Table of Contents

TP 2 - Programmation pour la préparation à l'agrégation maths option info

- En OCaml.

```
In [1]: let print = Printf.printf;;
        Sys.command "ocaml -version";;
```

```
Out[1]: val print : ('a, out_channel, unit) format → 'a = <fun>

        The OCaml toplevel, version 4.04.2

Out[1]: - : int = 0
```

Listes

Exercice 1 : `taille`

In [2]:
```
let rec taille (liste : 'a list) : int =
    match liste with
    | [] -> 0
    | _ :: q -> (taille q) + 1
;;

taille [];;
taille [1; 2; 3];;
```

```
Out[2]: val taille : 'a list → int = <fun>
Out[2]: - : int = 0
Out[2]: - : int = 3
```

Pas sûr qu'elle soit récursive terminale, alors que celle là oui :

In [3]:
```
let taille (liste : 'a list) : int =
    let rec aux acc = function
        | [] -> acc
        | _ :: q -> aux (acc + 1) q
    in
    aux 0 liste
;;

taille [];;
taille [1; 2; 3];;
```

```
Out[3]: val taille : 'a list → int = <fun>
Out[3]: - : int = 0
Out[3]: - : int = 3
```

Solution plus ésotérique :

In [6]:
```
let taille = List.fold_left (fun acc _ -> acc + 1) 0;;

taille [];;
taille [1; 2; 3];;
```

```
Out[6]: val taille : '_a list → int = <fun>
Out[6]: - : int = 0
Out[6]: - : int = 3
```

In [7]:
```
List.length [];;
List.length [1; 2; 3];;
```

```
Out[7]: - : int = 0
Out[7]: - : int = 3
```

Exercice 2 : `concat`

In [8]:
```
let rec concatene (liste1 : 'a list) (liste2 : 'a list) : 'a list =
    match liste1 with
    | [] -> liste2
    | h :: q -> h :: (concatene q liste2)
;;

concatene [1; 2] [3; 4];;
```

```
Out[8]: val concatene : 'a list → 'a list → 'a list = <fun>
Out[8]: - : int list = [1; 2; 3; 4]
```

Autre approche, moins simple mais de même complexité en $\mathcal{O}(n)$.

In [9]:
```
let miroir (liste : 'a list) : 'a list =
    let rec aux acc = function
    | [] -> acc
    | h :: q -> aux (h :: acc) q
    in aux [] liste
;;
```

```
Out[9]: val miroir : 'a list → 'a list = <fun>
```

In [10]:
```
let concatene (liste1 : 'a list) (liste2 : 'a list) : 'a list =
    let rec aux acc l1 l2 =
        match l1 with
        | [] when l2 = [] -> acc
        | [] -> aux acc l2 []
        | h :: q -> aux (h :: acc) q l2
    in
    miroir (aux [] liste1 liste2)
;;
```

```
Out[10]: val concatene : 'a list → 'a list → 'a list = <fun>
```

In [11]:
```
concatene [1; 2] [3; 4];;
```

```
Out[11]: - : int list = [1; 2; 3; 4]
```

In [12]:
```
List.append [1; 2] [3; 4];;
```

```
Out[12]: - : int list = [1; 2; 3; 4]
```

Exercice 3 : `appartient`

In [13]:
```
let rec appartient x = function
    | [] -> false
    | h :: _ when h = x -> true
    | _ :: q -> appartient x q
;;

appartient 1 [];;
appartient 1 [1];;
appartient 1 [1; 2; 3];;
appartient 4 [1; 2; 3];;
```

```
Out[13]: val appartient : 'a → 'a list → bool = <fun>
Out[13]: - : bool = false
Out[13]: - : bool = true
Out[13]: - : bool = true
Out[13]: - : bool = false
```

In [14]:
```
List.mem 1 [];;
List.mem 1 [1];;
List.mem 1 [1; 2; 3];;
List.mem 4 [1; 2; 3];;
```

```
Out[14]: - : bool = false
Out[14]: - : bool = true
Out[14]: - : bool = true
Out[14]: - : bool = false
```

Exercice 4 : `miroir`

In [15]:
```
let miroir (liste : 'a list) : 'a list =
    let rec aux acc = function
    | [] -> acc
    | h :: q -> aux (h :: acc) q
    in aux [] liste
;;
```

```
Out[15]: val miroir : 'a list → 'a list = <fun>
```

In [16]:
```
miroir [2; 3; 5; 7; 11];;

List.rev [2; 3; 5; 7; 11];;
```

```
Out[16]: - : int list = [11; 7; 5; 3; 2]
Out[16]: - : int list = [11; 7; 5; 3; 2]
```

Exercice 5 : `alterne`

La sémantique n'était pas très claire, mais on peut imaginer quelque chose comme ça :

In [17]:
```
let alterne (liste1 : 'a list) (liste2 : 'a list) : 'a list =
    let rec aux acc l1 l2 =
        match (l1, l2) with
        | [], [] -> acc
        | [], ll2 -> acc @ ll2
        | ll1, [] -> acc @ ll1
        | h1 :: q1, h2 :: q2 -> aux (h1 :: h2 :: acc) q1 q2
    in List.rev (aux [] liste1 liste2)
;;
```

```
Out[17]: val alterne : 'a list → 'a list → 'a list = <fun>
```

In [18]:
```
alterne [1; 3; 5] [2; 4; 6];;
```

```
Out[18]: - : int list = [2; 1; 4; 3; 6; 5]
```

La complexité est linéaire en $\mathcal{O}(\max(|\text{liste 1}|, |\text{liste 2}|)$.

Mais on manque souvent la version la plus simple :

In [19]:
```
let rec alterne (l1 : 'a list) (l2 : 'a list) : 'a list =
    match l1 with
    | [] -> l2
    | t::q -> t::(alterne l2 q)
;;
```

```
Out[19]: val alterne : 'a list → 'a list → 'a list = <fun>
```

Exercice 6 : `nb_occurrences`

In [20]:
```
let nb_occurrences (x : 'a) (liste : 'a list) : int =
    let rec aux acc x = function
    | [] -> acc
    | h :: q when h = x -> aux (acc + 1) x q
    | _ :: q -> aux acc x q
    in aux 0 x liste
;;

nb_occurrences 0 [1; 2; 3; 4];;
nb_occurrences 2 [1; 2; 3; 4];;
nb_occurrences 2 [1; 2; 2; 3; 3; 4];;
nb_occurrences 5 [1; 2; 3; 4];;
```

```
Out[20]: val nb_occurrences : 'a → 'a list → int = <fun>

Out[20]: - : int = 0

Out[20]: - : int = 1

Out[20]: - : int = 2

Out[20]: - : int = 0
```

Autre approche, avec un `List.fold_left` bien placé :

In [21]:
```
let nb_occurrences (x : 'a) : 'a list -> int =
    List.fold_left (fun acc y -> if x = y then (acc + 1) else acc) 0
;;

nb_occurrences 0 [1; 2; 3; 4];;
nb_occurrences 2 [1; 2; 3; 4];;
nb_occurrences 2 [1; 2; 2; 3; 3; 4];;
nb_occurrences 5 [1; 2; 3; 4];;
```

```
Out[21]: val nb_occurrences : 'a → 'a list → int = <fun>

Out[21]: - : int = 0

Out[21]: - : int = 1

Out[21]: - : int = 2

Out[21]: - : int = 0
```

Exercice 7 : `pairs`

C'est un filtrage :

In [22]:
```
let pairs = List.filter (fun x -> x mod 2 = 0);;
```

```
Out[22]: val pairs : int list → int list = <fun>
```

In [23]:
```
pairs [1; 2; 3; 4; 5; 6];;
pairs [1; 2; 3; 4; 5; 6; 7; 100000];;
pairs [1; 2; 3; 4; 5; 6; 7; 100000000000];;
pairs [1; 2; 3; 4; 5; 6; 7; 1000000000000000000];;
```

```
Out[23]: - : int list = [2; 4; 6]

Out[23]: - : int list = [2; 4; 6; 100000]

Out[23]: - : int list = [2; 4; 6; 100000000000]

Out[23]: - : int list = [2; 4; 6; 1000000000000000000]
```

Exercice 8 : `range`

In [24]:
```
let range (n : int) : int list =
    let rec aux acc = function
    | 0 -> acc
    | n -> aux (n :: acc) (n - 1)
    in aux [] n
;;
```

Out[24]:
```
val range : int → int list = <fun>
```

In [25]:
```
range 30;;
```

Out[25]:
```
- : int list =
[1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16; 17; 18; 19; 20; 21;
 22; 23; 24; 25; 26; 27; 28; 29; 30]
```

In [26]:
```
(* Vous avez parfois eu du mal à construire la liste des entiers de a à b
   ca serait bon de savoir le faire car ca peut vous donner des exemples
   d'entree pour vos algos *)

let entiers a b =
  let rec construit x =
    if x > b
    then []
    else x::(construit (x + 1))
  in construit a
;;
```

Out[26]:
```
val entiers : int → int → int list = <fun>
```

In [27]:
```
let premiers_entiers = entiers 0;; (* Bel exemple de currification *)

entiers 4 10;;
premiers_entiers 7;;
```

Out[27]:
```
val premiers_entiers : int → int list = <fun>
```

Out[27]:
```
- : int list = [4; 5; 6; 7; 8; 9; 10]
```

Out[27]:
```
- : int list = [0; 1; 2; 3; 4; 5; 6; 7]
```

Exercice 9 : `premiers`

Plusieurs possibilités. Un filtre d'Erathosthène marche bien, ou une filtration. Je ne vais pas utiliser de tableaux donc on est un peu réduit d'utiliser une filtration (filtrage ? *pattern matching*)

In [28]:
```
let racine (n : int) : int =
    int_of_float (floor (sqrt (float_of_int n)))
;;

racine 17;;
```

Out[28]:
```
val racine : int → int = <fun>
```

Out[28]:
```
- : int = 4
```

In [29]:
```
let estDivisible (a : int) (b : int) : bool =
    (a mod b) = 0
;;

estDivisible 10 2;;
estDivisible 10 6;;
estDivisible 10 5;;
```

```
Out[29]: val estDivisible : int → int → bool = <fun>
Out[29]: - : bool = true
Out[29]: - : bool = false
Out[29]: - : bool = true
```

In [30]:
```
let range2 (debut : int) (fin : int) (taille : int) : int list =
    let rec aux acc = function
    | n when n > fin -> acc
    | n -> aux (n :: acc) (n + taille)
    in
    List.rev (aux [] debut)
;;
```

```
Out[30]: val range2 : int → int → int → int list = <fun>
```

In [31]:
```
range2 2 12 3;;
```

```
Out[31]: - : int list = [2; 5; 8; 11]
```

Une version purement fonctionnelle est moins facile qu'une version impérative avec une référence booléenne (rappel : pas de `break` dans les boucles `for` en OCaml...).

In [32]:
```
let estPremier (n : int) : bool =
    List.fold_left (fun b k -> b && (not (estDivisible n k))) true (range2 2 (racine n) 1)
;;
```

```
Out[32]: val estPremier : int → bool = <fun>
```

In [33]:
```
let premiers (n : int) : int list =
    List.filter estPremier (range2 2 n 1)
;;
```

```
Out[33]: val premiers : int → int list = <fun>
```

In [34]:
```
premiers 10;;
```

```
Out[34]: - : int list = [2; 3; 5; 7]
```

In [35]:
```
premiers 100;;
```

```
Out[35]: - : int list =
         [2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 53; 59; 61; 67; 71;
          73; 79; 83; 89; 97]
```

Quelques tris par comparaison

On fera les tris en ordre croissant.

In [5]:
```
let test = [3; 1; 8; 4; 5; 6; 1; 2];;
```

```
Out[5]: val test : int list = [3; 1; 8; 4; 5; 6; 1; 2]
```

Exercice 10 : Tri insertion

In [37]:
```
let rec insere (x : 'a) : 'a list -> 'a list = function
    | [] -> [x]
    | t :: q ->
        if x <= t then
            x :: t :: q
        else
            t :: (insere x q)
;;
```

Out[37]:
```
val insere : 'a → 'a list → 'a list = <fun>
```

In [38]:
```
let rec tri_insertion : 'a list -> 'a list = function
    | [] -> []
    | t :: q -> insere t (tri_insertion q)
;;
```

Out[38]:
```
val tri_insertion : 'a list → 'a list = <fun>
```

In [39]:
```
tri_insertion test;;
```

Out[39]:
```
- : int list = [1; 1; 2; 3; 4; 5; 6; 8]
```

Complexité en temps $\mathcal{O}(n^2)$.

Exercice 11 : Tri insertion générique

In [40]:
```
let rec insere2 (ordre : 'a -> 'a -> bool) (x : 'a) : 'a list -> 'a list = function
    | [] -> [x]
    | t :: q ->
        if ordre x t then
            x :: t :: q
        else
            t :: (insere2 ordre x q)
;;
```

Out[40]:
```
val insere2 : ('a → 'a → bool) → 'a → 'a list → 'a list = <fun>
```

In [41]:
```
let rec tri_insertion2 (ordre : 'a -> 'a -> bool) : 'a list -> 'a list = function
    | [] -> []
    | t :: q -> insere2 ordre t (tri_insertion2 ordre q)
;;
```

Out[41]:
```
val tri_insertion2 : ('a → 'a → bool) → 'a list → 'a list = <fun>
```

In [16]:
```
let ordre_croissant a b = a <= b;;
```

Out[16]:
```
val ordre_croissant : 'a → 'a → bool = <fun>
```

In [43]:
```
tri_insertion2 ordre_croissant test;;
```

Out[43]:
```
- : int list = [1; 1; 2; 3; 4; 5; 6; 8]
```

In [44]:
```
let ordre_decroissant = (>=);;
```

Out[44]:
```
val ordre_decroissant : 'a → 'a → bool = <fun>
```

In [45]:
```
tri_insertion2 ordre_decroissant test;;
```

Out[45]:
```
- : int list = [8; 6; 5; 4; 3; 2; 1; 1]
```

Exercice 12 : Tri selection

In [46]:
```
let selectionne_min (l : 'a list) : ('a * 'a list) =
    let rec cherche_min min autres = function
        | [] -> (min, autres)
        | t :: q ->
            if t < min then
                cherche_min t (min :: autres) q
            else
                cherche_min min (t :: autres) q
    in
    match l with
    | [] -> failwith "Selectionne_min sur liste vide"
    | t :: q -> cherche_min t [] q
;;
```

Out[46]:
```
val selectionne_min : 'a list → 'a * 'a list = <fun>
```

In [47]:
```
selectionne_min test;;
```

Out[47]:
```
- : int * int list = (1, [2; 1; 6; 5; 4; 8; 3])
```

In [48]:
```
let rec tri_selection : 'a list -> 'a list = function
    | [] -> []
    | l ->
        let (min, autres) = selectionne_min l in
        min :: (tri_selection autres)
;;
```

Out[48]:
```
val tri_selection : 'a list → 'a list = <fun>
```

In [49]:
```
tri_selection test;;
```

Out[49]:
```
- : int list = [1; 1; 2; 3; 4; 5; 6; 8]
```

Complexité en temps : $\mathcal{O}(n^2)$.

Exercices 13, 14, 15 : Tri fusion

In [3]:
```
let print_list (liste : int list) : unit =
    print_string "[";
    List.iter (fun i -> print_int i; print_string "; " ) liste;
    print_endline "]";
;;
```

Out[3]:
```
val print_list : int list → unit = <fun>
```

In [7]:
```
test;;
print_list test;;
```

Out[7]:
```
- : int list = [3; 1; 8; 4; 5; 6; 1; 2]
```

Out[7]:
```
- : unit = ()
[3; 1; 8; 4; 5; 6; 1; 2; ]
```

In [50]:
```
let rec separe : 'a list -> ('a list * 'a list) = function
    | [] -> ([], [])
    | [x] -> ([x], [])
    | x :: y :: q ->
        let (a, b) = separe q
        in (x::a, y::b)
;;

separe test;;
```

Out[50]:
```
val separe : 'a list → 'a list * 'a list = <fun>
```

Out[50]:
```
- : int list * int list = ([3; 8; 5; 1], [1; 4; 6; 2])
```

In [51]:
```
let rec fusion (l1 : 'a list) (l2 : 'a list) : 'a list =
    match (l1, l2) with
    | (l, []) | ([], l) -> l  (* syntaxe concise pour deux cas identiques *)
    | (x::a, y::b) ->
        if x <= y then
            x :: (fusion a (y :: b))
        else
            y :: (fusion (x :: a) b)
;;

fusion [1; 3; 7] [2; 3; 8];;
```

Out[51]:
```
val fusion : 'a list → 'a list → 'a list = <fun>
```

Out[51]:
```
- : int list = [1; 2; 3; 3; 7; 8]
```

In [52]:
```
let rec tri_fusion : 'a list -> 'a list = function
    | [] -> []
    | [x] -> [x] (* ATTENTION A NE PAS OUBLIER CE CAS *)
    | l ->
        let a, b = separe l in
        fusion (tri_fusion a) (tri_fusion b)
;;
```

Out[52]:
```
val tri_fusion : 'a list → 'a list = <fun>
```

In [53]:
```
tri_fusion test;;
```

Out[53]:
```
- : int list = [1; 1; 2; 3; 4; 5; 6; 8]
```

Complexité en temps $\mathcal{O}(n \log n)$.

Listes : l'ordre supérieur

Je ne corrige pas les questions qui étaient traitées dans le TP1.

Exercice 16 : `applique`

In [54]:
```
let rec applique f = function
    | [] -> []
    | h :: q -> (f h) :: (applique f q)
;;
```

Out[54]:
```
val applique : ('a → 'b) → 'a list → 'b list = <fun>
```

Exercice 17

In [55]:
```
let premiers_carres_parfaits (n : int) : int list =
  applique (fun x -> x * x) (entiers 1 n)
;;
```

Out[55]:
```
val premiers_carres_parfaits : int → int list = <fun>
```

In [56]:
```
premiers_carres_parfaits 12;;
```

Out[56]:
```
- : int list = [1; 4; 9; 16; 25; 36; 49; 64; 81; 100; 121; 144]
```

Exercice 18 : `itere`

In [57]:
```
let rec itere (f : 'a -> unit) = function
  | [] -> ()
  | h :: q -> begin
    f h;
    itere f q
  end
;;
```

Out[57]:
```
val itere : ('a → unit) → 'a list → unit = <fun>
```

Exercice 19

In [58]:
```
let print = Printf.printf

let f x = print "%i\n" x;;
```

Out[58]:
```
val print : ('a, out_channel, unit) format → 'a = <fun>
```

Out[58]:
```
val f : int → unit = <fun>
```

In [62]:
```
let affiche_liste_entiers (liste : int list) =
  print "Debut\n";
  itere (print "%i\n") liste;
  print "Fin\n";
  flush_all ();
;;

affiche_liste_entiers [1; 2; 4; 5];;
```

Out[62]:
```
val affiche_liste_entiers : int list → unit = <fun>
```

Out[62]:
```
- : unit = ()
```

```
Debut
1
2
4
5
Fin
```

Exercice 20 : `qqsoit` et `ilexiste`

In [63]:
```
let rec qqsoit (pred : 'a -> bool) = function
  | [] -> true (* piege ! *)
  | h :: q -> (pred h) && (qqsoit pred q)
  (* le && n'évalue pas le deuxième si le premier argument est false
     donc ceci est efficace et récursif terminal.
  *)
;;
```

Out[63]:
```
val qqsoit : ('a → bool) → 'a list → bool = <fun>
```

In [64]:
```
let rec ilexiste (pred : 'a -> bool) = function
    | [] -> false
    | h :: q -> (pred h) || (ilexiste pred q)
    (* le || n'évalue pas le deuxième si le premier argument est true
       donc ceci est efficace et récursif terminal.
    *)
;;
```

Out[64]:
```
val ilexiste : ('a → bool) → 'a list → bool = <fun>
```

In [65]:
```
qqsoit (fun x -> (x mod 2) = 0) [1; 2; 3; 4; 5];;

ilexiste (fun x -> (x mod 2) = 0) [1; 2; 3; 4; 5];;
```

Out[65]:
```
- : bool = false
```

Out[65]:
```
- : bool = true
```

Exercice 21 : `appartient` version 2

In [66]:
```
let appartient x = ilexiste (fun y -> x = y);;

let appartient x = ilexiste ((=) x);; (* syntaxe simplifiée par curification *)
```

Out[66]:
```
val appartient : 'a → 'a list → bool = <fun>
```

Out[66]:
```
val appartient : 'a → 'a list → bool = <fun>
```

In [67]:
```
let toutes_egales x = qqsoit ((=) x);;
```

Out[67]:
```
val toutes_egales : 'a → 'a list → bool = <fun>
```

In [68]:
```
appartient 1 [1; 2; 3];;
appartient 5 [1; 2; 3];;

toutes_egales 1 [1; 2; 3];;
toutes_egales 2 [2; 2; 2];;
```

Out[68]:
```
- : bool = true
```

Out[68]:
```
- : bool = false
```

Out[68]:
```
- : bool = false
```

Out[68]:
```
- : bool = true
```

Exercice 22 : `filtre`

In [69]:
```
let rec filtre (pred : 'a -> bool) : 'a list -> 'a list = function
    | [] -> []
    | h :: q when pred h -> h :: (filtre pred q)
    | _ :: q -> filtre pred q
;;
```

Out[69]:
```
val filtre : ('a → bool) → 'a list → 'a list = <fun>
```

In [70]:
```
filtre (fun x -> (x mod 2) = 0) [1; 2; 3; 4; 5];;

filtre (fun x -> (x mod 2) != 0) [1; 2; 3; 4; 5];;
filtre (fun x -> (x mod 2) <> 0) [1; 2; 3; 4; 5];; (* syntaxe non conseillée *)
```

Out[70]:
```
- : int list = [2; 4]
```

Out[70]:
```
- : int list = [1; 3; 5]
```

Out[70]:
```
- : int list = [1; 3; 5]
```

Exercice 23

Je vous laisse trouver pour `premiers`.

In [71]:
```
let pairs = filtre (fun x -> (x mod 2) = 0);;
let impairs = filtre (fun x -> (x mod 2) != 0);;
```

```
Out[71]: val pairs : int list → int list = <fun>
Out[71]: val impairs : int list → int list = <fun>
```

Exercice 24 : `reduit`

In [72]:
```
let rec reduit (tr : 'a -> 'b -> 'a) (acc : 'a) (liste : 'b list) : 'a =
    match liste with
    | [] -> acc
    | h :: q -> reduit tr (tr acc h) q
;;
```

```
Out[72]: val reduit : ('a → 'b → 'a) → 'a → 'b list → 'a = <fun>
```

Très pratique pour calculer des sommes, notamment.

Exercice 25 : `somme`, `produit`

In [73]:
```
let somme = reduit (+) 0;;

somme [1; 2; 3; 4; 5];;
List.fold_left (+) 0 [1; 2; 3; 4; 5];;
```

```
Out[73]: val somme : int list → int = <fun>
Out[73]: - : int = 15
Out[73]: - : int = 15
```

In [74]:
```
let produit = reduit ( * ) 1;;

produit [1; 2; 3; 4; 5];;
List.fold_left ( * ) 1 [1; 2; 3; 4; 5];;
```

```
Out[74]: val produit : int list → int = <fun>
Out[74]: - : int = 120
Out[74]: - : int = 120
```

Exercice 26 : `miroir` version 2

In [75]:
```
let miroir = reduit (fun a b -> b :: a) [];;
```

```
Out[75]: val miroir : '_a list → '_a list = <fun>
```

In [76]:
```
miroir [2; 3; 5; 7; 11];;

List.rev [2; 3; 5; 7; 11];;
```

```
Out[76]: - : int list = [11; 7; 5; 3; 2]
Out[76]: - : int list = [11; 7; 5; 3; 2]
```

In [77]:
```
miroir [2.; 3.; 5.; 7.; 11.];;
```

```
File "[77]", line 1, characters 8-10:
Error: This expression has type float but an expression was expected of type
         int

¤M¤M¤M¤M¤M# miroir [2.; 3.; 5.; 7.; 11.];;
```

Arbres

Exercice 27

In [5]:
```
type 'a arbre_bin0 = Feuille0 of 'a | Noeud0 of ('a arbre_bin0) * 'a * ('a arbre_bin0);;
```

Out[5]:
```
type 'a arbre_bin0 =
    Feuille0 of 'a
  | Noeud0 of 'a arbre_bin0 * 'a * 'a arbre_bin0
```

In [6]:
```
let rec arbre_complet_entier (n : int) : int arbre_bin0 =
  match n with
  | n when n < 2 -> Feuille0 0
  | n -> Noeud0((arbre_complet_entier (n / 2)), n, (arbre_complet_entier (n / 2)))
;;

arbre_complet_entier 4;;
```

Out[6]:
```
val arbre_complet_entier : int → int arbre_bin0 = <fun>
```

Out[6]:
```
- : int arbre_bin0 =
Noeud0 (Noeud0 (Feuille0 0, 2, Feuille0 0), 4,
 Noeud0 (Feuille0 0, 2, Feuille0 0))
```

Autre variante, plus simple :

In [7]:
```
type arbre_bin = Feuille | Noeud of arbre_bin * arbre_bin;;
```

Out[7]:
```
type arbre_bin = Feuille | Noeud of arbre_bin * arbre_bin
```

In [8]:
```
let arbre_test = Noeud (Noeud (Noeud (Feuille, Feuille), Feuille), Feuille);;
```

Out[8]:
```
val arbre_test : arbre_bin =
  Noeud (Noeud (Noeud (Feuille, Feuille), Feuille), Feuille)
```

Exercice 28

Compte le nombre de feuilles et de sommets.

In [10]:
```
let rec taille : arbre_bin -> int = function
  | Feuille -> 1
  | Noeud(x, y) -> 1 + (taille x) + (taille y)
;;
```

Out[10]:
```
val taille : arbre_bin → int = <fun>
```

In [11]:
```
taille arbre_test;;
```

Out[11]:
```
- : int = 7
```

Exercice 29

In [12]:
```
let rec hauteur : arbre_bin -> int = function
    | Feuille -> 0
    | Noeud(x, y) -> 1 + (max (hauteur x) (hauteur y)) (* peut etre plus simple *)
;;
```

Out[12]:
```
val hauteur : arbre_bin → int = <fun>
```

In [13]:
```
hauteur arbre_test;;
```

Out[13]:
```
- : int = 3
```

Exercice 30

Bonus.

Parcours d'arbres binaires

Exercice 31

In [14]:
```
type element_parcours = F | N;;

type parcours = element_parcours list;;
```

Out[14]:
```
type element_parcours = F | N
```

Out[14]:
```
type parcours = element_parcours list
```

Exercice 32 : Parcours naifs (complexité quadratique)

In [15]:
```
let rec parcours_prefixe : arbre_bin -> element_parcours list = function
    | Feuille -> [F]
    | Noeud (g, d) -> [N] @ (parcours_prefixe g) @ (parcours_prefixe d)
;;

parcours_prefixe arbre_test;;
```

Out[15]:
```
val parcours_prefixe : arbre_bin → element_parcours list = <fun>
```

Out[15]:
```
- : element_parcours list = [N; N; N; F; F; F; F]
```

In [16]:
```
let rec parcours_postfixe : arbre_bin -> element_parcours list = function
    | Feuille -> [F]
    | Noeud(g, d) -> (parcours_postfixe g) @ (parcours_postfixe d) @ [N]
;;

parcours_postfixe arbre_test;;
```

Out[16]:
```
val parcours_postfixe : arbre_bin → element_parcours list = <fun>
```

Out[16]:
```
- : element_parcours list = [F; F; N; F; N; F; N]
```

In [17]:
```
let rec parcours_infixe : arbre_bin -> element_parcours list = function
    | Feuille -> [F]
    | Noeud(g, d) -> (parcours_infixe g) @ [N] @ (parcours_infixe d)
;;

parcours_infixe arbre_test;;
```

```
Out[17]: val parcours_infixe : arbre_bin → element_parcours list = <fun>
Out[17]: - : element_parcours list = [F; N; F; N; F; N; F]
```

Pourquoi ont-ils une complexité quadratique ? La concaténation (`@`) ne se fait pas en temps constant mais linéaire dans la taille de la première liste.

Exercice 33 : Parcours linéaires

On ajoute une fonction auxiliaire et un argument `vus` qui est une liste qui stocke les élements observés dans l'ordre du parcours

In [18]:
```
let parcours_prefixe2 a =
    let rec parcours vus = function
        | Feuille -> F :: vus
        | Noeud(g, d) -> parcours (parcours (N :: vus) g) d
    in List.rev (parcours [] a)
;;

parcours_prefixe2 arbre_test;;
```

```
Out[18]: val parcours_prefixe2 : arbre_bin → element_parcours list = <fun>
Out[18]: - : element_parcours list = [N; N; N; F; F; F; F]
```

In [21]:
```
let parcours_postfixe2 a =
    let rec parcours vus = function
        | Feuille -> F :: vus
        | Noeud(g, d) -> N :: (parcours (parcours vus g) d)
    in List.rev (parcours [] a)
;;

parcours_postfixe2 arbre_test;;
```

```
Out[21]: val parcours_postfixe2 : arbre_bin → element_parcours list = <fun>
Out[21]: - : element_parcours list = [F; F; N; F; N; F; N]
```

In [22]:
```
let parcours_infixe2 a =
    let rec parcours vus = function
        | Feuille -> F :: vus
        | Noeud(g, d) -> parcours (N :: (parcours vus g)) d
    in List.rev (parcours [] a)
;;

parcours_infixe2 arbre_test;;
```

```
Out[22]: val parcours_infixe2 : arbre_bin → element_parcours list = <fun>
Out[22]: - : element_parcours list = [F; N; F; N; F; N; F]
```

Exercice 34 : parcours en largeur et en profondeur

In [23]:
```
let parcours_largeur a =
   let file = Queue.create () in
   (* fonction avec effet de bord sur la file *)
   let rec parcours () =
      if Queue.is_empty file
      then []
      else match Queue.pop file with
         | Feuille -> F :: (parcours ())
         | Noeud(g, d) -> begin
            Queue.push g file;
            Queue.push d file;
            N :: (parcours ())
         end
   in
   Queue.push a file;
   parcours ()
;;

parcours_largeur arbre_test;;
```

Out[23]:
```
val parcours_largeur : arbre_bin → element_parcours list = <fun>
```

Out[23]:
```
- : element_parcours list = [N; N; F; N; F; F; F]
```

En remplaçant la file par une pile (`Stack`), on obtient le parcours en profondeur, avec la même complexité.

In [24]:
```
let parcours_profondeur a =
   let file = Stack.create () in
   (* fonction avec effet de bord sur la file *)
   let rec parcours () =
      if Stack.is_empty file
      then []
      else match Stack.pop file with
         | Feuille -> F :: (parcours ())
         | Noeud(g, d) -> begin
            Stack.push g file;
            Stack.push d file;
            N :: (parcours ())
         end
   in
   Stack.push a file;
   parcours ()
;;

parcours_profondeur arbre_test;;
```

Out[24]:
```
val parcours_profondeur : arbre_bin → element_parcours list = <fun>
```

Out[24]:
```
- : element_parcours list = [N; F; N; F; N; F; F]
```

Exercice 35 et fin

In [25]:
```
(* Reconstruction depuis le parcours prefixe *)

let test_prefixe = parcours_prefixe2 arbre_test;;
```

Out[25]:
```
val test_prefixe : element_parcours list = [N; N; N; F; F; F; F]
```

In [26]:
```
(* L'idée de cette solution est la suivante :
   j'aimerais une fonction récursive qui fasse le travail;
   le problème c'est que si on prend un parcours prefixe, soit il commence
   par F et l'arbre doit être une feuille; soit il est de la forme N::q
   où q n'est plus un parcours prefixe mais la concaténation de DEUX parcours
   prefixe, on ne peut donc plus appeler la fonction sur q.
   On va donc écrire une fonction qui prend une liste qui contient plusieurs
   parcours concaténé et qui renvoie l'arbre correspondant au premier parcours
   et ce qui n'a pas été utilisé : *)

let reconstruit_prefixe parcours =
    let rec reconstruit = function
    | F :: p -> (Feuille, p)
    | N :: p ->
        let (g, q) = reconstruit p in
        let (d, r) = reconstruit q in
        (Noeud(g, d), r)
    | [] -> failwith "pacours invalide"
    in
    match reconstruit parcours with
    | (a, []) -> a
    | _ -> failwith "parcours invalide"
;;

reconstruit_prefixe test_prefixe;;
reconstruit_prefixe (N :: F :: F :: test_prefixe);; (* échoue *)
```

Out[26]:
```
val reconstruit_prefixe : element_parcours list → arbre_bin = <fun>
```

Out[26]:
```
- : arbre_bin = Noeud (Noeud (Noeud (Feuille, Feuille), Feuille), Feuille)

Exception: Failure "parcours invalide".
Raised at file "pervasives.ml", line 32, characters 22-33
Called from file "toplevel/toploop.ml", line 180, characters 17-56
```

In [27]:
```
(* Reconstruction depuis le parcours en largeur *)

(* Ce n'est pas évident quand on ne connait pas. L'idée est de se servir d'une file
   pour stocker les arbres qu'on reconstruit peu à peu depuis les feuilles. La file
   permet de récupérer les bons sous-arbres quand on rencontre un noeud *)

let largeur_test = parcours_largeur arbre_test;;
```

Out[27]:
```
val largeur_test : element_parcours list = [N; N; F; N; F; F; F]
```

In [28]:
```
let reconstruit_largeur parcours =
    let file = Queue.create () in
    (* Fonction avec effets de bord *)
    let lire_element = function
    | F -> Queue.push Feuille file
    | N ->
        let d = Queue.pop file in
        let g = Queue.pop file in
        Queue.push (Noeud(g, d)) file
    in
    List.iter lire_element (List.rev parcours);
    if Queue.length file = 1 then
        Queue.pop file
    else
        failwith "parcours invalide"
;;

reconstruit_largeur largeur_test;;
```

Out[28]:
```
val reconstruit_largeur : element_parcours list → arbre_bin = <fun>
```

Out[28]:
```
- : arbre_bin = Noeud (Noeud (Noeud (Feuille, Feuille), Feuille), Feuille)
```

In [29]:
```
(* Le même algorithme (enfin presque, modulo interversion de g et d)
   avec une pile donne une autre version de la reconstruction du parcours prefixe *)

let reconstruit_prefixe2 parcours =
    let pile = Stack.create () in
    let lire_element = function
    | F -> Stack.push Feuille pile
    | N ->
        let g = Stack.pop pile in
        let d = Stack.pop pile in
        Stack.push (Noeud(g, d)) pile
    in
    List.iter lire_element (List.rev parcours);
    if Stack.length pile = 1 then
        Stack.pop pile
    else
        failwith "parcours invalide"
;;

reconstruit_prefixe2 test_prefixe;;
```

```
Out[29]: val reconstruit_prefixe2 : element_parcours list → arbre_bin = <fun>
```

```
Out[29]: - : arbre_bin = Noeud (Noeud (Noeud (Feuille, Feuille), Feuille), Feuille)
```

Conclusion

Fin. À la séance prochaine.