Lycée Chateaubriand
Informatique MP 2002/03
Mathieu Giraud

TP Caml n° 8 – Corrigé
Majorité qualifiée

1 La procédure de vote

a.

```
let rec total_votes
  | [], [] -> 0
  | [], _
  | _ , [] -> failwith "total_votes : les listes n'ont pas la même longueur"
  | t1::q1, t2::q2 -> it t1
                      then t2 + (total_votes q1 q2)
                      else (total_votes q1 q2) ;;
```

b.

```
let résultat_vote poids majorité config  =  (total_votes poids config) >= majorité ;;
```

2 Votes rapportés à la population

c. Je commence par écrire une fonction `map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list` qui parcourt simultanément deux listes de même longueur et applique une fonction à deux arguments à chaque couple d'éléments de même position. Cette fonction servira plusieurs fois au cours du TP.

```
let rec map2 f l1 l2 = match (l1, l2) with
  | [], [] -> []
  | [], _
  | _ , [] -> failwith "map2 : les listes n'ont pas la même longueur"
  | t1::q1, t2::q2 -> (f t1 t2) :: (map2 f q1 q2) ;;

let pop_votes1  =  map2 (fun x y -> (float_of_int x) /. (float_of_int y))  ;;
```

Certains ont préféré au cours du TP mettre toutes les listes du problème en flottants pour éviter les conversions. C'est une solution, même si je préfère ne faire que des opérations entières tant que cela est suffisant.

Il est même possible de faire encore plus générique, par exemple :

```
let rec map3 b f l1 l2 = match (l1, l2) with
  | [], [] -> b
  | [], _
  | _ , [] -> failwith "map3 : les listes n'ont pas la même longueur"
  | t1::q1, t2::q2 -> f t1 t2 (map3 b f q1 q2) ;;


let pop_votes1 = map3 []
                 (fun x y l -> ((float_of_int x) /. (float_of_int y)) :: l) ;;

let total_votes = map3 0
                  (fun vote poids total -> if vote then (poids + total) else (total)) ;;
```

d. La fonction `normalise` peut se réduire à un appel de la fonction `map`.

```
let normalise = function
  | [] -> []
  | t::q -> 1.0 :: (map (fun x -> x /. t) q) ;;

let pop_votes poids config = normalise (pop_votes1 poids config) ;;
```

3 Indices de Banzhaf-Coleman

e. Dans le système `[5; 6; 7; 8]` avec majorité absolue `14` :

- `5` fait basculer la seule coalition `(6,7)`;
- `6` fait basculer les trois coalitions `(8)`, `(5,7)`, et `(5,8)`;
- `7` fait basculer les trois coalitions `(8)`, `(5,6)`, et `(5,8)`;
- et enfin `8` fait basculer les cinq coalitions `(6)`, `(7)`, `(5,6)`, `(5,7)` et `(6,7)`.

Les indices de Banzhaf-Coleman sont donc ici de `[1; 3; 3; 5]`.

3.1 Indice pour un pays

f. La fonction `compte_configs` est très similaire à la fonction `compte_paiements` avec les porte-monnaies : dénombrer, c'est ajouter des valeurs `1` ou `0` sur toutes les 2^n configurations possibles. Pour cela, on on fait circuler un argument supplémentaire `total` représentant le poids des votes favorables. Une première version peut ainsi s'écrire

```
let compte_configs mini maxi poids =
  let rec aux total l = function
    | [] -> if (mini <= total) && (total <= maxi) then 1 else 0
    | t::q -> (aux total q) + (aux (total + t) q)
  in aux 0 poids ;;
```

Il est possbile d'éviter l'argument `total` en écrivant par exemple :

```
let rec compte_configs mini maxi poids =
    | [] -> if (mini <= 0) && (0 <= maxi) then 1 else 0
    | t::q -> (compte_configs mini maxi q) + (compte_configs (mini-t) (maxi-t) q) ;;
```

On peut ensuite chercher à améliorer légèrement cette fonction en arrêtant le traitement lorsqu'on est sûr de ne plus pouvoir arriver dans une bonne situation, par exemple lorsque le `total` courant est déjà supérieur à la limite `maxi`.

```
let compte_configs mini maxi poids =
  let rec aux total l =
    if (total > maxi)
    then 0
    else match l with
          | [] -> if (mini <= total) then 1 else 0
          | t::q -> (aux total q) + (aux (total + t) q)
  in aux 0 poids ;;
```

Même avec cette amélioration, la fonction examine dans le cas le pire les 2^n configurations existantes.

g. Étant donné un seuil s, le vote d'un pays de poids n_0 est décisif si la somme N des poids des autres pays votant favorablement vérifie $N < s$ et $N + n_0 \geq s$, c'est à dire si $s - n_0 \leq P \leq s - 1$.

```
let bancol1 majorité poids1 poidssauf1 =
   compte_configs (majorité - poids1) (majorité - 1) poidssauf1 ;;
```

3.2 Indice pour tous les pays

h. On peut tout d'abord penser à écrire `bancol` qui mime un comportement de tableau en écrivant préalablement des fonctions d'accès au n^{e}élément.

```
let bancol majorité poids =
  let s = ref []
  in for i = (longueur poids) - 1 downto 0  do
       s := (bancol1 majorité (nième i poids) (enlève i poids)) :: !s
     done ;
     !s ;;

#bancol 14 [5; 6; 7; 8] ;;
- : int list = [1; 3; 3; 5]
```

Une autre solution qui m'a été suggérée par l'un d'entre vous consiste à reconstruire la liste des poids au fur et à mesure.

```
let bancol majorité poids =
  let rec aux déjà = function
     | []   -> []
     | t::q -> (bancol1 majorité t (déjà @ q)) :: (aux (t::déjà) q)
  in aux [] poids ;;

#bancol 50 [49; 49; 2] ;;
- : int list = [2; 2; 2]
```

Dans les deux cas, les opérations sur les listes mènent à une complexité de $\mathcal{O}(p^2)$ (p appels de `nieme/enleve` ou de `@`). Mais cette complexité est de toute façon négligeable aux p appels de `bancol1` qui sont à chaque fois exponentiels : la complexité totale est de $\mathcal{O}(p2^p)$.

i. La fonction `pop_bancol` peut se réaliser en utilisant les fonctions `map2` et `normalise` précédemment définies.

```
let pop_bancol majorité poids populations =
    normalise (  map2 (fun s p -> (float_of_int s) /. (float_of_int p))
                      (bancol majorité poids) populations
              ) ;;
```

On programme rapidement une fonction générique `trouve_opt` qui permet de trouver le maximum et le minimum d'une liste, puis on s'en sert pour écrire la fonction `ecart`.

```
let rec trouve_opt opt = function
   | [] -> failwith "trouve_opt : liste vide"
   | [a] -> a
   | t :: q -> opt t (trouve_opt opt q) ;;

let ecart majorité poids populations =
  let pb = pop_bancol majorité poids populations
  in (trouve_opt max pb) /. (trouve_opt min pb) ;;
```

j. Nous avons vu ci-dessus que la complexité est de $\mathcal{O}(p2^p)$. Sur les ordinateurs de la salle de TP, un appel à `pop_bancol` avec $p = 15$ dure déjà entre 1 et 2 secondes. Cette durée est problématique car de nombreux cas réels comportent plus de quinze votants. Il existe d'autres méthodes plus rapides pour cacluler les indices de Banzhaf-Coleman :

- On peut traiter quelques configurations au hasard parmi les 2^p possibles (méthodes à la Monte-Carlo). La qualité des résultats dépend alors du "hasard" choisi...
- Une autre méthode consiste à construire un tableau donnant pour chaque total de voix (de 1 à n) le nombre de configurations y arrivant. On arrive ainsi à des algorithmes en $\mathcal{O}(p)$ mais qui nécessitent une mémoire $\mathcal{O}(n)$.

4 Recherche des poids optimaux

k. Il y a C_{n+p-1}^{p-1} systèmes de poids donnant n votes à p pays.

l. On suppose que l'on a programmé la fonction `add` tel que `add k i` ajoute l'entier `k` au `i`^{e} élément de la liste `l`. On écrit aussi des fonctions pour trouver en quel endroit une liste atteint son minimum et son maximum :

```
let arg_opt opt l =
   let valeur = trouve_opt opt l
   in let rec aux n = function
         | [] -> failwith "arg_opt"
         | t::q -> if (t = valeur) then n else aux (n+1) q
   in aux 0 l ;;
```

On peut maintenant écrire la fonction `iter_bancol`. Celle-ci doit renvoyer la liste `poids` juste modifiée en deux positions.

```
let iteration_bancol majorité poids populations =
    let ps = pop_bancol majorité poids populations
    in let arg_max = arg_opt max ps
       and arg_min = arg_opt min ps
    in add (1) arg_min (add (-1) arg_max poids) ;;
```

m. Chaque étape de l'algorithme faisant varier deux poids d'une valeur `1`, cette méthode ne converge pas. On peut par exemple écrire une fonction effectuant n itérations puis s'arrêtant...

```
let rec converge majorité poids populations = function
  | 0 -> poids
  | n -> let p = iteration_bancol majorité poids populations
         in print_list_int p ;
         converge majorité p populations (n-1) ;;
```

Dans le test qui suit, j'ai ajouté quelques lignes pour le programme affiche à chaque étape les indices rapportés à la population (normalisés par rapport à `1000`) et la nouvelle valeur de `poids`. Le début n'est pas surprenant : le Luxembourg perd progressivement ses sièges au profit de l'Allemagne.

```
#converge 169 euro_poids euro_pops 4 ;;

   [ 1000; 1378; 1387; 1424; 1940; 2408; 3404; 3505; 3590; 3361; 3676; 4037; 4162; 5699; 32192 ]
[ 30; 29; 29; 29; 27; 13; 12; 12; 12; 10; 10; 7; 7; 7; 3 ]
   [ 1000; 1312; 1321; 1356; 1882; 2458; 3259; 3357; 3438; 3259; 3564; 3636; 3748; 5133; 21107 ]
[ 31; 29; 29; 29; 27; 13; 12; 12; 12; 10; 10; 7; 7; 7; 2 ]
   [ 1000; 1288; 1297; 1332; 1825; 2286; 3245; 3342; 3423; 3142; 3436; 3599; 3709; 5080; 14408 ]
[ 32; 29; 29; 29; 27; 13; 12; 12; 12; 10; 10; 7; 7; 7; 1 ]
   [ 1000; 1265; 1274; 1308; 1807; 2319; 3129; 3222; 3300; 3169; 3466; 3438; 3543; 4852; 10445 ]
[ 33; 29; 29; 29; 27; 13; 12; 12; 12; 10; 10; 7; 7; 7; 0 ]

- : int list = [33; 29; 29; 29; 27; 13; 12; 12; 12; 10; 10; 7; 7; 7; 0]
```

Enfin, je laisse la question sur le point de vue des citoyens ouverte...

Voilà, les TPs sont finis !

*N'hésitez pas à m'écrire (**magiraud@free.fr**) pour*
- *me demander des précisions sur tel ou tel point*
- *me dire ce qui était bien / ce qui ne l'était pas dans les TPs*
- *me donner des nouvelles de vous / de vos résultats aux épreuves d'info*
- *...*

Bon courage à tous pour vos révisions et les concours !