MINISTÈRE
DE L'ÉDUCATION
NATIONALE
Liberté
Égalité
Fraternité

EAE INF 2

SESSION 2025

AGREGATION
CONCOURS EXTERNE

Section : INFORMATIQUE

ÉTUDE D'UN PROBLÈME INFORMATIQUE

Durée : 6 heures

L'usage de tout ouvrage de référence, de tout dictionnaire et de tout matériel électronique (y compris la calculatrice) est rigoureusement interdit.

Il appartient au candidat de vérifier qu'il a reçu un sujet complet et correspondant à l'épreuve à laquelle il se présente.

Si vous repérez ce qui vous semble être une erreur d'énoncé, vous devez le signaler très lisiblement sur votre copie, en proposer la correction et poursuivre l'épreuve en conséquence. De même, si cela vous conduit à formuler une ou plusieurs hypothèses, vous devez la (ou les) mentionner explicitement.

NB : Conformément au principe d'anonymat, votre copie ne doit comporter aucun signe distinctif, tel que nom, signature, origine, etc. Si le travail qui vous est demandé consiste notamment en la rédaction d'un projet ou d'une note, vous devrez impérativement vous abstenir de la signer ou de l'identifier.
Le fait de rendre une copie blanche est éliminatoire.

Tournez la page S.V.P.

INFORMATION AUX CANDIDATS

Vous trouverez ci-après les codes nécessaires vous permettant de compléter les rubriques figurant en en-tête de votre copie.

Ces codes doivent être reportés sur chacune des copies que vous remettrez.

Concours	Section/option	Epreuve	Matière
EAE	6200A	102	9423

IMPRIMERIE NATIONALE – 25 0114 – D'après documents fournis

Mastermind

Ce sujet est consacré à l'étude d'un jeu de stratégie, inspiré du jeu de société Mastermind. On s'intéresse en particulier à la recherche de stratégies optimales et aux moyens d'effectuer cette recherche efficacement.

Préliminaires

Organisation du sujet Le sujet est constitué de quatre parties. Les définitions, concepts et idées introduits dans une partie sont généralement utiles pour les parties suivantes ; cependant, il est toujours possible d'admettre le résultat d'une question (ou de supposer qu'une fonction demandée à une question a été écrite) pour traiter les questions suivantes.

Attendus Il est attendu des candidates et des candidats des réponses construites. Leur évaluation portera aussi sur la précision, le soin et la clarté de la rédaction.

Programmation Au début de chaque partie comportant des questions de programmation, le langage à utiliser (Python, OCaml ou C suivant la partie) est précisé : cette consigne devra impérativement être respectée.

Pour les parties en Python, on se limitera aux fonctions disponibles sans effectuer de `import`.

Pour les parties en C, on supposera que les fichiers d'entête suivants ont été inclus : `stdlib.h`, `stdio.h`, `string.h`, `assert.h`, `stdbool.h` et `stdint.h`.

Pour les parties en OCaml, on pourra utiliser librement les fonctions des modules `List`, `Array` et `Hashtbl` de la bibliothèque standard, ainsi que les fonctions du module initialement ouvert (celles n'étant pas préfixées par un nom de module, comme `incr`, `min`, ...). On rappelle la spécification de quelques fonctions d'ordre supérieur pouvant être utiles :

- `List.map : ('a -> 'b) -> 'a list -> 'b list`
  `List.map f [x1; ...; xn]` renvoie `[f x1; ...; f xn]` ;
- `List.iter : ('a -> unit) -> 'a list -> unit`
  `List.iter f [x1; ...; xn]` équivaut à `f x1; ...; f xn; ()` ;
- `List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
  `List.fold_left f init [x1; ...; xn]` renvoie `f (... (f (f init x1) x2) ...)` ;
- `List.filter : ('a -> bool) -> 'a list -> 'a list`
  `List.filter pred u` renvoie la liste constituée des éléments `x` de `u` tels que `pred x = true`, dans l'ordre de la liste `u`.

Ces fonctions (sauf `List.filter`) sont également présentes dans le module `Array`, avec un type adapté.

Pour le module `Hahstbl`, on rappelle les fonctions essentielles :

— `('a, 'b) Hashtbl.t` est le type d'une table de hachage dont les clés sont de type `'a` et les valeurs de type `'b` ;
— `Hashtbl.create : int -> ('a, 'b) Hashtbl.t`
`Hashtbl.create 1` renvoie une table de hachage vide (l'argument entier est une indication sur la taille initiale du tableau sous-jacent et pourra systématiquement être pris égal à 1) ;
— `Hahstbl.find_opt : ('a, 'b) Hashtbl.t -> 'a -> 'b option`
`Hahstbl.find_opt h x` renvoie `Some y` si `y` est la valeur associée à la clé `x` dans `h`, ou `None` si la clé `x` n'est pas présente dans `h` ;
— `Hashtbl.replace : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit`
`Hashtbl.replace h x y` associe la clé `x` à la valeur `y` dans la table `h`, en remplaçant l'ancienne association pour `x` s'il y en avait une (et en créant l'association dans le cas contraire) ;
— `Hashtbl.iter : ('a -> 'b -> unit) -> ('a, 'b) Hashtbl.t -> unit`
`Hashtbl.iter f h` équivaut à `f x1 y1; ... ; f xn yn; ()`, où $(x_1, y_1), \ldots, (x_n, y_n)$ sont les associations présentes dans la table `h`, dans un ordre non spécifié.

On supposera que les fonctions `Hashtbl.create`, `Hashtbl.find_opt` et `Hashtbl.replace` s'exécutent en temps constant, et que `Hashtbl.iter` prend un temps proportionnel à la somme des complexités des appels à `f` effectués.

Présentation informelle du jeu

Dans sa version standard, ce jeu fait intervenir un *arbitre* et un joueur, que l'on appellera J_1. L'arbitre commence par choisir une *combinaison* secrète, qui est constituée d'une suite de N jetons colorés (N est un paramètre du jeu, fixé à l'avance). La couleur de chaque jeton de la combinaison est choisie librement parmi un ensemble de P couleurs, l'ordre des jetons à l'intérieur de la combinaison est important et les répétitions sont autorisées.

Par exemple, si $N = 5$ et $P = 3$ (disons que les couleurs possibles sont le rouge, le vert et le bleu, notées R, V, B), l'arbitre peut choisir la combinaison (R, R, V, B, R). Le but du joueur J_1 est de deviner cette combinaison en utilisant le moins d'essais possibles. La partie se déroule comme suit :
— J_1 propose une combinaison c : par exemple, $c =$ (V, B, V, R, B) ;
— l'arbitre répond à J_1 en indiquant le nombre de jetons bien placés et mal placés dans sa combinaison ;
— ici, il y a un V bien placé :

R	R	**V**	B	R
V	B	**V**	R	B

— pour déterminer le nombre de jetons mal placés, on commence par éliminer les jetons bien placés puis l'on oublie l'ordre – ici, il y a deux jetons mal placés (un R et un B) ;

R	R	R			**B**	
R			V		**B**	B

— si l'arbitre répond que tous les jetons sont bien placés (réponse $(N, 0)$), la partie s'arrête ;
— sinon, J_1 fait une nouvelle proposition et l'on continue.

Formalisation

- On se donne deux entiers strictement positifs N et P – dans la version standard du jeu, on a $N = 4$ et $P = 6$.
- On appelle *combinaison* un N-uplet d'éléments de $\{0, \ldots, P-1\}$. *Attention, il s'agit bien d'un N-uplet et non d'une combinaison au sens usuel en mathématiques : l'ordre compte et les répétitions sont autorisées.*
- On notera C (ou $C(N, P)$ s'il y a risque d'ambiguïté) l'ensemble de ces combinaisons : $C = \{0, \ldots, P-1\}^N$. On a donc $|C| = P^N$.
- On appellera *couleurs* les éléments de $\{0, \ldots, P-1\}$ et *jetons* les éléments du N-uplet.
- Étant données deux combinaisons $x = (x_0, \ldots, x_{N-1})$ et $y = (y_0, \ldots, y_{N-1})$, la *similarité* $sim(x, y)$ entre x et y est le couple d'entiers (b, m) où :
  - b est le nombre de jetons *bien placés* de y (par rapport à x), c'est-à-dire le nombre d'indices $i \in \{0, \ldots, N-1\}$ tels que $x_i = y_i$;
  - m est le nombre de jetons *mal placés* de y (par rapport à x). Ce nombre est tel que $b+m$ soit égal au nombre de jetons en commun entre x et y si l'on ne tient pas compte de l'ordre.

  Autrement dit, on commence par compter le nombre de jetons bien placés puis on les élimine et l'on compte le nombre de jetons identiques sans tenir compte de l'ordre, comme illustré au paragraphe précédent.
- Quelques exemples :
  - $sim((2,3,0,2,3),(3,3,3,1,2)) = (1,2)$ – en effet, il y a au total trois éléments communs (un « 2 » et deux « 3 »), dont l'un est bien placé (le « 3 » à l'indice 1, c'est-à-dire en deuxième position) ;
  - $sim((1,2,3),(2,2,2)) = (1,0)$ (un seul élément commun, le « 2 » à l'indice 1 qui est bien placé) ;
  - $sim((1,2,3,1,4),(2,4,3,1,1)) = (2,3)$ (à l'ordre près, tous les éléments sont communs, et deux d'entre eux sont bien placés).
- De manière immédiate, on a $x = y$ si et seulement si $sim(x, y) = (N, 0)$. On remarque de plus que sim est symétrique.

Question 1. Donner sans justification les valeurs de :
- $sim((2,1,3,4),(1,2,3,4))$;
- $sim((0,0,1,1),(1,1,3,0))$;
- $sim((0,3,3,2,3),(3,0,3,4,4))$.

Notations et définitions supplémentaires

- On note R l'ensemble des similarités (ou *réponses*) possibles :

$$R = \{sim(x, y) \mid x, y \in C\}.$$

- Pour $X \subseteq C$ et $c \in C$, on note

$$sim(X, c) = \{sim(x, c) \mid x \in X\}.$$

- Pour $X \subseteq C$ et $c \in C$ et $r \in R$, on note

$$filtre(X, c, r) = \{x \in X \mid sim(c, x) = r\}.$$

Tournez la page S.V.P.

— Pour une liste (éventuellement vide) $h = [(c_1, r_1), \ldots, (c_k, r_k)]$ (appelée *historique*) de couples (combinaison, similarité), on définit l'ensemble des combinaisons *compatibles avec* h par

$$compat(h) = \{x \in C \mid \forall i \in \{1, \ldots, k\},\ sim(x, c_i) = r_i\}.$$

— Un tel historique est dit *admissible* si $compat(h) \neq \emptyset$ et si les r_i avec $1 \leq i < k$ sont différents de $(N, 0)$.
— Un historique admissible est dit *inachevé* si $k = 0$ (l'historique est vide) ou $r_k \neq (N, 0)$.

Jeu à un joueur Le principe du jeu est le suivant :
— une combinaison $but \in C$ est choisie par l'arbitre et gardée secrète ;
— le joueur J_1 doit deviner cette combinaison en un minimum d'essais ;
— pour ce faire, il propose à chaque coup une combinaison $c \in C$;
— l'arbitre indique alors au joueur J_1 la valeur de $sim(x, but)$;
— si cette valeur est $(N, 0)$ (c'est-à-dire si $x = but$), la partie s'arrête ;
— sinon, le joueur J_1 propose une nouvelle combinaison ;
— le score du joueur J_1 est le nombre total d'essais effectués pour trouver *but*. Le joueur J_1 cherche donc à minimiser ce score. Si la partie ne se termine jamais, on considère que ce score est infini.

Exemple. On prend ici $N = 3$ et $P = 3$.
— L'arbitre choisit $but = (2, 2, 0)$ comme combinaison secrète à deviner.
— Le joueur J_1 joue $(0, 0, 0)$, l'arbitre répond $sim((0, 0, 0), but) = (1, 0)$.
— Le joueur J_1 joue $(0, 1, 2)$, l'arbitre répond $sim((0, 1, 2), but) = (0, 2)$.
— Le joueur J_1 joue $(2, 2, 0)$, l'arbitre répond $(3, 0)$ et la partie se termine. Le score de la partie vaut 3.

Partie I. Programmation des fonctions élémentaires

Dans cette partie, on se propose d'implémenter **en utilisant le langage Python** les fonctions de base permettant de jouer, ainsi qu'une stratégie très simple pour le joueur J_1.
On suppose définies deux constantes globales `N` et `P`, et l'on représente :
— une combinaison par une `list` de longueur `N` à valeurs dans $\{0, \ldots, \texttt{P} - 1\}$;
— une réponse par un couple `(b, m)` d'entiers.
Une fonction prenant en entrée une combinaison pourra toujours supposer qu'elle est bien formée (c'est-à-dire de longueur N et ne contenant que des entiers de $\{0, \ldots, P - 1\}$), et une fontion renvoyant une combinaison devra s'assurer qu'elle est bien formée.

I.1 Calcul de la similarité

Question 2. Écrire une fonction `bien_ou_mal_places` prenant en entrée deux combinaisons x et y et renvoyant l'entier $b+m$, où $(b, m) = sim(x, y)$. On demande une complexité en $\mathcal{O}(N+P)$, que l'on justifiera brièvement.

Question 3. Écrire une fonction `sim` prenant en entrée deux combinaisons x et y et renvoyant $sim(x, y)$. Préciser sa complexité, en la justifiant brièvement.

Question 4. Écrire une fonction `compatible` prenant en entrée une combinaison c et un historique h, sous la forme d'une liste de couples (combinaison, réponse), et indiquant si $c \in compat(h)$.

I.2 Stratégie naïve

On suppose que le joueur J_1 utilise la stratégie suivante : il joue systématiquement la plus petite combinaison (dans l'ordre lexicographique) qui est compatible avec l'historique actuel.

Question 5. Écrire une fonction `avance` prenant en entrée une combinaison c et la modifiant en la remplaçant par la combinaison suivante dans l'ordre lexicographique (cette fonction ne renverra donc rien). Si cette fonction est appelée sur la liste `[P - 1, ..., P - 1]` (dernière combinaison dans l'ordre lexicographique), elle lèvera l'exception `ValueError`. Préciser sa complexité dans le pire cas (on ne demande pas de justification).

Question 6. On note $\Phi(c)$ le nombre d'occurrences de $P - 1$ dans la combinaison c et $T(c)$ le nombre d'éléments de `c` modifiés pendant l'appel `avance(c)`. En notant c' l'état de la liste c après l'appel, majorer la quantité $T(c) + \Phi(c') - \Phi(c)$.

Question 7. Que peut-on dire du coût total d'une série de k appels successifs sur une combinaison initialement égale à $(0, \dots, 0)$ (en supposant que les k appels ont lieu sans déclencher d'exception) ? Quel type de calcul de complexité vient-on d'effectuer ?

Question 8. Écrire une fonction `joue_naif` prenant en entrée la combinaison but choisie par l'arbitre et renvoyant l'historique de la partie que l'on obtient si le joueur J_1 utilise la stratégie naïve. Par exemple, pour $N = 3$ et $P = 4$, l'appel `joue_naif([3, 3, 1])` doit renvoyer la liste :

```
[([0, 0, 0], (0, 0)), ([1, 1, 1], (1, 0)), ([1, 2, 2], (0, 1)),
 ([3, 1, 3], (1, 2)), ([3, 3, 1], (3, 0))]
```

Partie II. Généralités sur les stratégies

Dans toute cette partie, D désigne une partie non vide de C.

II.1 Arbre de stratégie

Un *D-arbre de stratégie* est un arbre dont chaque nœud interne est étiqueté par une combinaison $c \in C$ et chaque arête est étiqueté par une réponse $r \in R$ et qui vérifie les conditions suivantes :

— en notant c l'étiquette de la racine, on a un sous-arbre pour chaque $r \in \mathit{sim}(D, c)$ (et l'arête reliant la racine à ce sous-arbre est étiquetée r) ;
— si $(N, 0) \in \mathit{sim}(D, c)$, alors le sous-arbre correspondant est une feuille (que l'on étiquettera parfois c, même si cette étiquette est redondante) ;
— pour chaque $r \in \mathit{sim}(D, c)$, $r \neq (N, 0)$:
  — $|\mathit{filtre}(D, c, r)| < |D|$;
  — le sous-arbre correspondant à r est un $\mathit{filtre}(D, c, r)$-arbre de stratégie.

On note $\mathcal{T}_D$ l'ensemble des D-arbres de stratégie.

Exemples

— Si $D = \{c\}$ (singleton), alors la combinaison à la racine est nécessairement c (tout autre choix conduirait à violer la condition $|\mathit{filtre}(D, c, r)| < |D|$) et l'unique D-arbre de stratégie est donc :

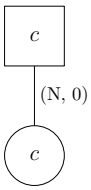

FIGURE 1 – Arbre pour un singleton.

— Pour $N = 3$, $P = 4$ et $D = \{(1, 2, 2), (3, 3, 1)\}$, de nombreux arbres sont possibles, dont les deux représentés ci-dessous :

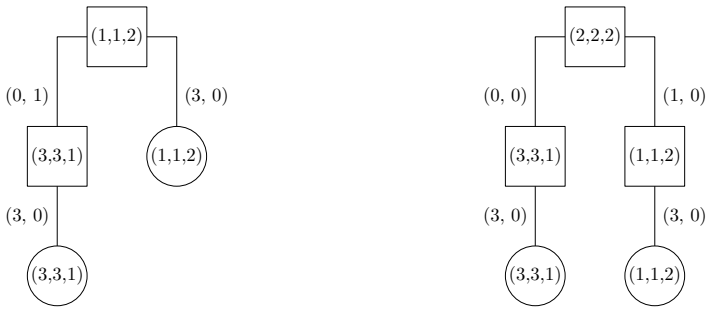

FIGURE 2 – Deux arbres possibles pour $D = \{(1, 2, 2), (3, 3, 1)\}$.

On notera en revanche qu'aucun D-arbre ne peut ici avoir de racine étiquetée par $c = (0, 0, 0)$: pour $r = (0, 0)$, on aurait $\mathit{filtre}(D, c, r) = D$.

— Pour $N = 2$ et $P = 3$, l'arbre de la figure 3 en page suivante est un C-arbre de stratégie (dans cet arbre, on a noté les couleurs a, b, c au lieu de 0, 1, 2 pour éviter les confusions entre combinaisons et réponses).

Question 9. On note n le nombre de nœuds internes et f le nombre de feuilles d'un D-arbre de stratégie T. Montrer que $f = |D|$ et $n \geq |D|$.

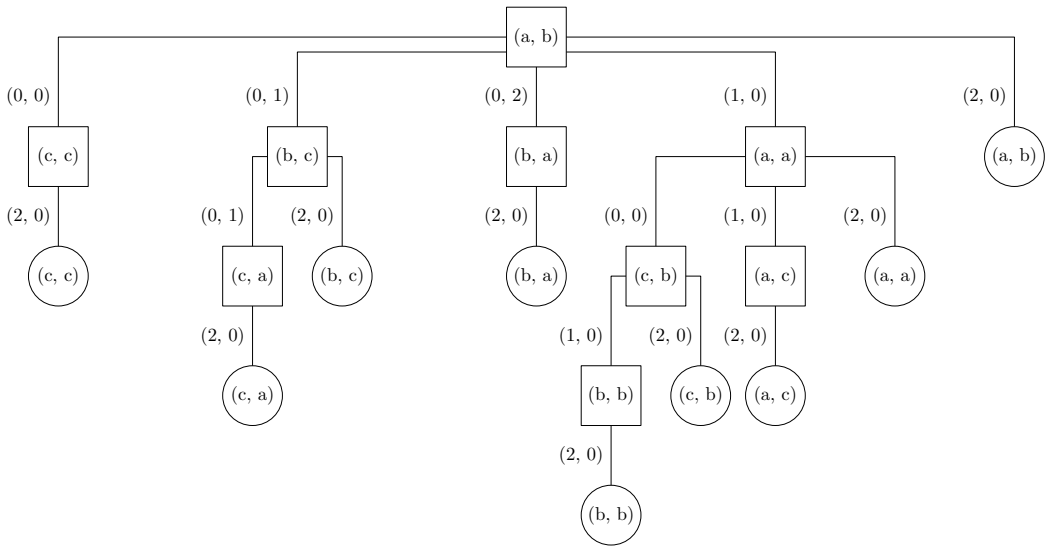

FIGURE 3 – L'arbre de stratégie T_0.

Jeu suivant un arbre de stratégie

— On dit qu'un historique $h = ((c_1, r_1), \ldots, (c_k, r_k))$ est *joué suivant l'arbre* T s'il existe un chemin partant de la racine et étiqueté $c_1, r_1, c_2, \ldots, c_k, r_k$ (en alternant étiquette des nœuds internes et étiquettes des arêtes). Notons que s'il existe, ce chemin est unique.

— Pour un D-arbre de stratégie T et une combinaison $x \in D$, il existe une unique feuille de T étiquetée x, et donc un unique chemin de la racine à cette feuille. L'historique associé à ce chemin (qui se termine par $(x, (N, 0))$) est appelé *partie jouée suivant* T *pour le but* x.

— On note alors $\mathit{score}(T, x)$ le score de cette partie.

Intuitivement, un D-arbre de stratégie indique à J_1 comment jouer, s'il sait déjà que le but appartient à D :

— la racine lui indique le premier coup à jouer ;

— il reçoit une réponse r et il sait maintenant que le but est dans $\mathit{filtre}(D, c, r)$;

— il descend dans le sous-arbre correspondant, qui est soit une feuille (si $r = (N, 0)$, et la partie est alors terminée), soit un $\mathit{filtre}(D, c, r)$-arbre qui lui indiquera comment jouer la suite de la partie.

Exemple On reprend l'arbre T_0 de la figure 3, et l'on suppose que $\mathit{but} = (c, a)$. La partie jouée suivant T_0 se déroule alors ainsi :

— le joueur J_1 joue (a, b) (le coup à la racine) ;

— comme $\mathit{sim}((a, b), \mathit{but}) = (0, 1)$, l'arbitre répond $(0, 1)$ et l'on descend dans la branche correspondante ;

— J_1 joue ensuite (b, c) ;

— l'arbitre répond $\mathit{sim}((b, c), \mathit{but}) = (0, 1)$, on descend dans cette branche ;

— J_1 joue (c, a), l'arbitre répond $(2, 0)$ et l'on descend dans cette branche ;

— on arrive sur une feuille, ce qui indique que la partie est terminée.

Tournez la page S.V.P.

II.2 Stratégie optimale

— Le *score dans le pire cas* $pire_D(T)$ d'un D-arbre de stratégie T est défini par :

$$pire_D(T) = \max_{x \in D} score(T, x).$$

— Le *poids total* $poids_D(T)$ d'un D-arbre de stratégie T est défini par :

$$poids_D(T) = \sum_{x \in D} score(T, x).$$

— On définit :

$$pire_{opt}(D) = \min\{pire_D(T) \mid T \in \mathcal{T}_D\}$$
$$poids_{opt}(D) = \min\{poids_D(T) \mid T \in \mathcal{T}_D\}$$

— Un D-arbre de stratégie est dit *optimal dans le pire cas* s'il vérifie $pire_D(s) = pire_{opt}(D)$, *optimal en moyenne* s'il vérifie $poids_D(s) = poids_{opt}(D)$.

Question 10. À quoi correspondent $pire_D(T)$ et $poids_D(T)$ sur un D-arbre de stratégie T ?

Question 11. Déterminer un C-arbre de stratégie optimal dans le pire cas pour $N = 2$ et $P = 3$, et justifier son caractère optimal. *On pourra modifier l'arbre T_0 de la figure 3.*

Question 12. Déterminer $pire_{opt}(D)$ et $poids_{opt}(D)$ pour D de cardinal 1 ou 2.

Question 13. Un arbre est dit *optimiste* si chaque nœud interne a un enfant feuille. On note $pire'_{opt}D$ le minimum des $pire_D(T)$ pour T un D-arbre de stratégie optimiste. A-t-on nécessairement $pire'_{opt}(D) = pire_{opt}(D)$?

II.3 Jeu à deux joueurs

On considère une variante à deux joueurs de notre jeu, où l'arbitre est remplacé par un joueur J_2 qui cherche à maximiser le score de la partie. Le k-ième tour de jeu consiste en un coup $c_k \in C$ du joueur J_1 et une réponse r_k du joueur J_2, et l'on note $h = ((c_1, r_1), \ldots, (c_{k-1}, r_{k-1}))$ l'historique des couples (coup, réponse) depuis le début de la partie (au premier tour, $k = 1$ et cette liste est donc vide).

— Le joueur J_1 choisit librement $c_k \in C$;
— le joueur J_2 choisit une réponse $r_k \in sim(compat(h), c_k)$;
— si $r_k = (N, 0)$, la partie se termine avec un score final de k, sinon on remplace h par $h, (c_k, r_k)$ et l'on passe au tour suivant.

Question 14. Dans le jeu à deux joueurs, à quoi correspond $pire_C(T)$ (où T est un C-arbre de stratégie) ? Même question pour $pire_{opt}(C)$.

II.4 Représentation informatique

On utilise dans cette partie **le langage OCaml**. On suppose disposer d'un type `combi` permettant de représenter une combinaison et d'un type `reponse` pour les réponses (on pourra tester l'égalité de deux réponses avec l'opérateur `=`). On suppose également disposer d'une fonction `sim` permettant de calculer la similarité entre deux combinaisons :

```
val sim : combi -> combi -> reponse
```

On représente un arbre de stratégie par un objet du type suivant :

```
type strat =
  | Gagne
  | Noeud of combi * ((reponse * strat) list)
```

On a choisi de ne pas étiqueter les feuilles (l'étiquette serait nécessairement la même que celle du nœud parent, donc redondante).

Question 15. Écrire une fonction `joue_un_joueur` prenant en entrée un D-arbre de stratégie s et un but x et renvoyant $score(s, x)$. On lèvera une exception si $x \notin D$.

```
val joue_un_joueur : strat -> combi -> int
```

Question 16. Écrire deux fonctions `pire` et `poids` prenant en entrée un D-arbre de stratégie s et renvoyant respectivement $pire_D(s)$ et $poids_D(s)$. On précisera leur complexité en fonction du nombre total n de nœuds de l'arbre et/ou de $|D|$ et l'on cherchera à écrire des versions raisonnablement efficaces.

```
val pire : strat -> int
val poids : strat -> int
```

II.5 Calcul naïf de $pire_{opt}(D)$ et $poids_{opt}(D)$

On définit $split(X)$ comme l'ensemble des $c \in C$ tels quel $|sim(X, c)| > 1$.

Question 17. Pour $X \subseteq C$ et $X \neq \emptyset$, on définit récursivement f par :

$$f(X) = \begin{cases} 1 & \text{si } |X| = 1 \\ 1 + \min_{c \in split(X)} \max_{r \in sim(X,c)} f(filtre(X, c, r)) & \text{sinon} \end{cases}$$

Montrer que f est bien définie et que $f(X) = pire_{opt}(X)$.

Question 18. Donner (sans justification) une définition similaire pour $poids_{opt}$.

On suppose que l'on dispose d'une fonction `reponses_possibles` prenant en entrée un ensemble X de combinaisons, sous forme d'une liste, ainsi qu'une combinaison c, et renvoyant l'ensemble $sim(X, c)$ sous forme d'une liste d'éléments distincts.

```
val reponses_possibles : combi list -> combi -> reponse list
```


Tournez la page S.V.P.

On suppose de plus que l'on dispose d'une constante `toutes_combis` de type `combi list` correspondant à l'ensemble C et d'une constante `tous_bons` de type `reponse` correspondant à la réponse $(N, 0)$ (autrement dit, `sim x y = tous_bons` si et seulement si les combinaisons `x` et `y` sont égales).

Question 19. Écrire une fonction `pire_opt_naif` prenant en entrée un ensemble $X \neq \emptyset$ de combinaisons sous forme de liste et renvoyant $pire_{opt}(X)$. On écrira la fonction la plus simple possible, sans attention particulière portée à l'efficacité.

```
val pire_opt_naif : combi list -> int
```

Question 20. En supposant toujours que l'on dispose de la fonction `reponses_possibles`, écrire une fonction `strategie_poids_naive` prenant en entrée un ensemble X de combinaisons et renvoyant l'arbre d'une stratégie optimale en moyenne pour X (c'est-à-dire de poids égal à $poids_{opt}(X)$).

```
val strategie_poids_naive : combi list -> strat
```

Partie III. Recherche efficace de stratégies

III.1 Premières optimisations

La recherche d'une stratégie optimale demande de calculer un très grand nombre de valeurs $sim(x, y)$ (y compris plusieurs fois pour le même couple (x, y)). Pour accélérer cette recherche, on décide donc de pré-calculer ces valeurs et de les stocker. Dans cette sous-partie, on utilise **le langage C** afin de pouvoir contrôler plus finement les représentations mémoire des objets.

Question 21. Justifier que $|R|$ (le nombre de réponses effectivement possibles) est inférieur ou égal à $\frac{N(N+3)}{2}$.

Question 22. Sachant que N et P seront tous les deux inférieurs ou égaux à 8, proposer deux définitions de type :

— une pour un type `combi_int_t` permettant de représenter un entier de $\{0, \ldots, |C| - 1\}$;

— l'autre pour un type `rep_int_t` permettant de représenter un entier de $\{0, \ldots, |R| - 1\}$.

On choisira des types de taille adaptée et l'on écrira explicitement les `typedef`.

On suppose définis un type `rep_couple_t` ainsi que trois constantes globales `N`, `P` et `NB_COMBIS` (égale à $|C|$) :

```
struct rep_couple_t {
  int bien;
  int mal;
};
typedef struct rep_couple_t rep_couple_t;
```

```
const int N = 4; // valeur purement indicative
const int P = 6; // idem
const int NB_COMBIS = ...; //
```

On suppose de plus que l'on dispose de deux fonctions

```
rep_couple_t similarite(int x[], int y[]);
rep_int_t rep_int_of_rep_couple(rep_couple_t couple);
```

La fonction `similarite` prend en entrée deux combinaisons (sous la forme de deux tableaux de N entiers de $\{0, \ldots, P-1\}$) et renvoie leur similarité sous la forme d'un couple ; on suppose qu'elle s'exécute en temps $\mathcal{O}(N+P)$.
La fonction `rep_int_of_rep_couple` prend un couple (b, m) (valide) et renvoie le code entier correspondant ; on suppose qu'elle s'exécute en temps constant.

Question 23. Écrire deux fonctions (réciproques l'une de l'autre) permettant la conversion entre la représentation « tableau » et la représentation « entier » d'une combinaison :

```
combi_int_t int_of_combi(int x[]);
int *combi_of_int(combi_int_t x);
```

Question 24. Proposer une méthode permettant d'obtenir une fonction `sim` prenant en entrée deux combinaisons (sous forme de `combi_int_t`) et renvoyant leur similarité (sous forme de `rep_int_t`) qui s'exécute en temps constant. On donnera le code de cette fonction, ainsi que celui des éventuelles autres fonctions utilisées, constantes définies, et code d'initialisation à exécuter avant le premier appel à `sim`.

```
rep_int_t sim(combi_int_t x, combi_int_t y);
```

Question 25. Quelle quantité de stockage votre solution utilise-t-elle ? Jusqu'à quelle valeur de $|C|$ cela vous paraît-il raisonnable, sur un ordinateur personnel « standard » ?

Important Dans toute la suite du sujet, on suppose qu'une technique similaire a été mise en œuvre en OCaml de manière à ne plus manipuler que des entiers. Ainsi, les deux types `combi` et `reponse` sont en fait définis ainsi :

```
type combi = int
type reponse = int
```

- On suppose de plus disposer de deux constantes globales `nb_combis` et `nb_reponses` correspondant respectivement à $|C|$ et $|R|$.
- Les combinaisons sont numérotées de 0 à `nb_combis - 1` et les réponses de 0 à `nb_reponses - 1`. On suppose de plus que la constante `tous_bons`, de type `reponse`, correspond au codage de la réponse $(N, 0)$.
- La constante `toutes_combis` est toujours disponible, et de type `combi list`.
- La fonction `sim` est toujours disponible (et utilise les nouveaux types) mais s'exécute à présent en temps constant.

Tournez la page S.V.P.

III.2 Stratégie gloutonne

Pour une partie $X \neq \emptyset$ de C et une combinaison c, on définit :

$$f(X, c) = \max_{x \in X} \big|\{y \in X \mid y \neq c \text{ et } \mathit{sim}(c, x) = \mathit{sim}(c, y)\}\big|.$$

Question 26. En supposant que $\mathit{compat}(h) = X$ (où h est l'historique actuel) et que la combinaison à deviner est x, que représente la quantité $\big|\{y \in X \mid y \neq c \text{ et } \mathit{sim}(c, x) = \mathit{sim}(c, y)\}\big|$?

Un D-arbre de stratégie est dit *glouton* si c'est une feuille ou si l'étiquette c de sa racine vérifie $f(D, c) = \min_{c' \in C} f(D, c')$ et si chacun de ses sous-arbres est glouton. On conviendra que, si plusieurs c réalisent ce minimum, on choisira le plus petit (ce qui permet d'assurer l'unicité de l'arbre glouton).

Question 27. Expliquer pourquoi cette stratégie est raisonnable, et pourquoi on peut la qualifier de *gloutonne*.

Question 28. On fait, dans cette question uniquement, l'hypothèse que $\mathit{pire}_{\mathit{opt}}(D)$ est une fonction croissante de $|D|$ (ce qui est en général faux). Que peut-on alors dire d'un arbre glouton ?

Question 29. Écrire une fonction `repartit` prenant en entrée un ensemble X de combinaisons (sous forme d'une liste) et une combinaison c et renvoyant une liste $u = [u_1; \ldots; u_k]$, de type `combi list list`, et telle que :
- les liste u_i sont non vides et disjointes ;
- l'union des éléments des listes u_i vaut $X \setminus \{c\}$;
- pour chaque u_i, il existe $r_i \neq (N, 0)$ tel que $v_i = \mathit{filtre}(X, c, r_i)$.

L'ordre des éléments, tant dans la liste u qu'à l'intérieur de chaque liste u_i, n'est pas spécifié. On demande une complexité en $\mathcal{O}(|X|)$, que l'on justifiera.

```
val repartit : combi list -> combi -> combi list list
```

Question 30. Écrire une fonction `choix_glouton` prenant en entrée un ensemble X de combinaisons ainsi qu'un ensemble *coups*, et renvoyant une combinaison $c \in \mathit{coups}$ telle que $f(X, c) = \min_{c' \in \mathit{coups}} f(X, c')$ (X et *coups* seront supposés non vides). Déterminer sa complexité.

```
val choix_glouton : combi list -> combi list -> combi
```

Question 31. Écrire une fonction `pire_glouton` prenant en entrée un ensemble X de combinaisons et renvoyant $\mathit{pire}_X(s_g)$, où s_g est l'arbre glouton pour X.

Question 32. Déterminer la complexité d'un appel à `pire_glouton` avec $X = C$ en fonction de $|C|$ et de $\mathit{pire}_C(s_g)$.

III.3 Majorations et minorations

Élagage $\alpha\beta$. On définit les deux fonction mutuellement récursives suivantes :

Algorithme 1 $\alpha\beta$.

```
fonction EVAL1(buts, α, β)
    si |buts| = 1 alors
        renvoyer 1
    h_min ← β
    pour c ∈ split(X) faire
        h ← EVAL2(buts, c, α − 1, h_min − 1)
        h_min ← min(h + 1, h_min)
        si h_min ≤ α alors
            renvoyer h_min
    renvoyer h_min

fonction EVAL2(buts, c, α, β)
    repartition = {(r, filtre(buts, c, r)) | r ∈ sim(buts, c)}
    h_max ← α
    pour (r, buts') ∈ repartition faire
        si r ≠ (N, 0) alors
            h ← EVAL1(buts', h_max, β)
            h_max ← max(h_max, h)
            si h_max ≥ β alors
                renvoyer h_max
    renvoyer h_max
```

Question 33. Donner la spécification de ces deux fonctions. Comment les utiliser pour calculer $pire_{opt}(C)$?

Question 34. Justifier rapidement leur correction. Expliquer l'intérêt de cette approche par rapport à celle de la partie II.5.

Question 35. Expliquer comment modifier ces fonctions pour calculer une stratégie optimale dans le pire cas pour le joueur J_1.

Question 36. Quelles modifications faudrait-il apporter aux fonctions EVAL1 et EVAL2 si l'on souhaitait calculer $poids_{opt}(C)$?

Tri des coups. Les boucles principales des fonctions EVAL1 et EVAL2 (*pour* $c \in C$ et *pour* $(r, buts') \in repartition$) s'effectuent pour l'instant dans un ordre non spécifié. On s'intéresse dans cette partie à l'impact que peut avoir le fait de parcourir ces deux ensembles dans un ordre spécifique.

Tournez la page S.V.P.

Question 37. Quel serait le pire ordre de parcours possible ? Comment se comporteraient alors les deux fonctions ? On ne demande pas de justification formelle.

Question 38. Est-il réaliste d'espérer parcourir les ensembles dans l'ordre optimal ? Proposer une manière simple (et ayant un coût raisonnable) d'obtenir un ordre satisfaisant pour la fonction Eval2.

Question 39. Proposer un ordre intéressant de traitement des coups pour Eval1.

Partie IV. Prise en compte des symétries

Intuitivement, il est clair qu'il n'est pas nécessaire d'étudier à la fois les coups initiaux $(0,0,0,0)$ et $(1,1,1,1)$ lorsqu'on recherche une stratégie optimale : rien ne différenciant *a priori* la couleur 0 de la couleur 1, toute stratégie commençant par l'un doit pouvoir être transformée en une stratégie commençant par l'autre sans modifier aucune des grandeurs qui nous intéressent. Le but de cette partie est de formaliser cette intuition, de l'étendre au-delà du premier coup et d'étudier comment la mettre en œuvre en pratique pour diminuer (massivement) le temps de calcul d'une stratégie optimale.

IV.1 Définitions et premières propriétés

- On appelle *permutation des positions* une permutation σ de l'ensemble $\{0,\dots,N-1\}$ des positions.
- On appelle *permutation des couleurs* une permutation τ de l'ensemble $\{0,\dots,P-1\}$ des couleurs.
- On dit qu'une application α de C dans C est une *transformation* s'il existe une permutation des positions σ et une permutation des couleurs τ telles que :

$$\forall x \in C,\ \alpha(x) = (\tau(x_{\sigma(0)}),\dots,\tau(x_{\sigma(N-1)})).$$

  On note alors $\alpha = (\sigma,\tau)$.
- On note $\mathcal{S}$ l'ensemble des transformations.

Exemple. Pour $N=4$, $P=6$, $x=(1,2,5,1)$, $\sigma=(2,1,3,0)$ (c'est-à-dire $\sigma(0)=2$, $\sigma(1)=1\dots$) et $\tau=(0,4,2,1,5,3)$, on obtient $(\sigma,\tau)(x)=(3,2,4,4)$.

Question 40. Montrer que si $\alpha,\beta \in \mathcal{S}$, alors :
- α est une bijection, et $\alpha^{-1} \in \mathcal{S}$;
- $\alpha \circ \beta \in \mathcal{S}$.

IV.2 Implémentation

Question 41. Discuter le coût en temps et en espace d'un pré-calcul permettant de disposer d'une fonction `applique` prenant en entrée une transformation α et une combinai-

son c et renvoyant $\alpha(c)$. Cette fonction devra s'exécuter en temps constant et avoir le type `transfo -> combi -> combi`, où l'on précisera le type `transfo` (on ne demande pas d'écrire le code du pré-calcul ni de la fonction `applique`).

On suppose dans la suite que l'on dispose d'une telle fonction, ainsi que d'une liste `toutes_transfos` contenant l'ensemble des transformations.

```
val applique : transfo -> combi -> combi
val toutes_transfos : transfo list
```

IV.3 Équivalence de coups

Une partie $\mathcal{T}$ de $\mathcal{S}$ est dite *close* si elle vérifie les trois conditions suivantes :

(i) $\mathrm{Id}_C \in \mathcal{T}$;

(ii) si $\alpha, \beta \in \mathcal{T}$, alors $\alpha \circ \beta \in \mathcal{T}$;

(iii) si $\alpha \in \mathcal{T}$, alors $\alpha^{-1} \in \mathcal{T}$.

Si $\mathcal{T}$ est close, on définit la relation $\sim_{\mathcal{T}}$ sur C par $x \sim_{\mathcal{T}} y$ s'il existe $\alpha \in \mathcal{T}$ tel que $y = \alpha(x)$.

Question 42. Montrer que si $\mathcal{T}$ est close, alors $\sim_{\mathcal{T}}$ est une relation d'équivalence.

Pour une combinaison x, on note :
— $[x]_{\mathcal{T}}$ la classe d'équivalence de x modulo la relation $\sim_{\mathcal{T}}$;
— $repr_{\mathcal{T}}(x)$ le plus petit élément de $[x]_{\mathcal{S}}$ pour l'ordre lexicographique (qu'on appelle *représentant de x modulo $\sim_{\mathcal{T}}$*).

On dit que x est $\mathcal{T}$*-canonique* si $x = repr_{\mathcal{T}}(x)$.

Question 43. Écrire une fonction `coups_canoniques` prenant en entrée un ensemble $\mathcal{T}$ (supposé clos) de transformations, sous forme de liste, et renvoyant la liste des coups $\mathcal{T}$-canoniques. On considérera que l'ordre sur les entiers pour le type `combi` coïncide avec l'ordre lexicographique sur les combinaisons. On demande une complexité en $\mathcal{O}(|C| + k|\mathcal{T}|)$, où k est le nombre de coups $\mathcal{T}$-canoniques, et l'on justifiera cette complexité.

```
val coups_canoniques : transfo list -> combi list
```

Question 44. Donner, sans justification, l'ensemble des combinaisons $\mathcal{S}$-canoniques dans le cas $N = 4$ et $P = 6$.

Question 45. Montrer que pour la recherche d'un C-arbre de stratégie optimal, on peut se limiter aux arbres dont l'étiquette à la racine est $\mathcal{S}$-canonique.

Tournez la page S.V.P.

IV.4 Extension aux coups suivants

Le gain apporté par cette réduction des coups à considérer est très important, mais il ne s'applique pour l'instant qu'au premier coup. On cherche à présent à étendre ce raisonnement aux coups suivants.

Pour $X \subseteq C$, on note $Fix(X)$ l'ensemble des transformations laissant X invariant :

$$\alpha \in Fix(X) \iff \alpha(X) = X$$

Question 46. Écrire une fonction `fix` prenant en entrée un ensemble X de combinaisons et renvoyant $Fix(X)$.

```
val fix : combi list -> transfo list
```

Question 47. Montrer que $Fix(X)$ est clos.

Question 48. En déduire une modification des fonctions Eval1 et/ou Eval2 permettant de restreindre les coups à considérer.

* *
*