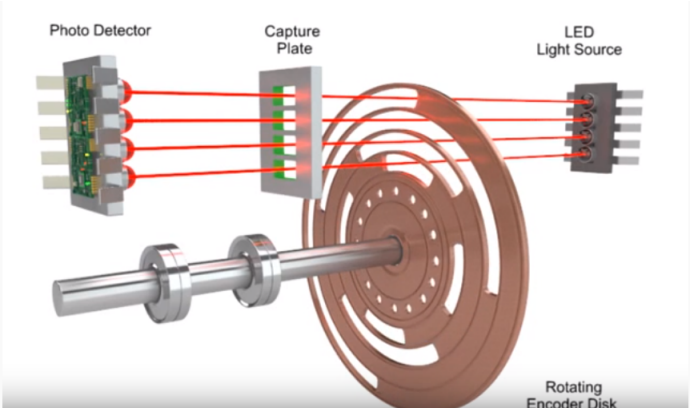
Travaux pratiques 1

Codes de Gray

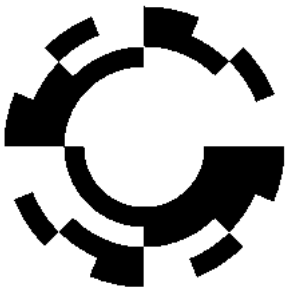
Option informatique MPSI1 & MPSI2

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.

I 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]`.

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

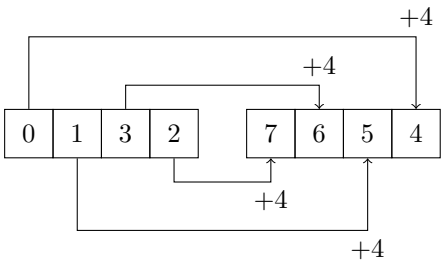
Question 2
Écrire la fonction réciproque `entier : int list-> int` qui convertit une liste de 0 et 1 en un entier. La fonction doit être récursive et ne calculera pas les puissance de 2.

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.

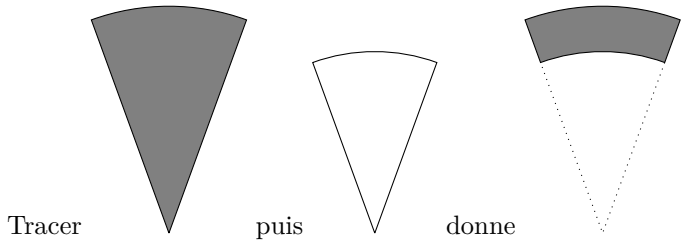


Question 3
Prouver que la suite $G_p = (g_0, g_1, \ldots, g_{2^p-1})$ verifie que chaque entier de 0 à $2^p - 1$ est atteint une fois et que deux entiers consécutifs (ainsi que le premier et le dernier) ont des décompositions binaires qui ne diffèrent qu'en un seul bit.

Question 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)`.

II 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.




```
#load "graphics.cma";;
(* ou, selon la version, #use "topfind";;
#require "graphics";;*)
open Graphics;;

let x0 = 300;;
let y0 = 300;;
let d = 10;;
let r0 = 50;;

let secteur ns k i =
  let r = r0 + i*d in
  set_color black;
  fill_arc x0 y0 (r + d) (r + d) (360*k/ns) (360*(k+1)/ns);
  set_color white;
  fill_arc x0 y0 r r (360*k/ns) (360*(k+1)/ns);;

open_graph " 600x600";;
(* Dessiner des trucs*)
let _ = wait_next_event [Button_down] in close_graph ();;
```

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

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

let x0 y0 d r0 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 $r_0 + (i+1).d$.

let secteur 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.

set_color définit la couleur : noir ou blanc ici.

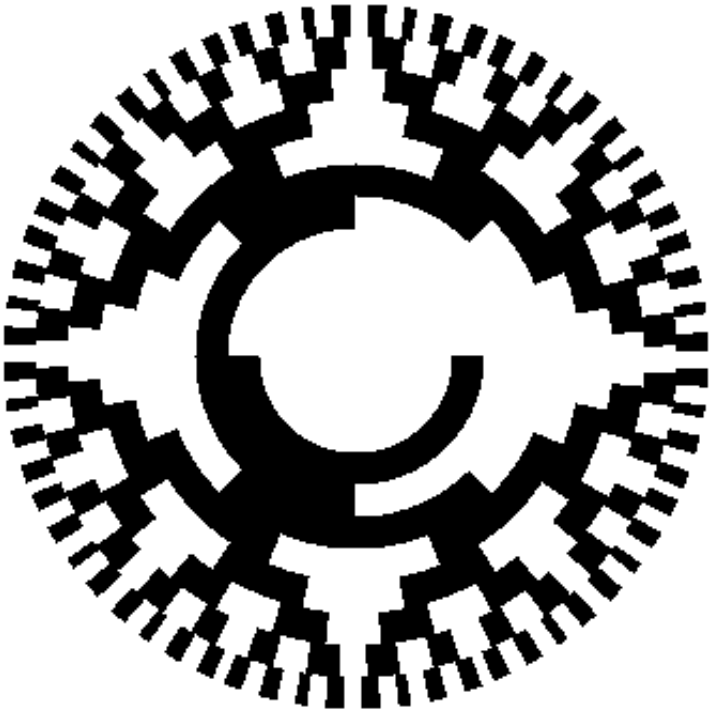
fill_arc On trace l'arc d'ellipse remplie : les deux premiers paramètres sont les coordonnées (entières) du centre, les deux suivantes sont les demi-rayons (entiers) de l'ellipse, s'ils sont égaux on a une portions de cercle, les deux derniers paramètres sont les angles qui délimitent l'arc en degrés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

open_graph " 600x600" On ouvre une fenêtre graphique en donnant sa taille, noter l'espace.

wait_next_event 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**.

`close_graph` ferme la fenêtre.

Question 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.



III 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.

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

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

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

Solutions

Solution de la question 1

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

Solution de la question 2

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

Solution de la question 3

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 diffèrent 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.

Solution de la question 4

```
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;;
```

Solution de la question 5

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 ();;
```

Solution de la question 6

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.

Solution de la question 7
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));;
```

Solution de la question 8
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));;
```