Chapeaux

September 13, 2017

1 Table of Contents

2 Texte d'oral de modélisation - Agrégation Option Informatique

2.1 Préparation à l'agrégation - ENS de Rennes, 2016-17

- *Date* : 7 avril 2017
- *Auteur* : Lilian Besson
- *Texte*: Pour la Science, Nř=319 mai 2014, par Jean-Paul Delahaye, "Couleurs des chapeaux et codes correcteurs d'erreurs"

2.2 À propos de ce document

- Ceci est une *proposition* de correction, partielle et probablement non-optimale, pour la partie implémentation d'un texte d'annale de l'agrégation de mathématiques, option informatique.

- Ce document est un notebook Jupyter, et est open-source sous Licence MIT sur GitHub, comme les autres solutions de textes de modélisation que j'ai écrite cette année.
- L'implémentation sera faite en OCaml, version 4+ :

```
In [1]: Sys.command "ocaml -version";;
```

```
The OCaml toplevel, version 4.04.2
```

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

> Notez que certaines fonctions des modules usuels `List` et `Array` ne sont pas disponibles en OCaml 3.12. J'essaie autant que possible de ne pas les utiliser, ou alors de les redéfinir si je m'en sers.

2.3 Question de programmation

La question de programmation pour ce texte était donnée en question II.1 et II.2 en page 7 :

> - "II.1) Écrire un programme de décodage, qui prend en entrée une suite de caractère codant un symbole (avec une erreur possible), et qui renvoie ce symbole."
> - "II.2) Écrire un programme qui décode une suite de symboles."

2.4 Réponse à l'exercice requis

2.4.1 Rapide vérification du dénombrement annoncé

On veut vérifier, numériquement, que le nombre de stratégies pour le jeu à n joueurs, annoncé à $3^{(n2^{n-1})}$, vaut bien 531441 pour $n = 3$, et de l'ordre de $1.853\ 10^{15}$ pour $n = 4$.

```
In [2]: let denombrement_regles n =
            3. ** (n *. (2. ** (n -. 1.)));;
```

```
Out[2]: val denombrement_regles : float -> float = <fun>
```

```
In [3]: denombrement_regles 3.;;
```

```
Out[3]: - : float = 531441.
```

```
In [4]: denombrement_regles 4.;;
```

```
Out[4]: - : float = 1853020188851841.
```

> C'est bon. C'était inutile, mais autant s'échauffer sur un petit truc rapide et facile.

2.4.2 Q.II.1) Décodage d'un symbole

Structures et types de données Comme toujours, on commence par proposer des structures de données (et déclarer des types) adaptés au problème.

Ici, il nous faut un type de donnée pour les caractères (`0` ou `1`), les suites de caractères (e.g., `0011`), et les symboles (e.g., `'A'`).

```
In [5]: type caractere = int;;

        type suite_caracteres = caractere array;;

        type symbole = string;;
```

```
Out[5]: type caractere = int
```

```
Out[5]: type suite_caracteres = caractere array
```

```
Out[5]: type symbole = string
```

On préfère utiliser des *tableaux* plutôt que des listes puisque leur taille sera fixe, et qu'aucune récursion ne semble utile ici.

Plutôt que de restreindre les symboles à être des `char` (une seule lettre), on autorise directement des chaînes entières.

Ensuite, il faut une structure de données qui représente le *codage de Hamming*, c'est-à-dire la liste des suites de caractère associé à un symbole.

- Une façon élégante serait d'utiliser des tables d'associations (avec le module `Map`), mais c'est un peu trop compliqué pour l'utilisation simple qu'on va en faire.
- On choisit donc d'utiliser de simples listes de paires `(suite, symbole)`, et l'association d'une suite de caractère à son symbole utilisera `List.assoc`.

```
In [6]: type codage = (suite_caracteres * symbole);;

        type codage_hamming = codage list;;
```

```
Out[6]: type codage = suite_caracteres * symbole
```

```
Out[6]: type codage_hamming = codage list
```

Exemples On peut commencer par définir les exemples du textes (page 5).

```
In [7]: let n = 7 ;;

        let suite1 : suite_caracteres = [| 0;0;0;0;0;0;0 |];;
        let suite2 : suite_caracteres = [| 1;1;1;1;1;1;1 |];;
        let suite3 : suite_caracteres = [| 0;1;0;1;0;1;0 |];;
        let suite4 : suite_caracteres = [| 1;0;1;0;1;0;1 |];;
```

```
Out[7]: val n : int = 7


Out[7]: val suite1 : suite_caracteres = [|0; 0; 0; 0; 0; 0; 0|]


Out[7]: val suite2 : suite_caracteres = [|1; 1; 1; 1; 1; 1; 1|]


Out[7]: val suite3 : suite_caracteres = [|0; 1; 0; 1; 0; 1; 0|]


Out[7]: val suite4 : suite_caracteres = [|1; 0; 1; 0; 1; 0; 1|]


In [8]: let codage_ex1 : codage_hamming = [
            (suite1, "A");
            (suite2, "B");
            (suite3, "C");
            (suite4, "D")
        ];;

Out[8]: val codage_ex1 : codage_hamming =
          [([|0; 0; 0; 0; 0; 0; 0|], "A"); ([|1; 1; 1; 1; 1; 1; 1|], "B");
           ([|0; 1; 0; 1; 0; 1; 0|], "C"); ([|1; 0; 1; 0; 1; 0; 1|], "D")]
```

L'association se fait facilement :

```
In [9]: let quel_symbole (code : codage_hamming) (suite : suite_caracteres) : symbole =
            List.assoc suite code
        ;;

Out[9]: val quel_symbole : codage_hamming -> suite_caracteres -> symbole = <fun>


In [10]: quel_symbole codage_ex1 suite3;; (* = "C" *)

Out[10]: - : symbole = "C"
```

Algorithme de décodage d'une séquence de caractères Pourquoi a-t-on besoin de définir un codage (liste des séquences et de leur symbole associé) ?

Et bien tout simplement parce que l'algorithme de décodage demandé en Q.II.2) nécessite de connaître le codage, pour savoir quelle séquence doit être associé à une séquence donnée (quelle erreur a été commise).

L'algorithme en question est assez simple à concevoir :

- Il prendra en entrée : un codage `code` (supposé de Hamming), et une séquence de caractère `seq`.

- Il devra renvoyer un et un seul symbole, associé à la séquence `seq` par le codage de Hamming `code`.

Il fonctionnera comme ça :

- Pour chaque séquence `seq_i` du codage, vérifie si `seq` a été obtenue depuis `seq_i` par *une et une seule* modification d'un caractère (fonction `est_issue`),
- si une unique séquence `seq_i` valide a été trouvée, alors le symbole associé par `code` à `seq_i` est celui associé à `seq`,
- par contre, si aucune séquence ou si au moins deux séquences ont été trouvées, alors l'algorithme échouée (avec `Assert_failure`).

Terminaison L'algorithme décrit ci-dessus ne fait que des boucles `for` sur l'ensemble des séquences du codage de Hamming, et sur les caractères, donc termine (évidemment).

Correction ? Si la séquence d'entrée a effectivement été obtenue par une modification d'un *seul* caractère d'une des séquence du codage d'entrée, l'hypothèse de Hamming impose que l'algorithme va la trouver, et qu'il n'en existe qu'une. Donc le symbole renvoyé par l'algorithme décrit ci-dessus est correct.

Complexité en temps (et espace)

- En mémoire, il n'utilise qu'au maximum une mémoire égale à celle de son entrée, donc l'algorithme de décodage d'*une* séquence est linéaire en mémoire.
- En temps, les deux boucles `for` imbriquées sont en $\mathcal{O}(N)$ pour la boucle sur les `N` séquences de caractères du codage, et en $\mathcal{O}(n)$ pour chaque caractère des séquences, soit en $\mathcal{O}(Nn)$ finalement, ce qui est de l'ordre de la taille du codage, donc l'algorithme de décodage d'*une* séquence est aussi linéaire en temps.

La sous-fonction `est_issue_de` On écrit chaque morceau, un par un.

On a besoin de pouvoir extraire un sous-tableau en enlevant juste une case.

```
In [11]: let sans_j tab j =
             let n = Array.length tab in
             let indice i =
                 if i < j then i else i + 1
             in Array.init (n - 1) (fun i -> tab.(indice i))
             (* Autre approche :
             let gauche = j
             and droite = n - j - 1
             in
             Array.append
                 (Array.init gauche (fun i -> tab.(i)))
                 (Array.init droite (fun i -> tab.(j + i + 1)))
             *)

         ;;
```

```
Out[11]: val sans_j : 'a array -> int -> 'a array = <fun>
```

Note : Ici, j'ai préféré utiliser une fonction qui renvoie `tab.(i)` ou bien `tab.(i + 1)` selon que `i < j` ou `j < i`, mais on pourrait aussi faire deux tableaux, et les concaténer avec `Array.append`.

```
In [12]: sans_j [|0;1;2;3|] 0;;
         sans_j [|0;1;2;3|] 1;;
         sans_j [|0;1;2;3|] 2;;
         sans_j [|0;1;2;3|] 3;;

Out[12]: - : int array = [|1; 2; 3|]


Out[12]: - : int array = [|0; 2; 3|]


Out[12]: - : int array = [|0; 1; 3|]


Out[12]: - : int array = [|0; 1; 2|]
```

Ensuite, il nous faut pouvoir établir que deux tableaux sont égaux (case par case). C'est déjà le cas, via le test = en Caml :

```
In [13]: [|1;2;3|] = [|3;4;1|];;
         [|1;2;3|] = [|1;2;3|]

Out[13]: - : bool = false


Out[13]: - : bool = true
```

Donc on peut facilement vérifier si une séquence `seq` a été issue depuis `seq_i` par une modification de la case numéro `j` :

```
In [14]: let est_issue_de_en_j (seq_i : suite_caracteres) (seq : suite_caracteres) (j : int) =
             let seq_i_sansj = sans_j seq_i j in
             let seq_sansj = sans_j seq j in
             seq_i_sansj = seq_sansj
         ;;

Out[14]: val est_issue_de_en_j : suite_caracteres -> suite_caracteres -> int -> bool =
           <fun>
```

```
In [15]: suite1;;
         est_issue_de_en_j suite1 suite1 0;;

         let suite1_delta0 = [|1; 0; 0; 0; 0; 0; 0|];;

         est_issue_de_en_j suite1 suite1_delta0 0;;
         est_issue_de_en_j suite1 suite1_delta0 1;;
         est_issue_de_en_j suite1 suite1_delta0 4;;
         est_issue_de_en_j suite1 suite1_delta0 6;;

         let suite1_delta5 = [|0; 0; 0; 0; 0; 1; 0|];;

         est_issue_de_en_j suite1 suite1_delta5 3;;
         est_issue_de_en_j suite1 suite1_delta5 4;;
         est_issue_de_en_j suite1 suite1_delta5 5;;
         est_issue_de_en_j suite1 suite1_delta5 6;;

Out[15]: - : suite_caracteres = [|0; 0; 0; 0; 0; 0; 0|]


Out[15]: - : bool = true


Out[15]: val suite1_delta0 : int array = [|1; 0; 0; 0; 0; 0; 0|]


Out[15]: - : bool = true


Out[15]: - : bool = false


Out[15]: - : bool = false


Out[15]: - : bool = false


Out[15]: val suite1_delta5 : int array = [|0; 0; 0; 0; 0; 1; 0|]


Out[15]: - : bool = false


Out[15]: - : bool = false


Out[15]: - : bool = true
```

```
Out[15]: - : bool = false
```

Pour finir, il faut vérifier qu'*un et un seul* indice `j` est valide.

```
In [16]: let est_issue_de (seq_i : suite_caracteres) (seq : suite_caracteres) =
             let taille = Array.length seq_i in
             let bon_j = ref (-1) in
             let nb_bon_j = ref 0 in
             (* Pour chaque *)
             for j = 0 to taille - 1 do
                 if est_issue_de_en_j seq_i seq j
                 then begin
                     bon_j := j;
                     (* s'ils diffèrent en cette case, ie si une vraie modification était néce
                     if seq_i.(j) != seq.(j) then
                         incr nb_bon_j;
                 end;
             done;
             (* Vérifie qu'au maximum une vraie modification a été utilisée *)
             assert (!nb_bon_j <= 1);
             (* 0 <= !bon_j, !bon_j, !nb_bon_j *) (* pour tester *)
             0 <= !bon_j
         ;;

Out[16]: val est_issue_de : suite_caracteres -> suite_caracteres -> bool = <fun>


In [17]: est_issue_de suite1 suite1;;
         est_issue_de suite1 suite1_delta0;;
         est_issue_de suite1 suite1_delta5;;

         let suite1_4delta = [|1; 1; 0; 1; 0; 1; 0|];;
         est_issue_de suite1 suite1_4delta;; (* false *)

Out[17]: - : bool = true


Out[17]: - : bool = true


Out[17]: - : bool = true


Out[17]: val suite1_4delta : int array = [|1; 1; 0; 1; 0; 1; 0|]


Out[17]: - : bool = false
```

Fonction finale `decodage_une_sequence` La même architecture peut-être utilisée :

```
In [19]: let decodage_une_sequence (code : codage_hamming) (seq : suite_caracteres) : symbole =
             let bon_indice = ref (-1) in
             let nb_bonne_seqi = ref 0 in
             (* Pour chaque seq_i du codage de Hamming *)
             List.iteri (
                 fun i (seq_i, _) -> (
                     if est_issue_de seq_i seq
                     then begin
                         bon_indice := i;
                         incr nb_bonne_seqi;
                     end;
                 )
             ) code;
             (* Vérifie qu'une seule séquence du codage soit valide *)
             assert (!nb_bonne_seqi = 1);
             (* Associe le symbole de la seule séquence du codage qui soit valide *)
             let bonne_seqi = fst (List.nth code !bon_indice) in
             quel_symbole code bonne_seqi
         ;;

Out[19]: val decodage_une_sequence : codage_hamming -> suite_caracteres -> symbole =
           <fun>
```

On va faire quelques essais, avec le codage de Hamming suivant, défini plus haut :

```
In [20]: codage_ex1;;

Out[20]: - : codage_hamming =
         [([|0; 0; 0; 0; 0; 0; 0|], "A"); ([|1; 1; 1; 1; 1; 1; 1|], "B");
          ([|0; 1; 0; 1; 0; 1; 0|], "C"); ([|1; 0; 1; 0; 1; 0; 1|], "D")]
```

On va vérifier que pour chacun des 4 séquences, elles sont bien décodées si aucune ou si exactement une case a été changée, mais que le décodage échoue si plus de deux changements ont eu lieu :

```
In [21]: suite1;;
         decodage_une_sequence codage_ex1 suite1;;

         suite1_delta0;;
         decodage_une_sequence codage_ex1 suite1_delta0;;

         suite1_delta5;;
         decodage_une_sequence codage_ex1 suite1_delta5;;

Out[21]: - : suite_caracteres = [|0; 0; 0; 0; 0; 0; 0|]
```

```
Out[21]: - : symbole = "A"


Out[21]: - : int array = [|1; 0; 0; 0; 0; 0; 0|]


Out[21]: - : symbole = "A"


Out[21]: - : int array = [|0; 0; 0; 0; 0; 1; 0|]


Out[21]: - : symbole = "A"
```

Jusqu'ici tout va bien, et sur un exemple avec quatre modifications, la séquence a été trop modifiée pour être encore identifiée comme `"A"` :

```
In [22]: suite1_4delta;;
         decodage_une_sequence codage_ex1 suite1_4delta;; (* va trouver "C" *)

Out[22]: - : int array = [|1; 1; 0; 1; 0; 1; 0|]


Out[22]: - : symbole = "C"


In [23]: suite2;;
         decodage_une_sequence codage_ex1 suite2;;

         let suite2_delta6 = [|1; 1; 1; 1; 1; 1; 0|];;
         decodage_une_sequence codage_ex1 suite2_delta6;;

Out[23]: - : suite_caracteres = [|1; 1; 1; 1; 1; 1; 1|]


Out[23]: - : symbole = "B"


Out[23]: val suite2_delta6 : int array = [|1; 1; 1; 1; 1; 1; 0|]


Out[23]: - : symbole = "B"


In [24]: suite3;;
         decodage_une_sequence codage_ex1 suite3;;

         let suite3_delta3 = [|0; 1; 0; 0; 0; 1; 0|];;
         decodage_une_sequence codage_ex1 suite3_delta3;;
```

```
Out[24]: - : suite_caracteres = [|0; 1; 0; 1; 0; 1; 0|]


Out[24]: - : symbole = "C"


Out[24]: val suite3_delta3 : int array = [|0; 1; 0; 0; 0; 1; 0|]


Out[24]: - : symbole = "C"


In [25]: suite4;;
         decodage_une_sequence codage_ex1 suite4;;

         let suite4_delta1 = [|1; 1; 1; 0; 1; 0; 1|];;
         decodage_une_sequence codage_ex1 suite4_delta1;;

Out[25]: - : suite_caracteres = [|1; 0; 1; 0; 1; 0; 1|]


Out[25]: - : symbole = "D"


Out[25]: val suite4_delta1 : int array = [|1; 1; 1; 0; 1; 0; 1|]


Out[25]: - : symbole = "D"
```

Tout ça semble bien marcher !
On va rapidement réussir à décoder le message bruité donné en exemple dans le texte.

2.4.3 Q.II.2) Décodage d'une suite de symboles

Cette deuxième étape ne va pas être très compliquée, il s'agit essentiellement d'appliquer l'algorithme précédent à chaque sous-séquence de la séquence donnée.

Ces sous-séquences sont faciles à extraire depuis la séquence entière, puisque le texte supposait (implicitement) que toutes les séquences d'un codage de Hamming ont la même taille !

1. on commence par extraire la taille `n` ($n \geq 1$) des séquences du codage de Hamming,
2. et ensuite on découpe la séquence d'entrée, de taille $n \times k$, en une *liste* de $k \geq 1$ sous-séquences, toutes de taille `n`,
3. pour chaque sous-séquence on utilise la fonction `decodage_une_sequence` précédente,
4. on reconstruit une liste de symboles, qu'on renvoie.

1. Extraire la taille des séquence du codage de Hamming

```
In [30]: let taille_sequences_codage (code : codage_hamming) =
             let tailles = List.map (fun (seq, _) -> Array.length seq) code in
             let taille = List.hd tailles in
             (* On vérifie que toutes les séquence du codage aient cette même taille ! *)
             assert (List.for_all (( = ) taille) tailles);
             taille
         ;;

Out[30]: val taille_sequences_codage : codage_hamming -> int = <fun>


In [31]: let n1 = taille_sequences_codage codage_ex1 ;;

Out[31]: val n1 : int = 7
```

Ça marche comme voulu !

2. Extraire les sous-séquences On commence par définir l'exemple d'une séquence longue de 4 sous-séquences de tailles $n = 7$, venant de la page 5.

```
In [32]: let seq_ex1 : suite_caracteres = [|
            1;1;0;1;0;1;0;
            1;0;1;1;1;0;1;
            1;0;0;0;0;0;0;
            0;1;1;1;1;1;1
         |];;

Out[32]: val seq_ex1 : suite_caracteres =
           [|1; 1; 0; 1; 0; 1; 0; 1; 0; 1; 1; 1; 0; 1; 1; 0; 0; 0; 0; 0; 0; 0; 1; 1;
             1; 1; 1; 1|]
```

On va être malin, et découper en utilisant une combinaison des deux fonctions `Array.init` et `Array.sub`.

Note : Le second argument de `Array.sub` est la longueur, pas l'indice de fin.

```
In [33]: Array.sub;;
         Array.init;;

Out[33]: - : 'a array -> int -> int -> 'a array = <fun>


Out[33]: - : int -> (int -> 'a) -> 'a array = <fun>
```

Astuce : Si vous êtes en galère le jour de l'oral, rappelez vous qu'il existe deux modules `ListLabels` et `ArrayLabels` dans la bibliothèque standard, qui sont comme `List` et `Array`, sauf que les signatures de chaque fonction contiennent des arguments *nommés* :

```
In [34]: ArrayLabels.sub;;
         ArrayLabels.init;;

Out[34]: - : 'a array -> pos:int -> len:int -> 'a array = <fun>


Out[34]: - : int -> f:(int -> 'a) -> 'a array = <fun>
```

`Array.sub tab i n` revient à extraire `tab[i],...,tab[i+n-1]`. Par exemple, la deuxième sous-séquence de taille `n = 7` du tableau défini ci-dessus est extraite comme ça :

```
In [35]: Array.sub seq_ex1 ((2 - 1) * 7) 7;;

         ArrayLabels.sub seq_ex1 ~pos:((2 - 1) * 7) ~len:7;;

Out[35]: - : caractere array = [|1; 0; 1; 1; 1; 0; 1|]


Out[35]: - : caractere array = [|1; 0; 1; 1; 1; 0; 1|]
```

C'est ensuite assez facile. On rajoute un test pour être sûr de ne pas rater des caractères.

```
In [36]: let decoupe_sequence (seq : suite_caracteres) (n : int) : suite_caracteres list =
             let len = Array.length seq in
             let k = len / n in
             assert (len = (k * n)); (* Test, bonus *)
             Array.to_list (
                 Array.init k (fun i ->
                     Array.sub seq (i * n) n
                 )
             )
         ;;

Out[36]: val decoupe_sequence : suite_caracteres -> int -> suite_caracteres list =
           <fun>


In [37]: decoupe_sequence seq_ex1 n1;;

Out[37]: - : suite_caracteres list =
         [[|1; 1; 0; 1; 0; 1; 0|]; [|1; 0; 1; 1; 1; 0; 1|]; [|1; 0; 0; 0; 0; 0; 0|];
          [|0; 1; 1; 1; 1; 1; 1|]]
```

Ça marche comme voulu.

3. et 4. Rassembler les morceaux

```
In [38]: let decodage_sequences (code : codage_hamming) (seq : suite_caracteres) : symbole list
             let n = taille_sequences_codage code in
             let seqs = decoupe_sequence seq n in
             List.map (decodage_une_sequence code) seqs
         ;;

Out[38]: val decodage_sequences : codage_hamming -> suite_caracteres -> symbole list =
           <fun>
```

Exemple

```
In [39]: decodage_sequences codage_ex1 seq_ex1;;

Out[39]: - : symbole list = ["C"; "D"; "A"; "B"]
```

On a bien retrouvé l'exemple de l'énoncé. Et voilà.

2.5 Conclusion

Voilà pour les deux questions obligatoires de programmation :

- on a décomposé le problème en sous-fonctions,
- on a essayé d'être fainéant, en réutilisant les sous-fonctions,
- on a fait des exemples et *on les garde* dans ce qu'on présente au jury,
- on a testé la fonction exigée sur un exemple venant du texte,
- mais on a pas vraiment essayé d'en faire plus (à part le calcul au début).

Bien-sûr, ce petit notebook ne se prétend pas être une solution optimale, ni exhaustive.