Lycée Faidherbe, 2020-2021

# T.P. d'informatique

**MPSI option informatique**

Version du 23 juin 2021

# Table des matières

# TP I

# Fonctions


## Résumé

*Les questions marquées (avec *) ne demandent pas de programmation.*
*On pourra en utiliser les résultats si besoin.*


On pose $E_n = \{0, ..., n-1\}$ pour $n \in \mathbb{N}^*$.
On s'intéresse aux fonctions de $E_n$ dans $E_p$. En numérotant ses éléments, tout ensemble fini peut être représenté par un ensemble $E_n$.
La suite des valeurs de $f$, $f(0)$, $f(1)$, ..., $f(n-1)$, peut être enregistrée dans un tableau `t` : `t.(i)` prend $f(i)$ pour valeur pour tout $i \in E_n$.

## 1 Cas général

Pour déterminer une fonction on a besoin de son ensemble de départ, de son ensemble d'arrivée et des valeurs. Un tableau permet de déterminer les valeurs et l'ensemble de départ : si `t` est tableau de taille $n$, il peut représenter une fonction $f_t$ définie sur $E_n$ par $f_t(i)$ est la valeur de `t.(i)`.
Pour déterminer l'ensemble d'arrivée on doit ajouter l'entier $p$ : **dans cette partie**, une fonction sera définie par un couple `(t, p)` où `t` est un tableau et $p$ un entier.
Par exemple, la fonction $f_0$ qui à $x \in E_7$ associe $x^2 \bmod 10 \in E_{10}$
est représentée par `f0 = (u, 10)` avec `u = [| 0; 1; 4; 9; 6; 5; 6|]`.

**Exercice 1 — Vérification**
*Écrire une fonction* `verif (t, p)` *qui renvoie un booléen* `true` *ou* `false` *selon que le couple représente bien une à valeurs dans* $E_p$.

Dans la suite une variable `f` sera le nom d'un couple `(t, p)` représentant une fonction de $E_n$ vers $E_p$ où $n$ est la taille du tableau `t`.
On pourra accéder aux composantes `t` et `p` dans une fonction `ma_fonction f`

- soit en les définissant par `let (t, p) = f in`
- soit par `let t = fst f in` et `let p = snd f in`
- soit en écrivant la définition de la fonction sous la forme

```
let ma_fonction (t, p) =
```

**Exercice 2 — Croissance**
*Écrire une fonction* `est_croissante f` *qui renvoie* `true` *ou* `false` *selon que la fonction est croissante ou non.*

**Exercice 3 — Composition**
*Écrire une fonction* `composition (t1, p1) (t2, p2)` *qui renvoie le couple associé associé à la fonction* $f_1 \circ f_2$ *où* $f_1$ *et* $f_2$ *sont les fonctions associées aux couples* `(t1, p1)` *et* `(t2, p2)`.
*La fonction doit vérifier que la composition est possible (l'ensemble d'arrivée de* $f_2$ *doit être inclus dans l'ensemble de départ de* $f_1$*) en renvoyant une erreur le cas échéant.*

**Exercice 4 — Injection**
*Écrire une fonction* `est_injective f` *qui renvoie* `true` *ou* `false` *selon que la fonction est injective ou non.*
*On emploiera la définition de l'injectivité : une solution plus rapide est proposée à la question 6*

**Exercice 5 — Surjection**
*Écrire une fonction* `est_surjective f` *qui renvoie* `true` *ou* `false` *selon que la fonction est surjective ou non.*
*Ici encore on pourra se contenter d'une solution qui applique la définition.*

**Exercice 6 — Amélioration**
*En utilisant un tableau qui compte le nombre antécédents pour chaque élément de l'ensemble d'arrivée écrire des fonctions qui effectuent un nombre d'instruction de l'ordre de $n+p$ pour déterminer si une fonction est injective (resp. surjective).*

**Exercice 7 — Inverse d'une bijection**
*Écrire une fonction* `inverse f` *qui, si* `f = (t, n)` *et* `t` *est un tableau de taille $n$ représentant une bijection, renvoie la bijection réciproque.*

On voit dans le cours de mathématiques que, si $f$ est injective (resp. surjective) de $A$ vers $B$, il existe une fonction $g$ de $B$ vers $A$ telle que $g \circ f = Id_A$ (resp. $f \circ g = Id_B$). $g$ est un **inverse à gauche** (resp. un **inverse à droite**) de $f$.

**Exercice 8 — Pseudo_inverse**
*Écrire une fonction* `pseudo_inverse f` *qui renvoie une fonction* `g` *avec la propriété suivante :*
- *si $f$ est injective alors $g$ est un inverse à gauche de $f$,*
- *si $f$ est surjective alors $g$ est un inverse à droite de $f$.*

# 2 Points fixes

On ne considère maintenant et **dans toute la suite** que des fonctions de $E_n$ vers $E_n$.

**Exercice 9 — Itération**
Écrire une fonction `itere k f` qui détermine la fonction $f^k$.

Un élément $x \in E_n$ est un **point fixe** de la fonction $f$ si et seulement si $f(x) = x$.

**Exercice 10 — Points fixes**
*Écrire une fonction* `nombrePtFixe f` *qui donne le nombre de points fixes d'une fonction.*

On suppose maintenant que la fonction est **croissante** de $E_n$ dans $E_n$.

**Exercice 11 * — Croissance stricte**
*Quelles sont les fonctions strictement croissantes de $E_n$ dans $E_n$ ?*

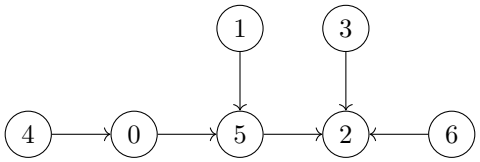
**Exercice 12 * — Existence**
*Prouver que si $f$ est croissante de $E_n$ dans $E_n$ et s'il existe deux points $i$ et $j$ de $E_n$ tels que $i \leqslant f(i) \leqslant f(j) \leqslant j$ alors il existe $k$ tel que $i \leqslant k \leqslant j$ et $f(k) = k$.*
*En déduire que toute fonction croissante sur $E_n$ admet un point fixe.*

**Exercice 13 — Calcul rapide d'un point fixe**
*Écrire une fonction* `point_fixe f` *qui retourne un point fixe de la fonction, supposée croissante, en un temps de calcul* ***logarithmique*** *en la taille $n$ du tableau.*

# 3 Point attracteur

**Définition :**
*Un élément $z \in E_n$ est dit **attracteur principal** de $f : E_n \to E_n$ si et seulement si $z$ est un point fixe de $f$ et, pour tout $x \in E_n$, il existe un entier $k \geqslant 0$ tel que $f^k(x) = z$.*

**Exemple**
La fonction définie par `t = [| 5; 5; 2; 2; 0; 2; 2]` admet 2 comme attracteur principal.




**Exercice 14 * — Propriétés**
*Montrer que si $f$ de $E_n$ vers $E_n$ admet un attracteur principal $z$ alors*
1. *$z$ est le seul point fixe de $f$.*
2. *$z$ est un attracteur principal de $f^m$ pour tout $m \in \mathbb{N}^*$.*
3. *$f^{n-1}(x) = z$ pour tout $x \in E_n$.*

**Exercice 15 — Existence d'un attracteur**
*Écrire une fonction* `admetAttracteur f` *qui renvoie* `true` *si et seulement si la fonction admet un attracteur principal. (On pourra utiliser la point 3 ci-dessus.)*

Si $f$ admet un attracteur $z$ alors le **temps de convergence** de $f$ en $x \in E_n$ est le plus petit entier $k \geqslant 0$ tel que $f^k(x) = z$ ; on le note $\mathrm{tc}(f, x)$

**Exercice 16 — Calcul du temps de convergence**
Écrire une fonction `tempsConvergence f x` qui renvoie le temps de convergence en $x$ de la fonction dont on sait qu'elle admet un attracteur principal.

**Exercice 17 — Calcul des temps de convergence**
*Écrire une fonction* `listeTC f` *qui renvoie le tableau des temps de convergence $tc(f, x)$ pour $0 \leqslant x < n$ en sachant que la fonction admet un attracteur principal.*
*On demande d'écrire une fonction dont le nombre d'instruction effectuées est majoré par $Kn$ où $K$ est une constante. Il sera plus simple d'écrire une fonction auxiliaire récursive.*

**Exercice 18 * — Cas des fonctions croissantes**
*$f$ est une fonction croissante de $E_n$ vers $E_n$.*
*Prouver que $f$ admet un point attracteur si et seulement si elle n'a qu'un seul point fixe.*
*Montrer que, dans ce cas, le temps de convergence maximal est atteint pour $x = 0$ ou $x = n - 1$.*

INDICATION : on pourra montrer que si $f$ et croissante et admet un seul point fixe $p$ alors $f(x) > x$ pour $0 \leqslant x < p$ et $f(x) < x$ pour $p < x < n$.

# 4 Solutions

**Exercice 1 (Solution)** Un algorithme simple utilise une boucle `for`.

```
let verif (t, p) =
   let n = Array.length t in
   let reponse = ref true in
   for i = 0 to (n-1) do
      if t.(i) < 0 || t.(i) >= p
      then reponse := false done;
   !reponse;;
```

Si on veut cesser de chercher dès que la réponse devient fausse, il faut utiliser une boucle `while`.

```
let verif (t, p) =
   let n = Array.length t in
   let reponse = ref true in
   let i = ref 0 in
   while !i < n && !reponse do
      if t.(!i) < 0 || t.(!i) >= p
      then reponse := false;
      i := !i + 1 done;
  !reponse;;
```

On peut écrire plus court, au risque de la lisibilité.

```
let verif t =
   let n = Array.length t in
   let i = ref 0 in
   while !i < n &&  t.(!i) >= 0 && t.(!i) < p do i := !i + 1
      done;
   !i = n;;
```

**Exercice 2 (Solution)**

```
let est_croissante (t, p) =
   let n = Array.length t in
   let reponse = ref true in
   for i = 0 to (n-2) do
      if t.(i) > t.(i+1) then reponse := false done;
   !reponse;;
```

On peut faire les mêmes améliorations qu'à la question précédente.

**Exercice 3 (Solution)**

```
let composition (t1, p1) (t2, p2) =
   let n1 = Array.length t1 in
   let n2 = Array.length t2 in
   if n1 < p2
   then failwith "La composition est impossible"
   else begin
        let t = Array.make n2 0 in
        for i = 0 to (n2-1) do t.(i) <- t1.(t2.(i)) done;
        (t, p1) end;;
```

**Exercice 4 (Solution)**

```
let est_injective (t, p) =
   let n = Array.length t in
   let reponse = ref true in
   for i = 0 to (n-2) do
      for j = (i+1) to (n-1) do
         if t.(i) = t.(j) then reponse := false done done;
   !reponse;;
```

**Exercice 5 (Solution)** On commence par une fonction qui teste si une valeur est atteinte.

```
let est_atteint k t =
   let n = Array.length t in
   let reponse = ref false in
   for i = 0 to (n-1) do
      if t.(i) = k then reponse := true done;
   !reponse;;
```

On peut alors tester tous les éléments de l'ensemble d'arrivée.

```
let est_surjective (t, p) =
   let reponse = ref true in
   for k = 0 to (p-1) do
      if not (est_atteint k t) then reponse := false done;
   !reponse;;
```

**Exercice 6 (Solution)** On commence par calculer le tableau des nombres d'antécédents.

```
let antecedents (t, p) =
   let n = Array.length t in
   let ante = Array.make p 0 in
   for i = 0 to (n-1) do
      let j = t.(i) in ante.(j) <- ante.(j) + 1 done;
   ante;;
```

On peut écrire les fonctions

```
let est_injective (t, p) =
   let ante = antecedents (t, p) in
   let reponse = ref true in
   for k = 0 to (p-1) do
      if ante.(k) > 1 then reponse := false done;
   !reponse;;
```

```
let est_surjective (t, p) =
   let ante = antecedents (t, p) in
   let reponse = ref true in
   for k = 0 to (p-1) do
      if ante.(k) = 0 then reponse := false done;
   !reponse;;
```

**Exercice 7 (Solution)** Il suffit de calculer un tableau d'antécédents ; on l'initialise avec une valeur possible, 0, et on place `i` en `u.(j)` si `t.(i)` vaut `j`.

```
let inverse (t, n) =
   let inv = Array.make n 0 in
   for i = 0 to (n-1) do inv.(t.(i)) <- i done;
   (inv, n);;
```

**Exercice 8 (Solution)** C'est la même fonction que l'inverse mais ici on peut calculer plusieurs fois un antécédent et il peut ne pas y en avoir.

```
let pseudo_inverse (t, p) =
  let n = Array.length t in
  let ante = Array.make p 0 in
  for i = 0 to (n-1) do ante.(t.(i)) <- i done;
  (ante, n);;
```

**Exercice 9 (Solution)**

```
let itere k (t, n) =
   let n = snd f in
   let fk = ref (Array.init n (fun i -> i), n) in
   for i = 1 to k do
      fk := composition f !fk done;
  !fk;;
```

On peut aussi écrire de nouveau la composition :

```
let itere k (t, n) =
   let tk = Array.init n (fun i -> i) in
   for i = 1 to k do
      for j = 0 to (n-1) do
         tk.(j) <- t.(tk.(j)) done done;
   (tk, n);;
```

**Exercice 10 (Solution)**

```
let nbPtFixes (t, n) =
   let reponse = ref 0 in
   for i = 0 to (n-1) do
      if t.(i) = i
      then incr reponse done;
   !reponse;;
```

**Exercice 11 (Solution)**
Soit $f$ strictement croissante de $E_n$ vers $E_n$ : $f(x+1) > f(x)$ donc $f(x+1) \geqslant f(x)+1$.
On pose $g(x) = f(x)-x$ : $g$ est définie de $E_n$ vers $\mathbb{N}$ et $g(x+1) = f(x+1)-(x+1) \geqslant f(x)+1-x-1 = g(x)$ donc $g$ est croissante. Or $g(0) = f(0) \geqslant 0$ et $g(n-1) = f(n-1)-(n-1) \leqslant n-1-(n-1) = 0$ donc $g(x) = 0$ pour tout $x \in E_n$ et $f$ est l'identité.

**Exercice 12 (Solution)**
On considère $A = \{x \in \{i, i+1, \ldots, j\} \ ; \ f(x) \geqslant x\}$. $A$ est non vide car il contient $i$.
Comme $A$ est fini non vide, il admet un maximum : on pose $k = \max(A)$, on a $f(k) \geqslant k$.

- Si $k = j$ on a $f(j) \geqslant j$. Or $f(j) \leqslant j$ ; on en déduit $f(k) = k$.
- Si $k < j$ alors $k+1 \leqslant j$ et $k+1$ n'appartient pas à $A$ d'où $f(k+1) < k+1$ c'est-à-dire $f(k+1) \leqslant k$.
  On a alors $k \leqslant f(k) \leqslant f(k+1) \leqslant k$ en raison de la croissance de $f$.
  Il y a donc égalité ; en particulier $f(k) = k$.

Dans les deux cas on a un point fixe.

On a $f(0) \in E_n$ donc $f(0) \geqslant 0$ et $f(n-1) \in E_n$ donc $f(n-1) \leqslant n-1$.
On peut appliquer le résultat ci-dessus si $f$ est croissante : $f$ admet un point fixe.

**Exercice 13 (Solution)**

```
let point_fixe t =
   let mini = ref 0 in
   let maxi = ref (Array.length t) in
   let c = ref ((!mini + !maxi)/2) in
   while (t.(!c) <> !c) do
      if t.(!c) > !c
      then mini := !c + 1
      else maxi := !c - 1;
      c := (!mini + !maxi)/2 done;
  !c;;
```

L'algorithme est plus simple à écrire récursivement.

```
let point_fixe t =
   let rec ptFixe a b =
      let c = (a + b)/2 in
      if t.(c) = c
      then c
      else (if t.(c) > c
            then ptFixe (c+1) b
            else ptFixe a (c-1)) in
   ptFixe 0 (Array.length t);;
```

**Exercice 14 (Solution)**

1. Si $y$ est un point fixe de $f$ alors $f^p(y) = y$ pour tout $p$ et il existe $k$ tel que $f^k(y) = z$ donc $y = z$.
2. Si $f^k(x) = z$ avec $z$ point fixe alors $f^l(x) = z$ pour tout $l \geqslant k$.
   En particulier $\left(f^m\right)^k(x) = f^{km}(x) = z$ car $km \geqslant k$. De plus $f^m(z) = z$
   Ainsi $z$ est un point fixe de $f^k$ et, pour tout $x \in E_n$, il existe un entier $k \geqslant 0$ tel que $\left(f^m\right)^k(x) = f^{km}(x) = z$ : $f^m$ admet un attracteur principal.
3. $x$, $f(x)$, $f^2(x)$, $\ldots f^n(x)$ sont $n+1$ valeurs appartenant à $E_n$.
   Comme le cardinal de $E_n$ est $n$, une même valeur au moins est obtenue deux fois :
   $f^p(x) = f^q(x)$ avec, par exemple $p < q$.
   Si on note $y = f^p(x)$ on a $f^{q-p}(y)$ donc $y$ est un point fixe de $f^{p-q}$.
   Le 2. indique que $f^{p-q}$ admet un attracteur principal et d'après le 1. son seul point fixe est $z$. On a donc $z = y = f^p(x)$ avec $0 \leqslant p < q \leqslant n$ donc $f^{n-1}(x) = z$ car $p \leqslant n-1$.

**Exercice 15 (Solution)**

```
let admetAttracteur (t, n) =
   let p = nbPtFixes t in
   if p <> 1
   then false
   else let tt = itere t (n-1) in
        let reponse = ref true in
        for i = 1 to (n-1) do
           if tt.(i) <> tt.(0)
           then reponse := false done;
        !reponse;;
```

**Exercice 16 (Solution)**

```
let tempsConvergence (t, n) x =
   let temps = ref 0 in
   let y = ref x in
   while t.(!y) <> !y do
      temps := !temps + 1;
      y := t.(!y) done;
   !temps;;
```

**Exercice 17 (Solution)** On commence par calculer le point fixe, $p$.
On définit un tableau de taille $n$ contenant des $-1$, `tempsC`. Il contiendra les temps de convergence.
On calcule le temps de convergence d'un premier point et on marque les temps de convergence de ses itérés. On donne la valeur 0 à `tc.(p)`.
Pour tout point on itère à partir de ce point jusqu'à arriver à un point dont on a déjà calculé le temps de convergence (dans `tempsc`).
On parcourt de nouveau les itérés pour écrire les temps de convergence.

```
let listeTC (t, n) =
   let tempsC = Array.make n (-1) in
   let k = tempsConvergence t 0 in
   let x = ref 0 in
   for i = 0 to k do tempsC.(!x) <- k - i; x:= t.(!x) done;
   for i = 0 to (n-1) do
     let x = ref i in
     let k = ref 0 in
     while tempsC.(!x) = -1 do k := !k + 1; x := t.(!x) done;
     let tc = tempsC.(!x) in
     let x = ref i in
     for j = 0 to (!k - 1)
        do tempsC.(!x) <- tc + !k - j; x := t.(!x) done;
   done;
   tempsC;;
```

Dans une fonction récursive on écrit les temps de convergence lors du retour.

```
let listeTC t =
   let n = Array.length t in
   let tempsC = Array.make n (-1) in
   let pf = ref 0 in
   while t.(!pf) <> !pf do pf := t.(!pf) done;
   tempsC.(!pf) <- 0;
   let rec calculTC i =
      if tempsC.(i) = -1
      then begin
           calculTC (t.(i));
           tempsC.(i) <- tempsC.(t.(i)) + 1 end in
   for i = 0 to (n-1) do calculTC i done;
   tempsC;;
```

**Exercice 18 (Solution)**
Si $f$ admet un point attracteur alors elle admet un point fixe unique, c'est le 1. de l'exercice 14.

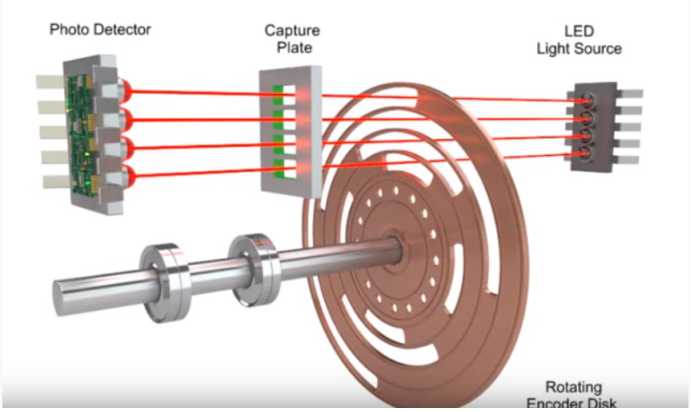
On suppose que $f$ est croissante et admet un point fixe unique $p$.

1. On prouve la propriété suggérée par l'indication.
   On suppose $p > 0$.
   Si on avait $f(x) \leqslant x$ avec $x < p$ alors, comme on a $f(0) \geqslant 0$ on aurait deux points 0 et $x$ tels que $0 \leqslant f(0) \leqslant f(x) \leqslant x$. Le résultat de l'exercice 12 prouverait alors que $f$ admet un point fixe entre 0 et $x$, ce qui est impossible car on a $x < p$.
   Ainsi on a $f(x) > x$ pour tout $x < p$.
   De même on a $f(x) < x$ pour tout $x > p$.
2. Soit $x < p$ ; on a $f$ croissante donc $f(x) \leqslant f(p) = p$. De plus on a vu qu'on a $x < f(x)$.
   On prouve alors, par récurrence qu'on a $f^k(x) \leqslant f^{k+1}(x) \leqslant p$ pour tout $k$.
   La suite $\left(f^k(x)\right)_{k\in\mathbb{N}}$ est croissante à valeurs dans $\{0, 1, \ldots, p\}$ (ensemble fini) donc est stationnaire à partir d'un certain rang. On arrive alors à un point fixe, ce ne peut être que $p$ : il existe $k$ tel que $f^k(x) = p$.
   De même, pour $x > p$, la suite $\left(f^k(x)\right)_{k\in\mathbb{N}}$ est décroissante minorée par $p$.
   La suite est donc stationnaire et parvient au seul point fixe : il existe $k$ tel que $f^k(x) = p$.
   Ainsi $p$ est un point attracteur de $f$.
3. Soit $k = \mathrm{tc}(f, 0)$ : on a $f^k(0) = p$.
   Pour $0 \leqslant x \leqslant p$ on a $p = f^k(0) \leqslant f^k(x) \leqslant f^k(p) = p$ en raison de la croissance de $f^k$ donc $f^k(x) = p$. Ainsi on a $\mathrm{tc}(f, x) \leqslant k = \mathrm{tc}(f, 0)$ pour tout $x \in \{0, 1, \ldots, p\}$.
   De même $\mathrm{tc}(f, x) \leqslant \mathrm{tc}(f, n-1)$ pour tout $x \in \{p, p+1, \ldots, n-1\}$.
   On en d"duit que le temps de convergence maximal est atteint pour $x = 0$ ou $x = n-1$.

TP II

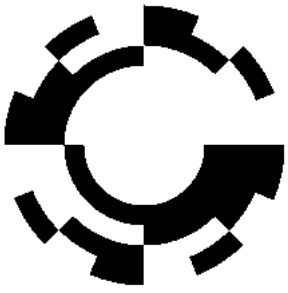
# Codes de Gray

On se propose ici de présenter une énumération des entiers (de 0 à $2^p - 1$) qui a des applications pratiques dans la fabrication des encodeurs. Un encodeur est un dispositif mécanique qui permet de déterminer une position ou un angle absolus




Le dispositif laisse passer (ou réfléchit) $p$ faisceaux de lumière vers des capteurs et on obtient ainsi $p$ valeurs 0 ou 1 (la lumière parvient ou ne parvient pas), ce qui permet de différentier $2^p$ positions ou $2^p$ angles.

Le plus immédiat serait de coder selon la décomposition binaire mais alors le passage d'une valeur à une autre pourrait nécessiter plusieurs changements de valeurs ; en cas d'incertitude de lecture (entre deux positions) on aurait des valeurs nombreuses possibles. Dans l'exemple ci-contre, entre les positions 0 et 15 toutes les valeurs sont possibles. Le but des questions est de définir un codage tel que, entre deux positions contiguës, il n'y a qu'un seul changement de valeur (comme dans l'illustration ci-dessus).



Les questions mêleront les tableaux pour la suite des $2^p$ entiers et les listes pour l'écriture des décompositions binaires.
Les exercices marqués d'une étoile sont des questions théoriques à résoudre en dehors du TP.

# 1 Codage

Les différentes positions sont codées par une suite de 0 et de 1 que l'on représentera par l'entier dont la suite est la décomposition en base 2. On utilisera une liste d'entiers pour le codage binaire, elles commenceront par le bit de poids faible, c'est-à-dire par le coefficient $a_0$ pour $n = \sum_{k=0}^{p-1} a_k 2^k$ et devront être minimales, le dernier terme doit être 1. Par exemple la conversion de 13 doit donner `[1; 0; 1; 1]`, la conversion de 100 doit donner `[0; 0; 1; 0; 0; 1; 1]`.

**Exercice 1**
*Écrire une fonction* **récursive** `binaire : int -> int list` *qui convertit un entier en sa représentation binaire.*

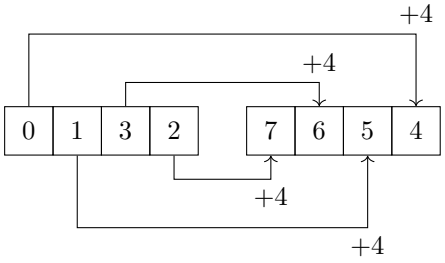
**Exercice 2**
*Écrire la fonction réciproque* `entier : int list-> int` *qui convertit une liste de 0 et 1 en un entier. La fonction doit être récursive et ne calculera pas les puissance de 2.*

On va maintenant définir, pour chaque $p$, un tableau des entiers de 0 à $2^p - 1$ avec la propriété supplémentaire que deux entiers consécutifs (ainsi que le premier et le dernier) ont des décompositions binaires qui ne diffèrent qu'en un seul bit. Par exemple, pour $p = 3$, le tableau `[|0; 1; 3; 7; 5; 4; 6; 2]` convient parce que les décompositions binaires (sur 3 bits) sont `[0;0;0], [1;0;0], [1;1;0], [1;1;1], [1;0;1], [0;0;1], [0;1;1], [0;1;0]`

La **suite de Gray**, $G = (g_p)_{p\in\mathbb{N}}$ est définie par $g_0 = 0$ et, par récurrence forte,

$$g_{2^p+k} = 2^p + g_{2^p-1-k} \text{ pour } 0 \leqslant k < 2^p$$

On a donc $g_1 = g_{1+0} = 1 + g_{1-1-0} = 1$, $g_2 = g_{2+0} = 2 + g_{2-1-0} = 3$,
$g_3 = g_{2+1} = 2 + g_{2-1-1} = 2$, $g_4 = g_{4+0} = 4 + g_{4-1-0} = 6$, ...
On remarque que, si on connaît $g_0$, $g_1$, ..., $g_{2^p-1}$ alors on définit $g_{2^p}$, ..., $g_{2^{p+1}-1}$ en additionnant $2^p$ aux $2^p$ éléments déjà définis en les écrivant dans l'ordre inverse.

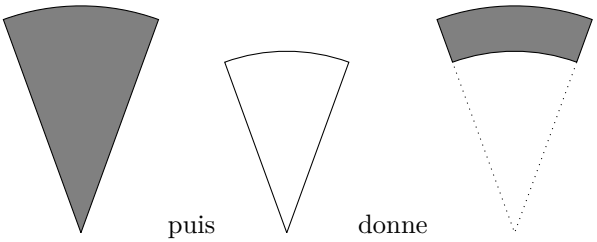



**Exercice 3** *
*Prouver que la suite $G_p = (g_0, g_1, \ldots, g_{2^p-1})$ verifie que chaque entier de 0 à $2^p - 1$ est atteint une fois et que deux entiers consécutifs (ainsi que le premier et le dernier) ont des décompositions binaires qui ne diffèrent qu'en un seul bit.*

**Exercice 4**
*Écrire une fonction* `gray : int -> int array` *telle que* `gray p` *calcule le tableau des valeurs des $G_p$. On notera que la valeur de $2^{p-1}$ est accessible sans calcul dans la récurrence, c'est la longueur de* `gray (p-1)`.

# 2 Dessin d'une roue

Dans cette partie nous allons écrire les instructions qui permettent de dessiner une roue.
OCaml contient un modules qui permet de faire des dessins élémentaires.
Nous allons utiliser les instructions qui dessinent (et remplissent) un arc de cercle, en fait un arc d'ellipse. Comme on veut tracer des portions concentriques on trace une portion remplie jusqu'au centre puis on repasse en blanc la portion plus petite. Cela impose de dessiner depuis l'extérieur.




```
1  # load "graphics.cma";;
2  open Graphics;;
3
4  let x0 = 300;;
5  let y0 = 300;;
6  let d = 10;;
7  let r0 = 50;;
8
9  let secteur ns k i =
10   let r = r0 + i*d in
11   set_color black;
12   fill_arc x0 y0 (r + d) (r + d) (360*k/ns) (360*(k+1)/ns);
13   set_color white;
14   fill_arc x0 y0 r r (360*k/ns) (360*(k+1)/ns);;
15
16 open_graph " 600x600";;
17 (* Dessiner des trucs*)
18 wait_next_event [Button_down];;
19 close_graph ();;
```

**Ligne 1** On charge la bibliothèque graphique, qui n'est pas chargée par défaut. Noter le `#`.

**Ligne 2** Les fonctions de la bibliothèque graphique sont intégrées, on n'aura pas besoin de les écrire avec le préfixe `Graphics`.

**Lignes 4 à 7** On définit les constantes : largeur et hauteur de la fenêtre graphique, coordonnées du centre, largeur d'une portion et rayon au centre. On tracera la $i$-ième portion concentrique entre $r_0 + i.d$ et $r_0 + (i+1).d$.

**ligne 9** On définit une fonction de tracé : `ns` est le nombre total de secteurs, `k` est le rang du secteur (de 0 à `ns - 1`) et $i$ est le rang dans les portions concentriques.

**ligne 11** On définit la couleur : noir.

**Ligne 12** On trace l'arc d'ellipse remplie : les deux premiers paramètres sont les coordonnées (entières) du centre, les deux suivantes sont les demi-rayons (entiers) de l'ellipse, s'ils sont égaux on a une portions de cercle, les deux derniers paramètres sont les angles qui délimitent l'arc en degrés (entiers !).

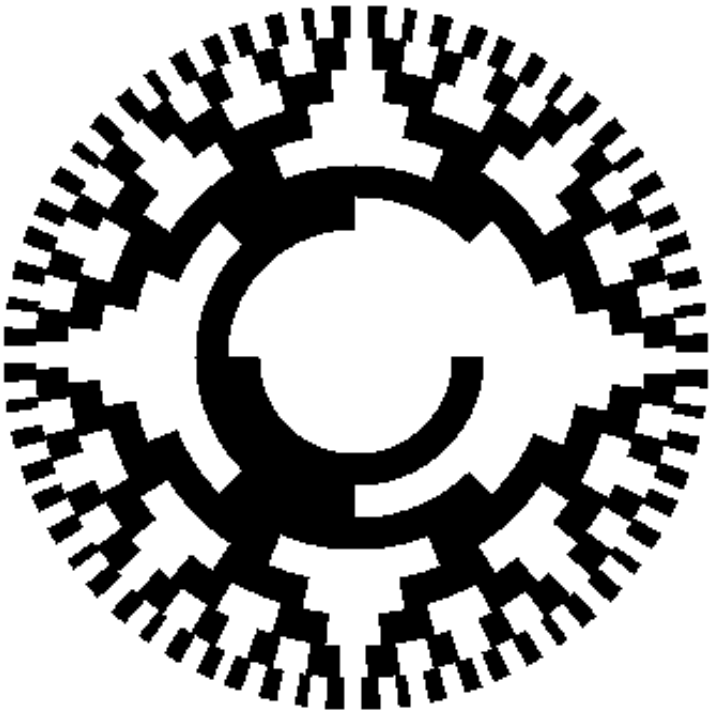
**lignes 13 et 14** Même chose pour la petite portion en blanc.

**ligne 16** On ouvre une fenêtre graphique.

**ligne 18** Cette fonction permet de laisser la fenêtre ouverte tant qu'on n'y a pas cliqué, c'est utile en mode exécution de script comme dans `gedit`.

**ligne 19** On ferme la fenêtre.

**Exercice 5**
*Écrire une fonction* `roue_gray : int -> unit` *qui telle que* `roue_gray p` *trace les secteurs (valeurs 1 de la liste binaire) correspondant au code de Gray* $G_p$.



# 3 Décodage

Lorsque l'appareil lit une position ou un angle il obtient la décomposition binaire d'un entier qui n'est pas le rang dans le tableau.
Il est donc nécessaire de décoder cet entier c'est-à-dire passer de $g_n$ à $n$.
Pour $2^{p-1} \leqslant n < 2^p$, on a vu qu'on a aussi $2^{p-1} \leqslant g_n < 2^p$.
On peut alors noter $n = \sum_{k=0}^{p-1} a_k 2^k$ et $g_n = \sum_{k=0}^{p-1} b_k 2^k$ avec $a_{p-1} = b_{p-1} = 1$ : les bits de poids forts sont les mêmes.

**Exercice 6** *
*Prouver qu'on a* $b_k \equiv a_k + a_{k+1}$ *modulo* 2 *pour* $0 \leqslant k < p-1$.

**Exercice 7**
*Écrire une fonction* `nombre_gray : int-> int` *qui calcule* $g_n$ *directement à partir de* $n$ *en utilisant les décompositions binaires.*

**Exercice 8**
*Écrire la fonction réciproque* `from_gray : int-> int` *qui calcule* $n$ *à partie de* $g_n$.

# 4 Solutions

**Exercice 1 (Solution)**

```
let rec binaire n =
  match n with
  |0 -> []
  |n -> (n mod 2) :: (binaire (n/2));;
```

**Exercice 2 (Solution)**

```
let rec entier liste =
  match liste with
  |[] -> 0
  |t::q -> t + 2*(entier q);;
```

**Exercice 3 (Solution)** On prouve le résultat par récurrence, $G_0 = (0)$ vérifie trivialement les propriétés.
Si $G_p$ vérifie les propriétés alors $\{g_0, \dots, g_{2^p-1}\} = \{0, 1, \dots, 2^p - 1\}$ et $\{g_{2^p}, \dots, g_{2^{p+1}-1}\}$ est obtenu en ajoutant $2^p$ aux valeurs donc décrit $\{2^p, \dots, 2^p + 2^p - 1\}$. Ainsi Tout entier de $\{0, \dots, 2^{p+1} - 1\}$ est atteint dans $G_{p+1}$.

- Pour $0 \leqslant i < 2^p - 1$, $g_i$ et $g_{i+1}$ sont dans $G_p$ donc, par hypothèse de récurrence, ne différent qu'en un bit.
- $g_{2^p} = g_{2^p-1} + 2^p$ donc les développement ne diffèrent que par le bit de poids $p$.
- Pour $2^p < i < 2^{p+1} - 1$, $g_i = g_j + 2^p$ et $g_{i+1} = g_{j-1} + 2^p$ avec $j = 2^p - 1 - k$ pour $i = 2^p + k$ donc $1 \leqslant j < 2^p - 2$. Ainsi $g_j$ et $g_{j-1}$ ne différent qu'en un bit donc $g_i$ et $g_{i+1}$ ne difèrent qu'en ce même bit puisqu'on a simplement ajouté le bit de poids $p$.
- $g_{2^{p+1}-1} = g_0 + 2^p$ donc les deux valeurs extrêmes ne diffèrent que par le bit de poids $p$.

Ainsi $G_{p+1}$ vérifient les propriétés d'où le résultat par récurrence sur $p$.

**Exercice 4 (Solution)**

```
let rec gray p =
  match p with
  |0 -> [|0|]
  |p -> let t = gray (p-1) in
        let puiss2 = Array.length t in
        let g = Array.make (2*puiss2) 0 in
        for i = 0 to (puiss2 - 1) do
          g.(i) <- t.(i);
          g.(i + puiss2) <- puiss2 + t.(puiss2 - 1 - i) done;
  g;;
```

**Exercice 5 (Solution)** On commence par le tracé d'une portion à partir d'une liste.

```
let rec portion liste nd k i =
  match liste with
  |[] -> ()
  |1::q -> secteur nd k i; portion q nd k (i-1)
  |_::q -> portion q nd k (i-1);;
```

On finit par le tracé.

```
let roue_gray p =
let t = gray p in
let nd = Array.length t in
open_graph " 600x600";
for i  = 0 to (nd - 1) do
portion (binaire t.(i)) nd i p done;
let _ = wait_next_event [Button_down] in close_graph ();;
```

**Exercice 6 (Solution)**

Pour $n < 2^{p-1}$, on peut toujours écrire $n = \sum_{k=0}^{p-1} a_k 2^k$ et $g_n = \sum_{k=0}^{p-1} b_k 2^k$.

On pose $k_0$ tel que $a_{k_0} = 1$ et $a_k = 0$ pour $k > k_0$. On a alors

- $b_k = a_k = 0$ pour $k > k_0$,
- $b_{k_0} = a_{k_0} = 1$,
- $b_k \equiv a_k + a_{k+1}$ modulo 2 pour $k < k_0$.

En conclusion, si la propriété est vrai pour $p \leqslant p_0$ alors, pour tout $n < 2^p$, les décompositions $n = \sum_{k=0}^{p-1} a_k 2^k$ et $g_n = \sum_{k=0}^{p-1} b_k 2^k$ vérifient $a_{p-1} = b_{p-1}$ et $b_k \equiv a_k + a_{k+1}$ modulo 2 pour $0 \leqslant k < p-1$.

On prouve alors le résultat demandé par récurrence sur $p$.

Il n'y a rien à prouver pour $p = 0$, et le résultat est trivial pour $p = 1$ car $g_1 = 1$.

On suppose la propriété vraie pour $p$ ; soit $n$ tel que $2^p \leqslant n < 2^{p+1}$ ; on a $g_n = 2^p + g_{2^p-1-m}$ avec $m = n - 2^p$. On note $n = \sum_{k=0}^{p} a_k 2^k$ donc $m = \sum_{k=0}^{p-1} a_k 2^k$, on enlève le terme $2^p$.

On a alors $2^p - 1 - m = \sum_{k=0}^{p-1} 2^k - \sum_{k=0}^{p-1} a_k 2^k = \sum_{k=0}^{p-1} (1 - a_k) 2^k$.

On écrit $g_{2^p-1-m} = \sum_{k=0}^{p-1} b'_k 2^k$ et on a $b'_{p-1} = 1 - a_{p-1}$ et

$b'_k \equiv (1 - a_k) + (1 - a_{k+1}) = a_k + a_{k+1} + 2.(1 - a_k - a_{k+1}) \equiv a_k + a_{k+1}$ modulo 2.

La décomposition de $g_n$ est $g_n = \sum_{k=0}^{p-1} b'_k 2^k + 2^p$ : on a bien la propriété demandée.

**Exercice 7 (Solution)** On commence par la conversion en binaire, puis on traduit

```
let rec gray_bin liste =
  match liste with
  |t1::t2::q -> (t1+t2) :: gray_bin (t2::q)
  |_  -> liste;;

let nombre_gray n = entier (gray_bin (binaire n));;
```

**Exercice 8 (Solution)** La conversion en binaire est moins simple

```
let rec bin_gray liste =
  match liste with
  |[] -> []
  |t::q -> begin match bin_gray q with
                 |[] -> [t]
                 |t1::q1 -> ((t+t1) mod 2) :: t1 :: q1 end ;;

let from_gray n = entier (bin_gray (binaire n));;
```

# TP III

# Polynômes

Dans ce T.P. nous allons manipuler les polynômes en choisissant une représentation **creuse** : on ne garde que les coefficients non nuls. On se restreint à des polynômes à coefficients **entiers**, on évite ainsi les opérations `+.`, `-.`, `*.`, `/.`

- Un polynôme est représenté par une liste de couples d'entiers
  `[(a`$_p$`, n`$_p$`); (a`$_{p-1}$`, n`$_{p-1}$`); ...; (a`$_1$`, n`$_1$`); (a`$_0$`, n`$_0$`)]`
  avec les conditions $a_i \neq 0$ pour tout $i$ et $n_p > n_{p-1} > \cdots > n_1 > n_0 \geqslant 0$.
  La liste représente le polynôme $a_pX^{n_p} + a_{p-1}X^{n_{p-1}} + \cdots + a_1X^{n_1} + a_0X^{n_0}$.
- Par exemple $P = 1 - 2X + 4X^3$ est représenté par `[4, 3; -2, 1; 1, 0]`.
  On notera que les parenthèses pour délimiter un couple ne sont pas obligatoires, on a alors une succession de `,` et de `;`.
- En particulier le polynôme nul est représenté par la liste vide `[]`.
- On définit le type de ces polynômes

```
type polynome = (int * int) list;;
```

  On pourra forcer ce typage en donnant un paramètre sous la forme `(p : polynome)` et un imposant le type pour le résultat en ajoutant `: polynome` avant le `=` d'une définition
- En particulier le polynôme nul est représenté par la liste vide `[]`.

**Exercice 1 — Vérification**
*Écrire une fonction* `validation p` *qui vérifie qu'une liste de couples d'entiers est une bonne représentation d'un polynôme : non nullité des coefficients et stricte décroissance des puissances.*

```
validation  : polynome -> bool
```

**N.B.** Le cas terminal de cette fonction récursive est le cas d'une liste à un élément ; c'est un cas (peu courant pour une liste) ou le pattern-matching admet 3 motifs :

```
match liste with
|[] ->
|[x] ->
|x::y::q ->
```

# 1 Opérations

**Exercice 2 — Évaluation**
*Écrire une fonction* `evaluation p a` *qui renvoie la valeur en a du polynôme $P$ représenté par* `p`.
*Il sera utile d'écrire une fonction qui calcule $a^n$ à partir de $a$ et de $n$.*

```
puissance : int -> int -> int
eval : polynome -> int -> int
```

**Exercice 3 — Somme de polynômes**
*Écrire une fonction* `plus p1 p2` *qui renvoie la représentation de la somme des polynômes représentés respectivement par* `p1` *et* `p2`

```
plus : polynome -> polynome -> polynome
```

**Exercice 4 — Produit de polynômes**
*Écrire une fonction* `fois p1 p2` *qui calcule une représentation du produit des polynômes représentés respectivement par* `p1` *et* `p2`.
*Il peut être utile d'écrire une fonction qui calcule le produit d'un monôme par un polynôme, le monôme étant représenté par un couple d'entiers.*

```
p_monome : int * int -> polynome -> polynome
fois : polynome -> polynome -> polynome
```

**Exercice 5 — Dérivée**
*Écrire une fonction* `derivee p` *qui renvoie une représentation de $P'$ quand $P$ est le polynôme représenté par* `p`.

```
derivee : polynome -> polynome
```

Les polynômes de Tchebychev sont définis par $T_n\big(\cos(x)\big) = \cos(nx)$.
On peut prouver qu'on a $T_0 = 1$, $T_1 = X$ et $T_n = 2X.T_{n-1} - T_{n-2}$ pour $n \geqslant 2$.

**Exercice 6 — Polynômes de Tchebychev**
*Écrire une fonction* `tch n` *qui renvoie une représentation de $T_n$.*
*Bien que la récursivité ne soit pas méthode la plus efficace ici, on écrira une fonction récursive.*

```
tch : int -> polynome
```

# 2 Division euclidienne

Dans cette partie nous étudions la division de polynômes. Nous supposerons que le polynôme diviseur est non nul et **unitaire**, le coefficient du terme de plus haut degré doit être 1.

La division euclidienne du polynôme $A$ par le polynôme $B$ est le couple de polynômes $Q$ et $R$ tels que $A = B.Q + R$ et $\deg(R) < \deg(B)$.
On admet que de tels polynômes existent et sont uniques.

Pour calculer les termes de la division euclidienne
1. on détermine les degrés de $A$, $p$, et de $B$, $q$,
2. si on a $p < q$ alors $Q = 0$ et $R = A$,
3. sinon
   (a) on détermine le coefficient de plus haut degré de $A$, $a$,
   (b) on calcule $A_1 = A - a.X^{p-q}.B$,
   (c) on calcule récursivement $A_1 = B.Q_1 + R$,
   (d) on a $A = (a.X^{p-q} + Q_1).B + R$.

La récursivité est possible car le degré de $A_1$ est strictement inférieur à celui de $A$.

**Exercice 7 — Division euclidienne**
*Écrire une fonction* `division a b` *qui renvoie des représentants du quotient et du reste de la division euclidienne de A par B respectivement représentés par* `a` *et* `b`.

```
division : polynome -> polynome -> polynome * polynome
```

Les polynômes cyclotomiques sont des polynômes définis par C.F. Gauss en 1801
Ils sont employés dans plusieurs domaines mathématiques : théorie de Galois, construction à la règle et au compas, ... Ils sont définis par $\Phi_n = \prod_{k=1,k\wedge n=1}^{n} \left(X - e^{2ik\pi/n}\right)$.
On prouve que ce sont des polynômes à coefficients entiers, irréductibles dans $\mathbb{Q}[X]$.
On peut les construire récursivement, à l'aide de la propriété ;

$$\Phi_1 = X - 1 \quad \Phi_n = \frac{X^n - 1}{\prod_{k|n,k<n} \Phi_k}$$

**Exercice 8 — Calcul**
*Écrire une fonction* `cycloto n` *qui renvoie* $\Phi_n$.

```
cycloto : int -> polynome
```

# 3 Solutions

**Exercice 1 (Solution)**

```
let rec validation (p:polynome) =
   match p with
   |[] -> true
   |[(a, n)] -> a <> 0
   |(a, n) :: (b, m) :: q -> a <> 0 && n > m && validation ((b
      , m) :: q);;
```

**Exercice 2 (Solution)**

```
let rec puissance a k =
  if k = 0 then 1
           else a * (puissance a (k-1));;

let rec eval (p:polynome) x =
  match p with
  |[] -> 0
  |(a, n)::q -> a*(puissance x n) + (eval q x);;
```

**Exercice 3 (Solution)** On prend le terme de plus haut de degré des deux polynômes puis on continue récursivement. Si les deux termes sont de même degré on doit étudier le cas particulier d'une somme nulle pour les coefficients.

```
let rec plus (p1:polynome) (p2:polynome) : polynome =
  match (p1, p2) with
  |[], _ -> p2
  |_, [] -> p1
  |(a1, n1)::q1, (a2, n2)::q2 when n1 > n2
                              -> (a1, n1) :: (plus q1 p2)
  |(a1, n1)::q1, (a2, n2)::q2 when n1 < n2
                              -> (a2, n2) :: (plus p1 q2)
  |(a1, n1)::q1, (a2, n2)::q2 when a1 + a2 = 0 -> (plus q1 q2)
  |(a1, n1)::q1, (a2, n2)::q2 -> (a1+a2, n1) :: (plus q1 q2);;
```

**Exercice 4 (Solution)**

```
let rec p_monome (a, n) (p:polynome) : polynome =
  match p with
  |[] -> []
  |(b, m)::q -> (a*b, n+m) :: (p_monome (a, n) q);;

let rec fois (p1:polynome) (p2:polynome) : polynome =
  match p1 with
  |[] -> []
  |m::q -> plus (p_monome m p2) (fois q p2);;
```

**Exercice 5 (Solution)**

```
let rec derivee (p:polynome) : polynome =
   match p with
    |[] -> []
    |[a, 0] -> []
    |(a, n)::q -> (a*n, n-1) :: (derivee q);;
```

**Exercice 6 (Solution)**

```
let rec tch n =
  match n with
  |0 -> [1, 0]
  |1 -> [1, 1]
  |n -> plus (fois [2, 1] (tch (n-1))) (fois [-1, 0] (tch (n
     -2)));;
```

**Exercice 7 (Solution)**

```
let rec division a b =
   match a, b with
   |_, [] -> failwith "Ceci ne devrait pas arriver"
   |[], _ -> [], []
   |(c, n)::aa, (d, m)::bb when n < m -> [], b
   |(c, n)::aa, (d, m)::bb ->
                  let a1 = plus a (fois [-c, n-m] b) in
                  let q1, r  = division a1 b in
                  ((c, n-m) :: q1), r;;
```

**Exercice 8 (Solution)**

```
let quotient a b = fst(division a b)

let rec cycloto n =
   let rec calcul k p : polynome =
      if k > n/2 then p
      else if n mod k = 0
           then calcul (k+1) (quotient p (cycloto k))
           else calcul (k+1) p in
   calcul 1 [1, n; -1, 0];;
```

# TP IV

# Mastermind

Dans ce TP nous allons programmer le jeu de mastermind. Nous verrons les possibilités (rudimentaires) de création d'une interface graphique de OCaml.

Le jeu de Mastermind est un jeu à deux joueurs. En voici le déroulement.

- Le premier joueur choisit secrètement une rangée de $p$ boules de couleur, avec un choix de $k$ couleurs, des couleurs peuvent être répétées
- Le second jour tente de deviner la configuration secrète en posant des questions au premier joueur sous la forme de rangées de $p$ boules colorées.
- Le premier joueur compare les deux rangées : sa réponse est la paire d'entiers ($N$,$B$) où $N$ (pour noir) est le nombre de boules de même couleur à la même place et $B$ (pour blanc) est le nombre de boules qui sont dans les deux configurations mais avec des places différentes.
- $N + B$ est donc le nombre de boules communes.
- Le second joueur fait ainsi plusieurs tentatives jusqu'à deviner la configuration du premier joueur.
- Le nombre de tentatives peut être limité, cela permet de déclarer un vainqueur.

**Exemple**
On suppose que $p$ vaut 4 et qu'il y a 6 couleurs représentées par un chiffre de 0 à 5.
Les couleurs sont représentées par les entiers de 0 à 5.
Voici la configuration à découvrir :

| | | | |
|---|---|---|---|
| 0 | 2 | 4 | 2 |

Voici un début de partie possible

| Question | | | | Réponse |
|---|---|---|---|---|
| 2 | 2 | 2 | 2 | (2,0) |
| 1 | 0 | 0 | 3 | (0,1) |
| 1 | 2 | 3 | 4 | (1,1) |

# 1 Ordinateur en premier joueur

Dans cette partie nous allons simuler le premier joueur.
Il s'agira de choisir une configuration et de calculer les entier $N$ et $B$.

**Exercice 1**
*Écrire une fonction* `secret n nC` *qui construit aléatoirement une* **liste** *de* `n` *valeurs choisies entre 0 et* `nC - 1`*. On pourra utiliser la fonction* `Random.int n` *qui choisit au hasard un entier compris dans* $[0; n[$.

```
secret : int -> int -> int list
```

**N.B.** Dans le cas de l'usage de GEDIT, le générateur aléatoire est ré-initialisé à la même valeur lors de chaque compilation. On peut imposer une initialisation aléatoire avec l'instruction

```
Random.self_init ();;
```

**Exercice 2**
*Écrire une fonction* `enPlace liste1 liste2` *qui compte le nombre de valeurs égales à la même place dans deux listes.*
*Il n'est pas nécessaire de supposer que les deux listes ont le même nombre d'éléments.*

```
enPlace : 'a list -> 'a list -> int
```

On veut maintenant compter le nombres d'éléments en commun à des places différentes.
Les questions qui suivent proposent une stratégie mais il existe sans doute d'autres possibilités.

**Exercice 3**
*Écrire une fonction* `retirer a liste` *qui reçoit un élément et une liste et renvoie un couple formé*

- *d'un booléen valant* `true` *si* `a` *est un élément de la liste et* `false` *sinon*
- *d'une liste avec les mêmes éléments que* `liste` *à l'exception de la première occurrence de* `a` *si* `a` *est un élément de la liste. L'ordre des éléments restants est conservé.*

```
retirer : 'a -> 'a list -> bool * 'a list
```

`retirer 3 [1; 4; 2]` renvoie `false, [1; 4; 2]`
`retirer 3 [1; 3; 2; 3]` renvoie `true, [1; 2; 3]`.

**Exercice 4**
*Écrire une fonction* `communs liste1 liste2` *qui compte le nombre d'éléments communs aux deux listes* `liste1` *et* `liste2`*.*

```
communs : 'a list -> 'a list -> int
```

`communs [1; 1; 5; 8] [1; 2; 3; 4]` renverra 1.
`communs [1; 1; 5; 8] [8; 2; 1; 8]` renverra 2.

**Exercice 5**
*Déterminer une fonction* `scoreNB liste1 liste2` *qui calcule le score de comparaison de deux listes comme décrit ci-dessus/*

```
scoreNB : 'a list -> 'a list -> int * int
```

`scoreNB [1; 1; 5; 8] [1; 2; 3; 4]` renverra `(1, 0)`.
`scoreNB [1; 1; 5; 8] [8; 2; 1; 8]` renverra `(1, 1)`.

# 2 Interface graphique

Un code source commenté est donné dans le dossier public.
Il reste à y inclure les fonctions ci-dessus pour pouvoir jouer des parties contre le programme.

# 3 Ordinateur en second joueur

Les outils développés ci-dessus permettent aussi de rechercher une solution.
La clé est de remarquer que les calculs de score sont symétriques. Si une liste `liste1` donne le score $(n, b)$ en le comparant avec la liste secrète alors celle-ci est une des listes qui donnent ce score en comparant avec `liste1`.
On va donc

- créer une liste de toutes les listes possibles
- de manière itérative on choisit une liste possible, on calcule son score et on élimine les listes qui n'ont pas le bon score
- jusqu'à ce que l'on trouve la liste.

**Exercice 6**
*Écrire une fonction* `configs n nC` *qui construit la liste des configurations (listes) de n éléments choisis entre 0 et* $nC - 1$. *(*`nC` *pour* **n***ombre de* **C***ouleurs).*

```
configs : int -> int -> int list list
```

**Exercice 7**
Écrire une fonction `solution config n nC` qui renvoie le nombre de tentatives faites pour trouver la solution (`config`) en utilisant la méthode ci-dessus.
La solution ne devra bien sur être utilisée que pour calculer le score.

```
resoudre : int list -> int -> int -> int
```

**Exercice 8**
Quelle est la moyenne du nombre de tentatives pour un jeu de 4 pions avec 6 couleurs possibles ?
Le calcul devient long.

```
moyenne :int -> int -> float
```

**Exercice 9**
Proposer le jeu graphiquement où l'ordinateur propose une suite de couleurs et vous donnez le score.

# 4 Solutions

**Exercice 1 (Solution)**

```
let rec secret n nC =
  if n = 0
  then []
  else (Random.int nC) :: (secret (n - 1) nC);;
```

**Exercice 2 (Solution)**

```
let rec enPlace liste1 liste2 =
  match liste1, liste2 with
  |t1::q1, t2::q2 when t1 = t2 -> 1 + (enPlace q1 q2)
  |t1::q1, t2::q2 -> enPlace q1 q2
  |_ -> 0;;
```

**Exercice 3 (Solution)**

```
let rec retirer a liste =
  match liste with
  |[] -> (false, [])
  |t::q when t = a -> (true, q)
  |t::q -> let (b, l) = retirer a q in (b, t::l);;
```

**Exercice 4 (Solution)**

```
let rec communs liste1 liste2 =
  match liste1 with
  |[] -> 0
  |t::q -> let b, l = retirer t liste2
           in communs q l + (if b then 1 else 0);;
```

**Exercice 5 (Solution)**

```
let scoreNB liste1 liste2 =
  let n = enPlace liste1 liste2 in
  let c = communs liste1 liste2 in
  n, (c - n);;
```

**Exercice 6 (Solution)**

```
let rec configs n nC =
  if n = 0
  then [[]]
  else let cfg1 = configs (n-1) nC in
       let cfg = ref [] in
       for i = 0 to (nC -1) do
         cfg := (List.map (fun l -> i::l) cfg1)@(!cfg) done;
        !cfg;;
```

**Exercice 7 (Solution)**

```
let resoudre liste n nC =
  let rec aux reste coups =
    match reste with
    |[] -> failwith "Je ne trouve pas de solution"
    |t::q -> let noir, blanc = scoreNB t liste in
             if noir = n
             then (coups + 1)
             else let filtre = List.filter (fun l -> (scoreNB
                l t) = (noir, blanc)) reste in
                  aux filtre (coups + 1)
  in aux (configs n nC) 0;;
```

**Exercice 8 (Solution)**

```
let moyenne n nC =
  let rec aux reste somme nombre =
    match reste with
    |[] -> (float somme)/.(float nombre)
    |t::q -> aux q (somme + (resoudre t n nC)) (nombre + 1) in
  aux (configs n nC) 0 0;;
```

**Exercice 9 (Solution)**

```
let cl = [|    red;   green;    blue;  yellow;
              cyan; magenta;   black;   white|]

let marge = 15;; (* Marges de séparation du bord *)
let pas = 50;;   (* Taille d'une cellule élémentaire *)
let rayon = 20;; (* Rayon des boules *)

let pos k = marge + pas/2 + pas*k;;
let ind x = (x - marge)/pas;;
```

```
let boule ligne col coul =
  set_color cl.(coul);  (* On choisit la couleur du tracé *)
  fill_circle (pos col) (pos ligne) rayon; (* tracé d'un
     disque *)
  set_color black;
  draw_circle (pos col) (pos ligne) rayon;; (* tracé d'un
     cercle noir au bord *)
```

```
let reponse ligne col noir blanc =
  set_color black;
  fill_circle (pos col) (pos ligne) (rayon/2);
  set_color white;
  moveto (pos col -2) (pos ligne -6);
  draw_string (string_of_int noir);
  fill_circle (pos (col + 1)) (pos ligne) (rayon/2);
  set_color black;
  draw_circle (pos (col +1)) (pos ligne) (rayon/2);
  moveto (pos (col + 1) -2) (pos ligne -6);
  draw_string (string_of_int blanc);;
```

```
let fin ligne col =
  set_color red;
  draw_circle (pos col) (pos ligne) rayon;
  moveto (pos col- 8) (pos ligne -4);(* on change la position
     *)
  set_color black;
  draw_string "Fin";; (* On écrit le texte, point bas gauche à
      la position *)
```

```
let tracer liste ligne =
  let rec aux reste k =
    match reste with
    |[]  -> ()
    |t::q -> set_color cl.(t);
             fill_circle (pos k) (pos ligne) rayon;
             aux q (k+1)
  in aux liste 0;;
```

```
let choix_reponse ligne n =
  reponse ligne n 0 0;
  fin ligne (n+2);
  let encore = ref true in
  let noir = ref 0 in
  let blanc = ref 0 in
    while !encore do
      let s = wait_next_event [Button_down] in
      let (i,j) = (ind s.mouse_x,ind s.mouse_y) in
      if j = ligne && i >= n && i <= n +2
      then if i = n
           then noir := (!noir + 1) mod (n+1)
           else if i = (n + 1)
                then blanc := (!blanc + 1) mod (n+1)
                else encore := false;
      reponse ligne n !noir !blanc done;
      !noir, !blanc;;
```

```
let jeu2 n nC =
  open_graph "";
  let pasTrouve = ref true in
  let ligne = ref 0 in
  let reste = ref (configs n nC) in
  while !pasTrouve do
    let t::q = !reste in
    tracer t !ligne;
    let noir, blanc = choix_reponse !ligne n in
    if noir < n
    then reste := List.filter (fun l -> (scoreNB l t) = (noir,
        blanc)) !reste
    else pasTrouve := false;
    ligne := !ligne + 1 done;
  moveto (pos 2) (pos !ligne);
  draw_string ("Solution en "^(string_of_int !ligne)^" coup(s)
     ");
  wait_next_event [Key_pressed];;
```

TP V

# Tri de tableaux

Dans ce TP nous allons appliquer le tri pivot à un tableau.
Nous avons écrit des algorithmes rapides pour les listes dans le cours mais on a besoin de copier les données dans des nouvelles listes à chaque étape. La nature modifiable (mutable) des valeurs d'un tableau permet d'espérer pouvoir effectuer le tri "en place".
Le grand avantage est alors qu'on ne crée pas de nouvelle structure en mémoire : la complexité spatiale est réduite.
Le tri fusion ne le permet pas mais cela est possible dans le cas du tri pivot.
On triera des tableaux de nombres, comparés avec < ou <=.

## 1 Généralités

Un outil constamment utilisé lorsque l'on modifie un tableau en gardant les valeurs est la fonction d'échange des valeurs entre deux positions du tableau.

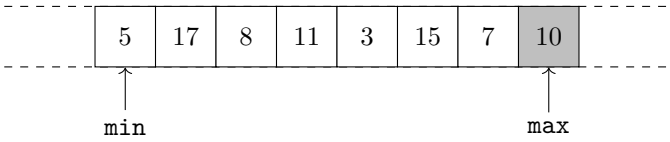
**Exercice 1**
*Écrire une fonction* `echange tab i j` *qui échange les valeurs des contenus du tableau* `tab` *aux indices $i$ et $j$. La fonction ne renvoie rien, elle modifie le tableau.*

Comme dans le cas du tri de listes, le principe du tri pivot est de trier des parties de l'ensemble des valeurs initiales. Mais cette partie sera ici représentée par une section du tableau, elle sera déterminée par les bornes supérieure et inférieure de cette section.

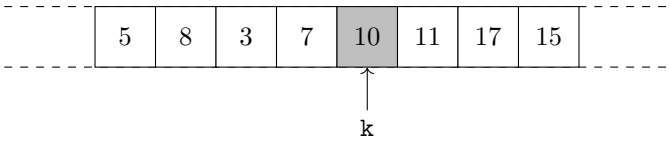
```
1  let tri_pivot tab =
2    let rec quicksort min max =
3      if min < max
4      then begin let k = separation tab min max in
5                 quicksort min (k-1);
6                 quicksort (k+1) max end in
7    quicksort 0 (Array.length tab -1);;
```

1. Lignes 1 et 7 : le programme principal appelle la fonction récursive pour tout le tableau
2. Ligne 2 : la fonction récursive
3. Ligne 3 : il faut au moins deux indices pour avoir quelque chose à trier
4. Ligne 4 : la fonction de séparation choisit un terme de la section, le **pivot**, et réordonne les termes de la section de telle manière que les valeurs plus petite que le pivot soient placées avant le pivot et les valeurs supérieures soient placées après. Le pivot est alors à sa place dans la section et c'est sa position qui renvoyée.
5. Lignes 5 et 6 : il suffit alors de trier les petites termes et les grands.

Voici un exemple de ce qui est demandé à la fonction de séparation.
On part de la situation suivante et on choisit le dernier terme comme pivot




on veut parvenir à la situation suivante.




L'ordre des 4 premiers éléments et des 3 derniers peut varier.

- Toute la difficulté de l'écriture est dans la fonction `separation`.
  Il existe deux méthodes principales que nous allons étudier dans les parties 2 et 4.
- Le choix du pivot a des conséquences lors du tri de tableaux presque ordonnés.
  La partie 3.2 expose le problème et propose des solutions.
- Dans un premier temps nous allons supposer que les valeurs du tableau sont **distinctes**.
  Dans la partie 5 nous verrons les conséquences des valeurs multiples.

# 2 Méthode de Lomuto

Nous allons commencer par une méthode de séparation qui n'est pas la méthode originale mais qui est plus simple à écrire.
La séparation entre `min` et `max` dans un tableau `tab` suit les étapes suivantes.

1. On choisit `tab.(max)` pour pivot.
2. On maintient une variable `k` initialisée à `min`.
   Cette variable est la position du premier élément supérieur au pivot
3. Pour $i$ variant de `min` à `max - 1` on veut maintenir l'invariant de boucle
   - les éléments d'indices `min` à `k-1` sont inférieurs au pivot,
   - les éléments d'indices `k` à `i-1` sont supérieurs au pivot.
4. Dans ce but
   - si `tab.(i)` est supérieur au pivot, on le laisse en place (on ne fait rien)
   - sinon on échange les termes aux positions `k` et `i` puis on incrémente `k` de 1.
5. Quand la boucle `for` est terminée, on place le pivot à sa place : on échange les termes aux positions `k` et `max`.
6. On renvoie la position du pivot : `k`.

**Exercice 2**
*Écrire une fonction* `separation tab min max` *qui effectue ces instructions.*

# 3 Tests

Nous allons tester le tri pivot sur des listes crées aléatoirement.
Pour créer une liste aléatoire on peut partir de la liste des $n$ premier entiers :

```
Array.init n (fun i -> i)
```

On la mélange en suite. On peut utiliser l'algorithme de Fisher-Yates (inversé) :
pour chaque $i$ de 0 à $n-1$ on choisit un entier $j$ au hasard entre $i$ et $n-1$ et on échange $i$ et $j$.
Pour simuler le hasard [1] on peut utiliser la fonction `Random.int p` qui renvoie un entier choisi au hasard entre 0 et $p-1$.

1. Ne pas oublier de réinitialiser le hasard avec `Random.self_init ()` avec GEDIT.

**Exercice 3**
*Écrire une fonction* `alea_perm n` *qui calcule un tableau aléatoire de taille* $n$.

## 3.1 Temps d'exécution

On peut alors tester le temps d'exécution. Pour cela on peut utiliser la fonction `Sys.time ()` qui renvoie le temps écoulé depuis le début de l'exécution.

**Exercice 4**
*Écrire une fonction* `test taille nombre` *qui calcule le temps moyen pris pour le tri de* `nombre` *tableaux aléatoires de* `taille` *éléments.*

**Exercice 5**
*Modifier* `test` *en une fonction* `coefficient taille nombre` *qui calcule le quotient du temps moyen par* $n\ln(n)$ *(*$n$ *est la valeur de taille). Que constate-t-on ?*

## 3.2 Le problème du tri pivot

Le tri pivot n'est efficace que lors du tri d'un tableau très "désordonnée".
Pour simuler de telles suites on partira du tableau trié à $n$ éléments en effectuant $n \div 1000$ échanges entres positions choisies au hasard.

**Exercice 6**
*Écrire une fonction* `alea_faible n` *qui calcule un tableau de taille* $n$ *faiblement désordonné contenant les entiers de 0 à* $n-1$.

**Exercice 7**
*Écrire une fonction* `test1 taille nombre` *qui calcule le temps moyen pris pour le tri de* `nombre` *tableaux aléatoires de* `taille` *éléments. Comparer avec les listes aléatoires.*

**Exercice 8**
*Vérifier que ce problème est résolu par l'un des moyens suivant :*

- *on choisit le pivot aléatoirement dans la section*
- *ou on commence par désordonner aléatoirement le tableau avant d'appeler la fonction* `quicksort`.

## 3.3 Nombre de comparaisons et d'échanges

On va ici modifier les fonctions pour permettre de compter les opérations élémentaires.
Bien entendu cela ralentira le tri, le but est de décompter les opérations.

**Exercice 9**
*Écrire une fonction* `separation_n tab min max` *qui effectue la séparation en renvoyant, en plus de l'indice du pivot, le nombre de comparaisons et le nombre d'échanges effectués.*

**Exercice 10**
*Écrire une fonction* `tri_pivot_n tab` *qui effectue le tri en revoyant le nombre de comparaisons et le nombre d'échanges effectués.*

**Exercice 11**
*Écrire une fonction* `test_n taille nombre` *qui renvoie le nombre moyen de comparaisons et d'échanges effectués pour le tri de* `nombre` *tableaux aléatoires de* `taille` *éléments.*

# 4 Méthode de Hoare

La méthode originale de Hoare (1961) pour la séparation des éléments est moins facile à écrire. Elle consiste à partir des deux extrémités de la section avec deux indices $i$ et $j$ que l'on fait se rejoindre en échangeant, en cours de route, les paires d'éléments `tab.(i)` supérieur au pivot et `tab.(j)` inférieur au pivot. L'invariant de boucle est ici que les termes d'indices entre `min` et $i-1$ sont inférieurs au pivot et les termes d'indices entre $j+1$ et `max` sont supérieurs au pivot.
On ne progresse que tant qu'on a $i \leqslant j$.
L'écriture de la séparation est difficile, dans un premier temps il vaut mieux tester `tab.(i) <= pivot` et `tab.(j) >= pivot` pour avancer. Cela ne change pas la séparation dans le cas de valeurs distinctes et évite des pièges quand il existe des valeurs égales (on ne peut pas avoir $i = j$ après avoir avancé les deux variables).

**Exercice 12**
*Écrire une fonction* `separationH tab min max` *qui partitionne selon cette méthode.*

**Exercice 13**
*Comparer les vitesses de tris d'une même liste (ou de plusieurs listes).*

**Exercice 14**
*En comparant le nombre de comparaisons et le nombre d'échanges, déterminer l'origine de l'amélioration.*

# 5 Valeurs multiples

Un autre problème du tri pivot est le cas des valeurs multiples.
En effet, le plus souvent, on aura besoin de trier selon des clés qui peuvent ne pas être uniques.
Le cas extrême est celui d'une portion de tableau dans laquelle toutes les valeurs comparées sont égales ; les découpages vus ci-dessus vont alors partager la portion en deux parties (en plus du pivot) dont l'une est vide et on aboutit au cas le pire, la complexité est quadratique.
Une solution possible est de faire l'échange même lorsqu'une valeur est égale au pivot dans la méthode de Hoare ; on commence par faire des échange inutiles mais la complexité totale sera diminuée. C'est un exemple où s'applique l'adage de D. Knuth

**Premature optimization is the root of all evil**

La fonction de séparation devient maintenant délicate à écrire [2] car il faut tenir compte des différents cas où on peut parvenir à l'égalité $i = j$.

**Exercice 15**
*Écrire une fonction de séparation.*
*Comparer les temps de tri pour une liste constante de taille 10000.*

2. This algorithm is easy to describe, and also easy to get wrong : J. Bentley

# 6 Solutions

**Exercice 1 (Solution)**

```
let echange tab i j =
  let temp = tab.(i) in
  tab.(i) <- tab.(j);
  tab.(j) <- temp;;
```

**Exercice 2 (Solution)**

```
let separation tab min max =
  let pivot = tab.(max) in
  let k = ref min in
  for i = min to (max -1) do
    if tab.(i) < pivot
      then begin echange tab i !k;
                 k := !k + 1 end done;
  echange tab !k max;
  !k;;
```

**Exercice 3 (Solution)**

```
let alea_perm n =
  let t = Array.init n (fun i -> i) in
  for i = 0 to (n-2) do
    let k = Random.int (n-i) in
    echange t i (i+k) done;
  t;;
```

**Exercice 4 (Solution)**

```
let test taille nombre =
  let temps = ref 0.0 in
  for i = 0 to (nombre -1) do
    let tab = alea_perm taille in
    let debut = Sys.time() in
    tri_pivot tab;
    let fin = Sys.time() in
    temps := !temps +. (fin -. debut) done;
  !temps /. (float_of_int nombre);
```

**Exercice 5 (Solution)**

```
let coefficient taille nombre =
  let temps = ref 0.0 in
  for i = 0 to (nombre -1) do
    let tab = alea_perm taille in
    let debut = Sys.time() in
    tri_pivot tab;
    let fin = Sys.time() in
    temps := !temps +. (fin -. debut) done;
  let nn = float_of_int taille in
  !temps /. (float_of_int nombre) /. nn /. (log nn);;
```

**Exercice 6 (Solution)**

```
let alea_faible n =
  let t = Array.init n (fun i -> i) in
  for i = 0 to n/1000 do
    let k1 = Random.int n in
    let k2 = Random.int n in
    echange t k1 k2 done;
  t;;
```

**Exercice 7 (Solution)**

```
let test1 taille nombre =
  let temps = ref 0.0 in
  for i = 0 to (nombre -1) do
    let tab = alea_faible taille in
    let debut = Sys.time() in
    tri_pivot tab;
    let fin = Sys.time() in
    temps := !temps +. (fin -. debut) done;
  !temps /. (float_of_int nombre);;
```

**Exercice 8 (Solution)**

```
let separation tab min max =
  let c = min + Random.int(max - min  +1) in
  echange tab c max;
  let pivot = tab.(max) in
  let k = ref min in
  for i = min to (max -1) do
    if tab.(i) < pivot
      then begin echange tab i !k;
                 k := !k + 1 end done;
  echange tab !k max;
  !k;;
```

ou

```
let tri_pivot tab =
  let n = Array.length tab in
  for i = 0 to (n-2) do
    let k = Random.int (n-i) in
    echange tab i (i+k) done;
  let rec quicksort min max =
    if min < max
    then begin let k = separation tab min max in
               quicksort min (k-1);
               quicksort (k+1) max end in
  quicksort 0 (n -1);;
```

**Exercice 9 (Solution)**

```
let separation_n tab min max =
  let pivot = tab.(max) in
  let cp = ref 0 in
  let ech = ref 0 in
  let k = ref min in
  for i = min to (max -1) do
    cp := !cp + 1;
    if tab.(i) < pivot
    then begin echange tab i !k;
               ech := !ech + 1;
               k := !k + 1 end done;
  echange tab !k max;
  !k, !cp, !ech;;
```

**Exercice 10 (Solution)**

```
let tri_pivot_n tab =
  let rec quicksort min max =
    if min < max
    then begin let k, cp, ech = separation_n tab min max in
               let cp1, ech1 = quicksort min (k-1) in
               let cp2, ech2 = quicksort (k+1) max in
               (cp + cp1 + cp2), (ech + ech1 + ech2) end
         else 0, 0 in
  quicksort 0 (Array.length tab -1);;
```

**Exercice 11 (Solution)**

```
let test_n taille nombre =
  let somme_cp = ref 0 in
  let somme_ech = ref 0 in
  for i = 0 to (nombre -1) do
    let tab = alea_perm taille in
    let cp, ech = tri_pivot tab in
    somme_cp := !somme_cp + cp;
    somme_ech := !somme_ech + ech done;
  !somme_cp/nombre, !somme_ech/nombre;;
```

**Exercice 12 (Solution)**

```
let separationH tab min max =
  let pivot = tab.(max) in
  let i = ref min in
  let j = ref (max - 1) in
  while !i <= !j do
    while !i <= !j && tab.(!i) <= pivot do i := !i + 1 done;
    while !i <= !j && tab.(!j) >= pivot do j := !j - 1 done;
    if !i < !j
    then begin echange tab !i !j;
               i := !i + 1;
               j := !j - 1 end done;
  echange tab max !i;
  !i;;
```

La condition `!i <= !j` permet d'éviter de continuer d'avancer quand plusieurs valeurs sont égales au pivot.

```
let tri_pivotH tab =
  let rec quicksort min max =
    if min < max
    then begin let k = separationH tab min max in
               quicksort min (k-1);
               quicksort (k+1) max end in
  quicksort 0 (Array.length tab -1);;
```

**Exercice 13 (Solution)**

```
let test2 taille nombre =
  let temps1 = ref 0.0 in
  let temps2 = ref 0.0 in
  for i = 0 to (nombre -1) do
    let tab1 = alea_perm taille in
    let tab2 = Array.init taille (fun i -> tab1.(i)) in
    let debut = Sys.time() in
    tri_pivot tab1;
    let fin = Sys.time() in
    temps1 := !temps1 +. (fin -. debut);
    let debut = Sys.time() in
    tri_pivotH tab2;
    let fin = Sys.time() in
    temps2 := !temps2 +. (fin -. debut) done;
  !temps1 /. (float_of_int nombre), !temps2 /. (float_of_int
     nombre);;
```

**Exercice 14 (Solution)**

```
let separationH_n tab min max =
  let pivot = tab.(max) in
  let i = ref min in
  let j = ref (max - 1) in
  let cp = ref 0 in
  let ech = ref 0 in
  while !i <= !j do
    while !i <= !j && tab.(!i) <= pivot do i := !i + 1; incr
       cp done;
    while !i <= !j && tab.(!j) >= pivot do j := !j - 1; incr
       cp done;
    cp := !cp + 2;
    if !i < !j
    then begin echange tab !i !j;
               i := !i + 1;
               j := !j - 1;
               incr ech end done;
  echange tab max !i;
  incr ech;
  !i, !cp, !ech;;
```

```
let tri_pivotH_n tab =
  let rec quicksort min max =
    if min < max
    then begin let k, cp, ech = separationH_n tab min max in
               let cp1, ech1 = quicksort min (k-1) in
               let cp2, ech2 = quicksort (k+1) max in
               (cp + cp1 + cp2), (ech + ech1 + ech2) end
    else 0,0 in
  quicksort 0 (Array.length tab -1);;
```

```
let test2_n taille nombre =
  let cp1 = ref 0 in
  let ech1 = ref 0 in
  let cp2 = ref 0 in
  let ech2 = ref 0 in
  for i = 0 to (nombre -1) do
    let tab1 = alea_perm taille in
    let tab2 = Array.init taille (fun i -> tab1.(i)) in
    let cp, ech = tri_pivot_n tab1 in
    cp1 := !cp1 + cp;
    ech1 := !ech1 + ech;
    let cp, ech = tri_pivotH_n tab2 in
    cp2 := !cp2 + cp;
    ech2 := !ech2 + ech done;
  print_string "Comparaisons - Lomuto : ";
  print_int (!cp1/nombre);
  print_string ", Hoare : ";
  print_int (!cp2/nombre);
  print_newline();
  print_string "Echanges - Lomuto : ";
  print_int (!ech1/nombre);
  print_string ", Hoare : ";
  print_int (!ech2/nombre);
  print_newline();;
```

**Exercice 15 (Solution)** Si on parvient à $i = j$ avant l'échange, c'est que l'on a une valeur égale au pivot en $i$.

```
let separationHs tab min max =
  let pivot = tab.(max) in
  let i = ref min in
  let j = ref (max - 1) in
  while !i <= !j do
    while !i <= !j && tab.(!i) < pivot do incr i done;
    while !i <= !j && tab.(!j) > pivot do decr j done;
    if i = j then decr j;
    if !i < !j
    then begin echange tab !i !j;
               incr i;
               decr j end done;
  echange tab max !i;
  !i;;
```

La formulation de Sedgewick ne fait pas ce test supplémentaire.

```
let separationHs tab min max =
  let pivot = tab.(max) in
  let i = ref (min - 1) in
  let j = ref max in
  while !i < !j do
    incr i;
    while tab.(!i) < pivot do incr i done;
    decr j;
    while !i <= !j && tab.(!j) > pivot do decr j done;
    if !i < !j then echange tab !i !j done;
  echange tab max !i;
  !i;;
```

Pour trier 10000 éléments égaux :

```
Temps - Lomuto : 3.565094, Hoare <= : 4.146845, Hoare < :
   0.027982
```

# TP VI

# Formules arithmétiques

Dans ce TP nous allons interpréter les programmes dans un langage très simple : les expressions arithmétiques. Le but est de lire une expression arithmétique sous forme d'une chaîne de caractères et d'en calculer la valeur.
Nous nous restreindrons au cas de valeurs entières positives (le cas des entiers négatifs est compliqué par le double sens du symbole "-") et des opérations "+", "-", "*" et "/".
**Rappels**

- Une chaîne de caractères est encadrée par des guillemets doubles : `"Bonjour"`.
- Sa longueur est fournie par `String.length ch`.
- Le caractère d'indice $i$ d'une chaîne est obtenu par `ch.[i]`.
- Un caractère est encadré par des guillemets simples.
- `Char.code` permet de renvoyer le code ascii d'un caractère.
- pour calculer le chiffre représenté par un caractère entre `'0'` et `'9'`, on peut écrire `Char.code c - Char.code '0'`. `'5'` donnera le chiffre 5.
- `List.rev` permet de retourner une liste.

## 1 Lexer

La première étape consiste à lire la chaîne de caractères et à la séparer en ses objets de bases appelés lexèmes : ce sera les entiers, les opérateurs et les parenthèses.
On utilisera le type suivant :

```
type lexeme =  Nb of int (* les entiers *)
             |Plus |Moins |Fois |Divise (* les opérateurs *)
             |ParO |ParF;; (* les parenthèses ouvrante et
                fermante *)
```

**Exercice 1**
*Écrire une fonction* `lecture : string -> lexeme list` *qui reçoit une chaîne de caractères et qui renvoie la liste des lexèmes associée. Les caractères qui ne correspondent pas à un lexème seront ignorés.* `lecture "5*(41 + 3) a"` *doit renvoyer*

```
[Nb 5; Fois; ParO; Nb 41; Plus; Nb 3; ParF]
```

## 2 Notation polonaise inversée

En 1920 le mathématicien polonais Jan Lukasiewicz propose la notation pré-fixé (ou polonaise) qui consiste à considérer les opérations comme des opérateurs à 2 variables, $7 + 11$ s'écrit $+\ 7\ 11$. L'avantage est que les parenthèses deviennent inutiles :+++
$(7 - 2).\sin(2x + \pi/3)$ devient $*\ -\ 7\ 2\ \sin\ +\ *\ 2\ x\ /\ \pi\ 3$.++ La notation polonaise inverse ou notation post-fixée a été proposée par le philosophe et inf
Elle a été diffusée dans le public comme interface utilisateur avec les calculatrices de bureau de Hewlett-Packard (HP-9100), puis avec la calculatrice scientifique HP-35 en 1972.
Elle consiste simplement à placer l'opérateur **après** les deux opérandes, `"7 + 11"` s'écrit `"7 11 +"`.
L'expression `"32 - 2*((7 / 2)*3 - 12)"` peut s'écrire `"32 2 7 2 / 3 * 12 - * -"`
L'avantage est que, combinée à une pile, cette notation permet d'effectuer les calculs sans faire référence à une quelconque adresse mémoire.

- On lit l'expression terme-à-terme :
- si on lit une valeur numérique, elle est empilée,
- si on lit une opération ♣,
  on dépile les deux derniers opérandes $a$ et $b$
  et on empile le résultat du calcul $a$ ♣ $b$.

**Un exemple** On part de l'expression `"32 2 7 2 / 3 * 12 - * -"` :

| Lecture | Action | Pile | | | |
|---|---|---|---|---|---|
| | On crée une pile vide | | | | |
| 32 | On empile | 32 | | | |
| 2 | On empile | 32 | 2 | | |
| 7 | On empile | 32 | 2 | 7 | |
| 2 | On empile | 32 | 2 | 7 | 2 |
| / | On dépile 2 éléments | 3 | 2 | | |
| | On empile 7/2 | 32 | 2 | 3 | |
| 3 | On empile | 32 | 2 | 3 | 3 |
| $*$ | On dépile 2 éléments | 32 | 2 | | |
| | On empile $3 * 3$ | 32 | 2 | 9 | |
| 12 | On empile | 32 | 2 | 9 | 12 |
| - | On dépile 2 éléments | 32 | 2 | | |
| | On empile $9 - 12$ | 32 | 2 | | -3 |
| $*$ | On dépile 2 éléments | 32 | | | |
| | On empile $2 * (-3)$ | 32 | -6 | | |
| $-$ | On dépile 2 éléments | | | | |
| | On empile $32 - (-6)$ | 38 | | | |
| | On dépile le résultat : 38 | | | | |

**Exercice 2**
*Définir un type* `pile` *pour une pile* **impérative** *et écrire les fonctions* `créerPile`, `estrPileVide`, `empiler`, `voir` *et* `depiler`. `depiler` *devra enlever l'élément au sommet de la pile* **et** *le renvoyer en même temps.*

**Exercice 3**
*Écrire une fonction* `calculNPI : string -> int` *qui calcule la valeur d'une expression* **écrite au format NPI** *dans une liste de lexèmes et qui calcule sa valeur.*

# 3 Cas général

Le procédé ci-dessus demande d'écrire la formule mathématique en notation N.P.I. ce qui n'est pas pratique. Nous allons maintenant traduire une formule "normale" en une formule N.P.I.
Pour cela on aura à gérer la priorité des opérations :
`"3 + 4 * 5"` est traduit en `"3 4 5 * +"` alors que `"3 * 4 + 5"` est traduit en `"3 4 * 5 +"`
On donne la priorité 1 aux opérations + et − et la priorité 2 aux opérations ∗ et /.

Pour traduire une formule on effectue les opérations suivantes.

- On lit la formule terme-à-terme :
- on crée une pile vide :
- si on lit une valeur numérique, elle est ajoutée à la sortie,
- si on lit une opération ♣, on dépile les opérations de priorité supérieure ou égale à celle de ♣ et on les ajoute à la sortie, on empile ensuite ♣,
- si on lit une parenthèse ouvrante on l'empile,
- si on lit une parenthèse fermante, on dépile et on ajoute à la sortie toutes les opérations jusqu'à ce qu'on trouve une parenthèse ouvrante, on dépile cette parenthèse ouvrante.
- Quand on a lu tous les lexèmes, on dépile et on ajoute à la sortie toutes les opérations resytantes.
- La sortie est une liste que l'on doit renverser pour obtenir la formule N.P.I.

**Un exemple** On part de l'expression `"2*(11-2+3*2)"` qui a été transcrite en

```
[Nb 2; Fois; ParO; Nb 11; Moins; Nb 2; Plus;
                                 Nb 3; Fois; Nb 2; ParF]
```

Pour des raisons de place on écrit les lexèmes sous forme "naturelle" dans la pile et la sortie.

| Lecture | Pile | Sortie |
|---|---|---|
| Nb 2 | | [2] |
| Fois | (*, 2) | [2] |
| ParO | (*, 2) ((, 0) | [2] |
| Nb 11 | (*, 2) ((, 0) | [11; 2] |
| Moins | (*, 2) ((, 0) (-, 1) | [11; 2] |
| Nb 2 | (*, 2) ((, 0) (-, 1) | [2; 11; 2] |
| Plus | (*, 2) ((, 0) (+, 1) | [-; 2; 11; 2] |
| Nb 3 | (*, 2) ((, 0) (+, 1) | [3; -; 2; 11; 2] |
| Fois | (*, 2) ((, 0) (+, 1) (*, 2) | [3; -; 2; 11; 2] |
| Nb 2 | (*, 2) ((, 0) (+, 1) (*, 2) | [2; 3; -; 2; 11; 2] |
| ParF | (*, 2) | [+; *; 2; 3; -; 2; 11; 2] |
| | | [*; +; *; 2; 3; -; 2; 11; 2] |

La notation N.P.I. est donc

```
[Nb 2; Nb 11; Nb 2; Moins; Nb 3; Nb 2; Fois; Plus; Fois]
```

**Exercice 4**
*Écrire une fonction* `convertirNPI : lexeme list -> lexeme list` *qui convertit une liste de lexèmes qui exprime une expression sous forme naturelle en une liste de lexèmes qui exprime une expression sous forme N.P.I.*

**Exercice 5**
*En déduire une fonction* `calcul : string -> int` *qui calcule la valeur d'une expression naturelle dans une chaîne de caractères et qui calcule sa valeur.*

# 4 Solutions

**Exercice 1 (Solution)**

```
let c0 = Char.code '0';;
let add k liste =
    if k = 0
    then liste
    else (Nb k) :: liste;;

let lecture texte =
   let n = String.length texte in
   let rec lire i k fait =
      if i = n
      then add k fait
      else match texte.[i] with
           |c when Char.code c >= c0 && Char.code c <= c0 + 9
                -> lire (i+1) (10*k + Char.code c - c0) fait
           |'+' -> lire (i+1) 0 (Plus   :: (add k fait))
           |'-' -> lire (i+1) 0 (Moins  :: (add k fait))
           |'*' -> lire (i+1) 0 (Fois   :: (add k fait))
           |'/' -> lire (i+1) 0 (Divise :: (add k fait))
           |'(' -> lire (i+1) 0 (ParO   :: (add k fait))
           |')' -> lire (i+1) 0 (ParF   :: (add k fait))
           |_   -> lire (i+1) 0 (add k fait) in
   List.rev (lire 0 0 []);;
```

**Exercice 2 (Solution)**

```
type 'a pile = {contenu = 'a list};;

let creerPile () = {contenu = []};;

let estPileVide pile =
   pile.contenu = [];;

let empiler x pile =
   pile.contenu  <- x :: pile.contenu;;

let depiler pile =
   match pile.contenu with
   |[] -> failwith "La pile est vide"
   |t::q -> pile.contenu <- q;
            t;;

let voir pile = List.hd pile.contenu;;
```

**Exercice 3 (Solution)**

```
let depiler2 pile =
    let b = depiler pile in
    let a = depiler pile in
    a, b;;

let calculNPI liste =
   let pile = creerPile () in
   let rec traiter liste =
      match liste with
      |[] -> depiler pile
      |(Nb k)::q -> empiler k pile;
                    traiter q
      |Plus::q -> let a, b = depiler2 pile in
                  empiler (a+b) pile;
                  traiter q
      |Moins::q -> let a, b = depiler2 pile in
                   empiler (a-b) pile;
                   traiter q
      |Fois::q -> let a, b = depiler2 pile in
                  empiler (a*b) pile;
                  traiter q
      |Divise::q -> let a, b = depiler2 pile in
                    empiler (a/b) pile;
                    traiter q
       |_::q -> traiter q  in
   traiter liste;;
```

**Exercice 4 (Solution)**

```
let sortir prio pile out =
    while not (estPileVide pile) && snd (voir pile) >= prio do
       empiler (fst (depiler pile)) out done;;

let vider pile =
   let rec transfert pile out =
      if estPileVide pile
      then out
      else let x = depiler pile in transfert pile (x::out)
   in transfert pile [];;
```

```
let convertirNPI liste =
   let ope = creerPile () in
   let out = creerPile () in
   let rec traiter liste =
      match liste with
      |(Nb k)::reste -> empiler (Nb k) out;
                        traiter reste
      |Plus::reste -> sortir 1 ope out;
                      empiler (Plus, 1) ope;
                      traiter reste
      |Moins::reste -> sortir 1 ope out;
                       empiler (Moins, 1) ope;
                       traiter reste
      |Fois::reste -> sortir 2 ope out;
                      empiler (Fois, 2) ope;
                      traiter reste
      |Divise::reste -> sortir 2 ope out;
                        empiler (Divise, 1) ope;
                        traiter reste
      |ParO::reste -> empiler (ParO, 0) ope;
                      traiter reste
      |ParF::reste -> sortir 1 ope out;
                      let _ = depiler ope in traiter reste
      |[] -> sortir 1 ope out;
             vider out
   in traiter liste;;
```

**Exercice 5 (Solution)**

```
let calcul ch =
   calculNPI (convertirNPI (lecture ch));;
```

# TP VII

# Le problème de partition

Après un casse réussi, deux cambrioleurs veulent partager équitablement leur butin. Ils commencent par s'accorder sur la valeur de chacun des n objets volés ; ceci leur donne un ensemble $X = \{x_0, \ldots, x_{n-1}\}$ de nombres entiers. Ils leur faut alors résoudre le problème suivant : déterminer une partition de $X$ en deux sous-ensembles $Y$ et $Z$ de même somme. Nos deux malfaiteurs sont confrontés au **problème de partition** : nous dirons que $X$ est une **instance** de ce problème.
Le **problème de décision** associé s'énonce ainsi : une telle partition existe-t-elle ?
On peut généraliser le problème décision en problème de somme partielle (SSP, subset sum problem en anglais) : étant donnée une suite $X = (x_0, x_1, \ldots, x_{n-1})$ d'entiers positifs et un entier $k$ existe-t-il un sous ensemble $I \subset \{0, 1, \ldots, n-1\}$ tel que $\sum_{i \in I} x_i = k$ ?

Si on $\|X\|$ la somme des éléments de $X$ : $\|X\| = \sum_{i=0}^{n-1} x_i$ le problème de décision de partition est le problème de somme partielle pour $k = \frac{\|X\|}{2}$, avec $\|X\|$ pair.

Dans ce TP, la suite $X$ sera représentée par une liste.
Pour résoudre le problème (de décision) SSP on cherchera une fonction

```
f : int list -> int -> bool
```

telle que `f x k` renvoie `true` s'il existe une partie de $X$, représenté par `x`, dont la somme est $k$ et qui renvoie `false` sinon.
On demandera aussi une fonction

```
f_sol : int list -> int -> int list
```

telle que `f_sol x k` renvoie une liste `x1` extraite de `x` dont la somme est $k$ et qui renvoie `[-1]` sinon. On dira que `f_sol` choisit une solution.

# 1 Passage en force

Nous allons commencer par traduire l'énoncé directement. On doit donc
- déterminer toutes les parties $I \subset \{0, 1, \ldots, n-1\}$,
- calculer $\sum_{i \in I} x_i$ tant que cette somme est distincte de $k$

Une idée (itérative) est de remarquer qu'on peut caractériser une partie de $\{0, 1, \ldots, n-1\}$ par la suite des valeurs (0 ou 1) de sa fonction caractéristique et qu'une telle suite de 0 et 1 peut être vue comme la suite issue de la décomposition en base 2 d'un entier entre 0 et $2^n - 1$.
On peut ainsi écrire la somme partielle associée à un entier $p$ :

```
let rec somme_partielle liste p =
   match liste with
   |[] -> 0
   |t::q -> t*(p mod 2) + somme_partielle q (p/2);;
```

**Exercice 1**
*Que donne* `somme_partielle [4; 2; 1; 2; 3] 25` *?*
*Déterminer $p$ pour que* `somme_partielle [a; b; c; d; e] p` *renvoie $a + b + d$.*

**Exercice 2**
*Écrire une fonction* `puissance2 : int list -> int` *qui renvoie $2^n$ où $n$ est la longueur de la liste* **sans** *utiliser* `List.length` *et en moins de 5 lignes.*

**Exercice 3**
*Écrire une fonction* `brute_force` *qui utilise la méthode exposée pour résoudre SSP.*

**Exercice 4**
*Calculer le nombre d'opérations élémentaires effectuées dans le pire des cas (quand la réponse est* `false` *en fonction de la taille $n$ de la liste.*

**Exercice 5**
*Modifier la fonction* `brute_force` *en une fonction* `brute_force_sol` *qui choisit une solution.*

# 2 Introduction de la récursivité

Une autre méthode pour déterminer les listes extraites d'une liste est de remarquer que les listes extraites de `t::q` sont

- soit des listes extraites de $q$
- soit des listes de la forme `t::q1` avec `q1` extraite de $q$.

Ici on traduit le problème par $k$ est une somme extraite de `t::q` si et seulement si

- $k$ est une somme extraite de `q` ou
- $k - t$ est une somme extraite de `q`.

**Exercice 6**
*Écrire une fonction* `recursivite` *qui utilise la méthode exposée pour résoudre SSP.*

**Exercice 7**
*Calculer le nombre d'opérations élémentaires effectuées dans le pire des cas (quand la réponse est* `false` *en fonction de la taille $n$ de la liste.*

**Exercice 8**
*Comme les termes sont positifs on peut écrire un algorithme dont espère qu'il sera plus rapide en utilisant le fait que la réponse est immédiate quand $k = 0$ ou $k < 0$.*
*Modifier* `recursivite` *dans ce sens.*

**Exercice 9**
*Modifier la fonction* `recursivite` *en une fonction* `recursivite_sol` *qui choisit une solution.*

# 3 Programmation dynamique

La formule ; $k$ est une somme extraite de `t::q` si et seulement si
- $k$ est une somme extraite de `q` ou
- $k - t$ est une somme extraite de `q`.

permet d'exprimer le résultat pour une liste de taille $n$ et un entier $k$ à partir de deux résultats pour le reste de la liste et des entiers inférieurs à $k$ : des sous-problèmes. Les conditions d'un algorithme utilisant la programmation dynamique sont réunies.
Si la suite des éléments est $X = (x_0, x_1, \ldots, x_{n-1})$ et que l'objectif est $k$, on définit une matrice de booléens `m` de taille $(n+1) \times (k+1)$ telle que `m.(i).(j)` vaut `true` si et seulement si $j$ est une somme partielle de $X_i = (x_0, x_1, \ldots, x_{i-1})$.

**Exercice 10**
*Écrire une fonction* `dynamique` *qui utilise la méthode de programmation dynamique pour résoudre SSP. On pourra utiliser la fonction* `Array.of_list` *pour convertir une liste en tableau mais cela n'est pas obligatoire.*

**Exercice 11**
*Déterminer la complexité de votre fonction en fonction de $n$ et de $k$.*
*Donner un exemple pour lequel $k \leqslant \|X\|$ et la complexité n'est pas polynomiale en $n$.*

**Exercice 12**
*La complexité spatiale, $nk$, peut être importante. Écrire une une fonction* `dynamique1` *qui utilise uniquement un tableau de taille $k+1$ pour résoudre SSP.*

**Exercice 13**
*Écrire une fonction* `dynamique_sol` *qui choisit une solution.*

# 4 Solutions

**Exercice 1 (Solution)** On obtient $1.4 + 0.2 + 0.1 + 1.2 + 1.3 = 9$.
On doit prendre $p = 1.2^0 + 1.2^1 + 0.2^2 + 1.2^3 + 0.2^4 = 11$.

**Exercice 2 (Solution)**

```
let rec puissance2 liste =
   match liste with
   |[] -> 1
   |t::q -> 2*(puissance2 q);;
```

**Exercice 3 (Solution)**

```
let brute_force liste k =
   let reponse = ref false in
   let i = ref 0 in
   let max = puissance2 liste in
   while not !reponse && !i < max do
      if somme_partielle liste !i = k
      then reponse := true;
      incr i done;
   !reponse;;
```

**Exercice 4 (Solution)** `puissance2` effectue $n$ multiplications.
`somme_partielle` effectue 4 opérations pour chaque élément de la liste donc $4n$ en tout.
La complexité est donc $n + 4n.2^n = \mathcal{O}\big(n.2^n\big)$.

**Exercice 5 (Solution)** On commence par écrire l'extraction d'une liste à l'aide d'un entier qui va représenter la fonction indicatrice.

```
let rec extraire liste p =
   match liste with
   |[] -> []
   |t::q when p mod 2 = 0 -> extraire q (p/2)
   |t::q -> t :: (extraire q (p/2));;
```

```
let brute_force_sol liste k =
   let sol = ref (-1) in
   let i = ref 0 in
   let max = puissance2 liste in
   while !sol = -1 && !i < max do
      if somme_partielle liste !i = k
      then begin sol := !i;
      incr i done;
   if !sol = -1
   then [-1]
   else extraire liste !sol;;
```

**Exercice 6 (Solution)**

```
let rec recursivite liste k =
   match liste with
   |[] -> k = 0
   |t::q -> recursivite q k || recursivite q (k-t);;
```

**Exercice 7 (Solution)** On note $C(n)$ la complexité maximale pour une liste de taille $n$.
$C(0) = 1$ (la comparaison)
$C(n) = 2 + 2C(n-1)$ car on calcule un "ou" et une différence.
En posant $u_n = C(n) + 2$ on a $u_0 = 3$ et $u_n = 2u_{n-1}$ donc $u_n = 3.2^n$ puis $C(n) = 3.2^n - 2$.

**Exercice 8 (Solution)**

```
let rec recursivite liste k =
   match liste, k with
   |_, 0 -> true
   |[], _ -> false
   |t::q, k when k < t -> recursivite q k
   |t::q, _ -> recursivite q k || recursivite q (k-t);;
```

**Exercice 9 (Solution)**

```
let rec recursivite_sol liste k =
   match liste, k with
   |_, 0 -> []
   |[], _ -> [-1]
   |t::q, k when k < t -> recursivite_sol q k
   |t::q, _ -> begin match recursivite_sol q (k-t) with
                     |[-1] ->  recursivite_sol q k
                     |l -> t::l end;;
```

**Exercice 10 (Solution)** La formule sur les sous-problèmes n'est valide que pour $t \leqslant k$.
Si on convertit en tableau.

```
let dynamique liste k =
   let tab = Array.of_list liste in
   let n = Array.length tab in
   let m = Array.make_matrix (n+1) (k+1) false in
   m.(0).(0) <- true;
   for i = 1 to n do
      for j = 0 to k do m.(i).(j) <- m.(i-1).(j) done;
      let t = tab.(i-1) in
      for j = t to k do m.(i).(j) <- m.(i).(j) || m.(i-1).(j-t
         ) done done;
   m.(n).(k);;
```

Sans conversion.

```
let dynamique liste k =
   let n = List.length liste in
   let m = Array.make_matrix (n+1) (k+1) false in
   m.(0).(0) <- true;
   let rec aux reste i =
      match reste with
      |[] -> m.(n).(k)
      |t::q -> for j = 0 to k do m.(i).(j) <- m.(i-1).(j) done
         ;
               for j = t to k do m.(i).(j) <- m.(i).(j) || m.(
                  i-1).(j-t) done;
               aux q (i+1) in
   aux liste 1;;
```

**Exercice 11 (Solution)** La complexité est un $\mathcal{O}(nk)$.
Si on a $x_i = 3^i$ on a $\|X\| = \frac{1}{2}(3^n - 1)$ donc, pour $k = \frac{3}{4}.3^n$, la complexité sera de l'ordre de $n.3^n$.

**Exercice 12 (Solution)**

```
let dynamique1 liste k =
   let tab = Array.of_list liste in
   let n = Array.length tab in
   let m = Array.make (k+1) false in
   m.(0) <- true;
   for i = 1 to n do
      let t = tab.(i-1) in
      for j =  k downto t do m.(j) <- m.(j) || m.(j-t) done
         done;
   m.(k);;
```

**Exercice 13 (Solution)** On va utiliser un tableau de solutions.

```
let dynamique_sol liste k =
   let tab = Array.of_list liste in
   let n = Array.length tab in
   let sol = Array.make (k+1) [-1] in
   m.(0) <- [];
   for i = 1 to n do
      let t = tab.(i-1) in
      for j =  k downto t do
         if sol.(j-t) <> [-1]
         then sol.(j) <- t::sol.(j-t) done done;
   sol.(k);;
```