1 Le dilemme du prisonnier

Soit $(c, p, d, t) \in \mathbb{Z}^4$ fixés tels que $t > c > p > d$ et $2c > t + d$.

On considère le jeu à deux joueurs suivant : lors d'une manche, les joueurs choisissent simultanément de **coopérer** ou de **trahir**. À l'issue de la manche :

- si les deux joueurs coopèrent, ils marquent chacun c points ;
- si les deux joueurs trahissent, ils marquent chacun p points ;
- si l'un des joueurs coopère et l'autre trahit :
  - celui qui coopère (et s'est fait duper) marque d points ;
  - celui qui a trahi marque t points.

Lors d'une **partie**, les joueurs jouent n manches, chaque choix étant fait en fonction de ce qui s'est passé aux manches précédentes. Le **gain** d'un joueur est alors la somme des gains de ses manches, et son objectif est de maximiser ce gain, indépendamment du gain de l'autre joueur.

1. En binôme, faire une partie de 10 manches avec les paramètres : $t = 5$, $c = 3$, $p = 1$, $d = 0$.

2 Implémentation en OCaml

On définit le type OCaml suivant pour représenter un coup :

```
type coup = Coop | Trahi
```

On représente une partie en cours par deux listes de type `coup list`, chacune d'entre elles contenant l'historique des coups joués par un joueur depuis le début de la partie (l'élément en tête de la liste est le coup le plus récent).

Une **stratégie** est une fonction de signature `coup list -> coup list -> coup` prenant en arguments deux listes `moi` et `autre` contenant l'historique des coups d'un joueur et de son adversaire, et renvoie le coup à jouer au prochain tour.

Par soucis de clarté, on définit les types suivants :

```
type strat = coup list -> coup list -> coup
type param = int * int * int * int              (* représente (t,c,p,d) *)
```

Par exemple, la stratégie suivante coopère toujours :

```
let bisounours moi autre = Coop
```

2. Écrire des fonctions implémentant les stratégies suivantes :
   (a) `traitre : coup list -> coup list -> coup` qui trahit toujours ;
   (b) `sympa_irl : coup list -> coup list -> coup` qui alterne entre deux coopérations successives et une trahison.

Exemple

```
# let rec test_Q2 l n = match n with
    | 0 -> l
    | _ -> test_Q2 (sympa_irl l [] :: l) (n-1)
  in test_Q2 [] 9
- : coup list = [Trahi; Coop; Coop; Trahi; Coop; Coop; Trahi; Coop; Coop]
```

3. Écrire une fonction `partie : int -> strat -> strat -> (coup list * coup list)` prenant en arguments le nombre n de manches à jouer et deux stratégies, et qui simule une partie de n manches entre ces deux stratégies puis renvoie les deux listes des coups joués lors de la partie.

Exemple

```
# partie 5 traitre sympa_irl ;;
- : coup list * coup list =
([Trahi; Trahi; Trahi; Trahi; Trahi], [Coop; Coop; Trahi; Coop; Coop])
```

4. Écrire une fonction `score : param -> coup list -> coup list -> (int * int)` prenant en arguments les paramètres (t, c, p, d) ainsi que deux listes de coups représentant une partie, et renvoyant le score de chaque joueur.

Exemple

```
# let l_traitre, l_sympa = partie 5 traitre sympa_irl in
score (5,3,1,0) l_traitre l_sympa ;;
- : int * int = (21, 1)
```

5. Écrire des fonctions implémentant les stratégies suivantes :
   (a) `rancunier : coup list -> coup list -> coup` qui coopère tant qu'il n'a pas été trahi, et répond à une trahison en trahissant jusqu'à la fin ;

Exemple

```
# rancunier [] [Coop; Coop; Coop; Coop; Coop] ;;
- : coup = Coop
# rancunier [] [Coop; Coop; Trahi; Coop; Coop] ;;
- : coup = Trahi
```

   (b) `miroir_gentil : coup list -> coup list -> coup` qui coopère au premier tour, puis joue le coup précédent de l'autre joueur ;

Exemple

```
# miroir_gentil [] [] ;;
- : coup = Coop
# miroir_gentil [] [Trahi; Coop; Coop] ;;
- : coup = Trahi
# miroir_gentil [] [Coop; Trahi; Trahi] ;;
- : coup = Coop
```

   (c) `miroir_mechant : coup list -> coup list -> coup` qui trahit au premier tour, puis joue le coup précédent de l'autre joueur ;

Exemple

```
# miroir_mechant [] [] ;;
- : coup = Trahi
# miroir_mechant [] [Trahi; Coop; Coop] ;;
- : coup = Trahi
# miroir_mechant [] [Coop; Trahi; Trahi] ;;
- : coup = Coop
```

(d) `majorite : coup list -> coup list -> coup` qui trahit si et seulement si l'autre joueur a joué strictement plus de trahisons que de coopérations ;

Exemple

```
# majorite [] [] ;;
- : coup = Coop
# majorite [] [Trahi; Coop] ;;
- : coup = Coop
# majorite [] [Coop; Trahi; Trahi] ;;
- : coup = Trahi
```

(e) `mastermind : coup list -> coup list -> coup` qui commence la partie par une trahison et deux coopérations, puis :

- si l'autre joueur a coopéré les 3 premiers tours, trahit jusqu'à la fin ;
- sinon, joue le coup précédent de l'autre joueur.

Exemple

```
# mastermind [] [] ;;
- : coup = Trahi
# mastermind [] [Coop] ;;
- : coup = Coop
# mastermind [] [Coop; Coop] ;;
- : coup = Coop
# mastermind [] [Coop; Coop; Coop] ;;
- : coup = Trahi
# mastermind [] [Trahi; Coop; Coop] ;;
- : coup = Trahi
```

On peut tirer au hasard une valeur $x \in [0, 1]$ (avec probabilité uniforme) via l'instruction suivante :

```
let x = Random.float 1. in ...
```

6. Écrire une fonction `indecis : float -> strat` prenant en argument un nombre flottant p (supposé entre 0 et 1), et qui renvoie la stratégie jouant à chaque coup une coopération avec probabilité p.

Exemple

```
# let test = indecis 0.5 in
  let rec test_Q6 l n = match n with
    | 0 -> l
    | _ -> test_Q6 (test l [] :: l) (n-1)
  in test_Q6 [] 9
- : coup list = [Coop; Trahi; Coop; Coop; Trahi; Coop; Trahi; Coop; Trahi]
```

On peut tirer au hasard un entier $n \in [\![0, n_{\max} - 1]\!]$ (avec probabilité uniforme) via l'instruction suivante :

```
let n = Random.int n_max in ...
```

7. Écrire une fonction `tournoi : param -> strat array -> int array` prenant en arguments les paramètres (t, c, p, d) ainsi qu'un tableau de stratégies, et qui :

- fixe un nombre de manches tirées aléatoirement entre 500 et 1000 ;
- fait se rencontrer chaque paire de stratégies ;
- renvoie un tableau des scores cumulés de chaque stratégie.

3 Détection des erreurs

Lorsqu'on fait s'affronter deux stratégies, l'une d'entre elles pourrait être buguée (erreur d'indice, division par zéro, etc). Dans ce cas, Ocaml va lever une exception lors de l'exécution de la stratégie.

On décide que lors d'une telle partie avec une stratégie buguée, la stratégie qui vient de buguer perd automatiquement la partie avec un score total de 0, et son adversaire gagne avec un score de 10 000.

8. Modifier votre fonction `partie` pour qu'elle détecte si l'une des stratégies a rencontré un bug et doit perdre instantanément. Dans un tel cas, on lèvera l'une des exceptions suivantes (selon quelle stratégie a perdu).

```
exception Perdu_J1
exception Perdu_J2
```

9. Modifier votre fonction tournoi pour qu'elle gère ces exceptions, et mette à jour les scores correctement en cas de stratégie buguée.

4 Devoir maison

Implémenter une stratégie originale qui participera à un tournoi rassemblant toutes les stratégies de ce TP (pour `indecis`, on utilisera $p = 0.4$), et les stratégies personnalisées des autres élèves de la classe, avec les paramètres $t = 5$, $c = 3$, $p = 1$, $d = 0$. La stratégie gagnante remportera des chouquettes !

Voici les consignes à respecter :

- votre stratégie doit m'être envoyée par mail (`anthony.lick@gmail.com`) avant le dimanche 13/10/2024 à 23h59 ;
- votre stratégie doit fonctionner (testez-là sur des exemples !) ;
- votre fonction doit s'appeler `strat_nom` et être toute seule dans un fichier `nom.ml` (où `nom` est à remplacer par votre nom).