 then x::l
      else y::(mid_adj x (pos-1) r);;

(*mid_adj : 'a -> int -> 'a list -> 'a list = <fun>*)

let rec permutations=function
  | 1->[[1]]
  | n->let p=permutations (n-1) and r=ref []
    in
      for pos=0 to n-1 do
        r:=(my_map (fun l->mid_adj n pos l) p)@(!r)
      done;
      !r;;

(*permutations : int -> int list list = <fun>*)

permutations 3;;
```

```
(*- : int list list =
[[1; 2; 3]; [2; 1; 3]; [1; 3; 2]; [2; 3; 1]; [3; 1; 2]; [3; 2; 1]]
*)


(*-------- ÃM-^Icritures en base b ---------------*)

let rec decomp n b t=
  if t=1 then [n]
  else (n mod b)::(decomp (n/b) b (t-1));;

(*decomp : int -> int -> int -> int list*)

decomp 10 3 5;;

(*- : int list = [1; 0; 1; 0; 0]*)

let rec recomp b=function
  | []->0
  | x::r->x+b*(recomp b r);;

(*recomp : int -> int list -> int*)


let rec affiche n p=
  for i=0 to (expo_naif n p)-1
  do
    print_string "(";
    do_list (fun x->print_int x;print_string " ") (decomp i n p);
    print_string "\b)";
    print_newline();
  done;;

(*affiche : int -> int -> unit*)

affiche 3 3;;

(* -> ce qu'on espÃ¨re ! *)
```

Tarjan et le Minotaure

Labyrinthe

On souhaite construire un labyrinthe :

- Réflechir en termes de composantes connexes, de chemins, d'accessibilité.
- Faire le lien avec Union-find vu en cours
- Programmer Union-find (au moins la version naïve et la compression de chemins ; et l'union par rang si vous voulez)
  - `MakeSet(u)` (en pratique, on va créer un gros tableau d'un coup)
  - `Find(u)` (`u` et `v` sont dans la même classe d'équivalence si `Find(u) = Find(v)`)
  - `Union(u, v)` (oui ça peut sembler bizarre pour désigner les classes d'équivalence)

Remarque : la structure de données Union-find sera simplement un tableau d'entiers (chaque noeud connaît son père) ; et cela correpond à une forêt (où les arcs sont orientés vers la racine)

Représentation

Réfléchir un peu à la représentation de votre labyrinthe :

- Comment sont numérotées les cases (et donc les noeuds du graphe sous-jacent) : un entier ? un couple ? comme vous voulez.
- Quelles vont être les coordonnées d'une case dans le plan ? il faudra peut-être utilisez des divisions euclidiennes, des modulos...
- Supposons que l'on connaisse l'ensemble des arêtes du graphe modélisant notre labyrinthe : préfère-t-on afficher un mur si l'arête n'appartient pas au graphe, ou le contraire ?

Graphics !

En théorie ça devrait fonctionner, si c'est bien installé au lycée :

```
#load "graphics.cma";;    (* <-- nécessaire ! *)

Graphics.open_graph "";;  (* ça doit ouvrir une fenêtre normalement *)
Graphics.close_graph ();; (* et la fermer *)
```

On peut aussi utiliser `open` pour ne pas avoir à réécrire `Graphics` à chaque fois :

```
#load "graphics.cma";;    (* <-- nécessaire ! *)
open Graphics;;           (* plus besoin de Graphics. ... *)

open_graph "";;
close_graph ();;
```

Pour voir toutes les fonctions disponibles :

```
#show Graphics;;          (* À partir de la version 4.02 *)
module X = Graphics;;     (* Avant la 4.02, on trichait *)
```

Premiers pas, avec des rectangles :

```
set_color black;;
fill_rect 0 0 1000 1000;;
set_color white;;
fill_rect 0 50 500 3;;    (* fill_rect x y dx dy *)
```


Vous avez tous les outils en main pour dessiner un joli labyrinthe !

Tarjan, seigneur de la Composante (Fortement) Connexe

Allez regarder l'"Algorithme de Tarjan" qui, non content d'avoir étudié un algorithme super efficace pour déterminer des composantes connexes, a aussi inventé un algorithme très élégant pour les composantes fortement connexes :

- Réfléchir à un algorithme pour les CFC.
- Trouver des contre-exemples sur lesquels votre algorithme ne fonctionne pas, ou critiquer sa complexité.
- Wikipédia > Algorithme de Tarjan.

Et après

- Des questions sur les concours peut-être ?

```
(* Labyrinthe - Tarjan *)

#load "graphics.cma";; (* sudo apt-get install ocaml-libs *)
module G = Graphics;;

let k = 3;;             (* Coefficient pour l'affichage *)
Random.self_init ();;   (* Random seed *)

(* Union-Find : Tarjan *)
let make_union_find n =
  Array.init n (fun x -> x)

let rec find x uf =
  if uf.(x) = x then x
  else let p = find uf.(x) uf in uf.(x) <- p; p;;

let union x y uf =
  let u, v = find x uf, find y uf in uf.(u) <- v;;

(* Labyrinthe *)
let creer_murs n =
  let m = Array.make (2*n*(n-1)) (0, 0, 0, 0) in
  for i = 0 to (n-1) do
    for j = 0 to (n-2) do
      m.(i*(n-1)+j) <- (i, j, i, j+1)
    done
  done;
  for i = 0 to (n-2) do
    for j = 0 to (n-1) do
      m.(n*(n-1)+i*n+j) <- (i, j, i+1, j)
    done
  done;
  m;;

let shuffle_fisher_yates t =
  for i = Array.length t - 1 downto 1 do
    let j = Random.int (i+1) and x = t.(i) in
    t.(i) <- t.(j); t.(j) <- x
  done;;

let labyrinthe n =
  let murs = creer_murs n and uf = make_union_find (n*n) in
  shuffle_fisher_yates murs;
  G.open_graph ""; G.draw_rect 0 0 (k*(2*n)) (k*(2*n));
  for i = 0 to Array.length murs - 1 do
    let x1, y1, x2, y2 = murs.(i) in let salle1,salle2 = x1*n+y1,x2*n+y2 in
    if find salle1 uf <> find salle2 uf then union salle1 salle2 uf
    else (if x1 = x2  (* donc le mur est horizontal ! *)
          then G.draw_segments [|k*(2*x1), k*(y1+y2+1), k*(2*x2+2), k*(y1+y2+1)|
]
          else G.draw_segments [|k*(x1+x2+1), k*(2*y1), k*(x1+x2+1), k*(2*y1+2)|
])
    done;;

labyrinthe 100;;
ignore (G.read_key ());; (* Pause *)
G.close_graph ();;
```