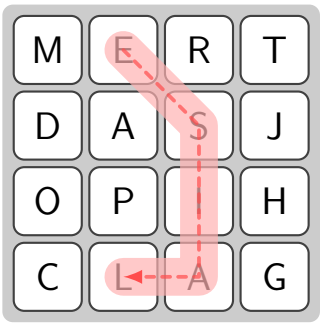
Boggle est un jeu de lettres déposé par Hasbro. Le jeu commence par le mélange d'un plateau (carré) de 16 dés à 6 faces, généralement en le secouant. Chaque dé possède une lettre différente sur chacune de ses faces. Les dés sont rangés sur le plateau 4 par 4, et seule leur face supérieure est visible. Après cette opération, un compte à rebours de 3 minutes est lancé et tous les joueurs commencent à jouer.



Chaque joueur cherche des mots pouvant être formés à partir de lettres adjacentes du plateau. Par *adjacentes*, il est sous-entendu horizontalement, verticalement ou en diagonale. Les mots doivent être de 3 lettres au minimum, peuvent être au singulier ou au pluriel, conjugués ou non, mais ne doivent pas utiliser plusieurs fois le même dé pour le même mot. Les joueurs écrivent tous les mots qu'ils ont trouvés sur leur feuille personnelle. Après les 3 minutes de recherche, les joueurs doivent arrêter d'écrire et le jeu entre dans la phase de calcul des points.

Taille du mot	3	4	5	6	7	8+
Points	1	1	2	3	5	11

Lors du calcul des points, chaque joueur lit à haute voix les mots trouvés. Un dictionnaire est utilisé pour vérifier la validité des propositions. Les mots trouvés par deux joueurs ou plus ne sont pas comptés. Les points sont attribués suivant la taille des mots trouvés. Le gagnant est le joueur ayant le plus grand nombre de points. *(d'après Wikipedia:* `http://fr.wikipedia.org/wiki/Boggle`)

★ Principe du projet

L'objectif de ce mini-projet est d'écrire un programme permettant à un joueur humain d'affronter un ordinateur. Il est malheureusement peu probable que l'humain parvienne à battre l'ordinateur.

Votre programme commencera par lire un fichier indiquant les lettres inscrites sur les faces de chaque dé. Ensuite, le programme réalisera un lancé des dés et positionnera ce jet sur le plateau de jeu.

C'est le joueur humain qui aura l'avantage de commencer la partie. Il devra saisir les mots qu'il trouve (un mot à la fois!). À chaque saisie, votre programme vérifiera que ce mot respecte la contrainte de longueur (au moins 3 caractères de long), que le mot n'a pas encore été proposé (un mot n'est comptabilisé qu'une seule fois, même s'il apparaît plusieurs fois sur le plateau), que le mot appartient bien au dictionnaire de mots connus et bien entendu qu'il est possible de former ce mot à partir des faces visibles des dés positionnés sur le plateau. Si tout ces tests sont valides alors le mot sera ajouté à la liste de mots trouvés par le joueur et le score du joueur sera crédité des points correspondants. Lorsque le joueur humain ne souhaite plus saisir de nouveau mot, il l'indique en saisissant un mot vide (uniquement un retour chariot).

C'est alors au tour du joueur artificiel de jouer. Votre programme va donc chercher tous les mots qui sont dans le dictionnaire et qu'il peut réaliser à partir du plateau. Chaque mot trouvé qui n'aurait pas été trouvé par le joueur humain est ajouté à la liste des mots trouvés par le joueur artificiel, et le score du joueur artificiel est crédité du nombre de points correspondants.

Les dés

Les lettres du Boggle sont placées de manière à ce que les lettres les plus communes soient tirées le plus souvent et de manière à ce que l'on ait une bonne combinaison entre le nombre de voyelles et de consonnes.

Pour recréer cette situation, votre programme devra être capable de lire un fichier dont le format est le suivant. Chaque ligne contient soit un commentaire si la ligne commence par un point virgule, soit la description d'un dé. La description d'un dé est un mot de 6 lettres indiquant les 6 lettres à placer sur les 6 faces du dés. Il y a exactement 16 descriptions de dé par fichier. Le contenu page suivante donne un exemple de fichier valide.

Lors de la phase d'initialisation, votre programme devra donc lire le fichier, créer les dés correspondants, générer une configuration d'un plateau en plaçant de manière aléatoire chaque dé sur le plateau. Par ailleurs, pour chaque dé, la face visible sera également tirée de manière aléatoire.

dices-definition.txt
```
; description des dés
BAJOQM
RALESC
LIBART
TOKUEN
ROFIAX
AVEZDN
NULEGY
MEDAPC
SUTELP
HEFSIE
ROMASI
GINEVT
RUEILW
RENISH
TIEAAO
DONEST
```

Une fois la phase d'initialisation terminée, vous êtes prêt à implémenter les deux types de recherches récursives: l'une pour le tour de jeu de l'utilisateur (recherche d'un mot spécifique), et l'autre pour le tour de jeu de l'ordinateur (recherche exhaustive de tous les mots).

Pour l'utilisateur, la méthode récursive cherche un mot spécifique sur le plateau et s'arrête dès que celui est trouvé. Pour l'ordinateur, la méthode récursive cherche tous les mots contenus dans le dictionnaire qui sont susceptibles d'être sur le plateau.

Vous pourriez être tenté d'unifier ces deux recherches récursives, mais c'est une très mauvaise idée. **Nous vous demandons explicitement d'implémenter séparément ces deux fonctions récursives**.

Le tour du joueur humain

Une fois que le plateau est affiché, le joueur peut saisir chaque mot qu'il trouve sur le plateau. Le joueur indiquera son souhait de ne plus saisir de mot en saisissant une simple ligne blanche (un simple retour chariot). Pour chaque mot proposé par le joueur, votre programme devra vérifier les conditions suivantes :

- le mot compte au moins trois lettres ;
- le mot est contenu dans le dictionnaire ;
- le mot apparaît sur le plateau (il est formé d'une séquence de lettres adjacentes et aucun dé n'est utilisé plusieurs fois) ;
- le mot n'est pas encore déjà contenu dans la liste de mots saisis par l'utilisateur.

Si l'une de ces conditions échoue, le programme doit en informer explicitement l'utilisateur et ne pas lui attribuer de points pour cette saisie. Par contre, si le mot satisfait toutes ces conditions, il doit être ajouté à la liste de mots saisis par l'utilisateur et le score de l'utilisateur doit être mis à jour en conséquence.

Le tour du joueur artificiel

Lors du tour du joueur artificiel, la tâche de l'ordinateur consiste à trouver tous les mots que le joueur humain aurait oublié, et ce, en cherchant de manière récursive tous les mots qui peuvent débuter par la lettre présente sur un dé du plateau. Lors de cette phase de jeu, les mêmes conditions sur les mots trouvés s'appliquent. Cependant, l'ordinateur n'est pas autorisé à comptabiliser les points des mots qui auraient déjà été trouvés par le joueur humain.

Comme dans tout algorithme de recherche exponentielle, il est important de limiter la recherche au maximum afin de s'assurer que la tâche sera achevée dans un temps raisonnable.

L'une des stratégies les plus importantes consiste à déterminer lorsque la recherche s'engage sur une voie morte afin de l'abandonner au plus vite. Par exemple, si la recherche a construit un chemin menant au préfixe `zx`, vous pouvez utiliser votre dictionnaire pour déterminer qu'il n'y a pas de mot qui commence par ce préfixe. Et donc, vous pouvez arrêter cette recherche pour continuer sur une voie plus prolifique. Si vous n'appliquez pas cette optimisation, votre ordinateur passera un temps non négligeable à vérifier la construction de mots qui n'existent pas tels que `zxgub` ou `zxaep`.

★ Étape 1 : Lecture de dés, affichage du plateau, mélange des dés.

Dans cette première étape, vous concevez et écrivez le code correspondant à un dé et au plateau de jeu.

Ce sujet incite à écrire du code découpé suivant la philosophie "orienté objet", même si l'on programme en C. Toutes les données seront encapsulées dans des structures opaques et accessible uniquement au travers de fonctions prenant un pointeur vers la structure en premier paramètre. Un exemple de cette approche se trouve sur le pense-bête du C[1], pour le module `point`.

Vous trouverez page suivante les diagrammes responsabilité/collaborateur (CRC) des classes à implémenter dans ce projet. Il est probable que la structure représentant votre classe ait au moins un champ par valeur à conserver dans le CRC, et que vous deviez implémenter une méthode par action permise par la classe. Les CRC sont un outil classique en pédagogie de la programmation orientée objet, détaillé dans l'article associé[2]

[1] Pense-bête du C: `https://mquinson.frama.io/ensr-arcsys1/refcard-c.pdf`

[2] K. Beck and W. Cunningham. A Laboratory for Teaching Object-Oriented Thinking, in Proceedings of OOPSLA'89. pp. 1–6, 1989. ACM Press. `http://doi.acm.org/10.1145/74877.74879`

▷ **Question 1:** Complétez les fichiers `dice.h` et `dice.c` pour implémenter la classe `dice_t`. Écrivez les fichiers `board.h` et `board.c`. Ajoutez les méthodes permettant d'afficher un dé et un plateau. Ajouter les méthodes nécessaires pour lire la définition des dés depuis un fichier. Ajouter les méthodes permettant de générer aléatoirement le plateau en mélangeant les dés.

Classe : `dice_t`	
Responsabilités : • Conserve la valeur (un caractère) de chacune des six faces du dés. • Conserve quelle est la face visible du dé. • Permet de consulter la face visible du dé. • Permet de calculer un nouveau lancer de dé (la face visible est donc mise à jour).	**Collaborateurs :** (aucun)

Classe : `board_t`	
Responsabilités : • Conserve la position de chacun des 16 dés. • Permet de mélanger les dés (la nouvelle position d'un dé est tirée aléatoirement, un nouveau lancer de ce dé est réalisé pour obtenir une nouvelle valeur). • Permet d'afficher l'état du plateau. • Un constructeur permet de lire un fichier et de créer les 16 dés qui y sont décrits.	**Collaborateurs :** • `dice_t`

On peut tirer un nombre aléatoire entre 1 et 6 avec la construction suivante : `rand() % 6 + 1`. Vous pouvez utiliser l'algorithme suivant pour mélanger des éléments dans un tableau à 2 dimensions.

Algorithme de permutation aléatoire des dés

```
for (int row = 0; row < num_rows; row++) {
    for (int col = 0; col < num_cols; col++) {
        int row_dest = rand() % num_rows;
        int col_dest = rand() % num_cols;
        swap(row, col, row_dest, col_dest); // change la place des deux dés
    }
}
```

★ Étape 2 : Le tour du joueur humain (sans recherche des mots sur le plateau).

Nous allons tout d'abord implémenter une classe représentant le dictionnaire. Son contenu sera chargé à partir d'un fichier qui contiendra sur chaque ligne un et un seul mot écrit en majuscules (non accentué). Un exemple de tel fichier est inclu dans l'archive fournie; on peut trouver d'autres dictionnaires intéressants dans le dépôt suivant:`https://gitlab.com/tube42/wordlists`

Classe : `lexicon_t`	
Responsabilités : • Permet de lire un fichier et d'ajouter les mots lus. • Permet d'ajouter un mot à la structure de données en mémoire, le fichier n'est jamais modifié. • Permet de supprimer un mot (en mémoire). • Permet de savoir si un mot est contenu dans le dictionnaire. • Permet de savoir combien de mots sont dans le dictionnaire.	**Collaborateurs :** Fonctions standards `gettext`, `malloc`, `realloc`.

▷ **Question 2:** Faites le schéma mémoire de la classe `lexicon_t` avant d'écrire la moindre ligne de C.

▷ **Question 3:** Implémentez maintenant la classe `lexicon_t`, et testez votre code. Vous rendrez les tests avec la version finale de votre projet.

▷ **Question 4:** Écrivez une fonction permettant à l'humain de jouer. Cette fonction aura le prototype suivant:

```
void play_human(board_t* board, char*** words, int* word_count);
```

Cette fonction alloue et retourne un tableau de tous les mots valides joués par l'humain. Elle s'utilise comme dans l'exemple page suivante. Elle est basée sur une boucle permettant à un joueur de saisir les mots qu'il trouve. Pour chaque mot, comptabilisez les points si toutes les conditions sont respectées (mot pas encore présent dans la liste, longueur minimale respectée, mot inclus dans le dictionnaire). Sinon, rejetez le explicitement. Ne faites pas de supposition quant au nombre de mots maximal que le joueur peut trouver, et utilisez `malloc` et `realloc`. Lors de cette étape, vous ne vous intéressez pas à savoir si le mot se trouve ou non sur le plateau.

Exemple d'usage de la fonction play_human

```
char** words;
int word_count;
play_human(board, &words, &word_count);
for (int i=0; i<word_count; i++) {
  printf(" Found %s\n", words[i]);
  free(words[i]);
}
free(words);
```

★ Étape 3 : Trouver un mot sur le plateau.

Écrivez la recherche récursive d'un mot sur le plateau afin de vérifier que le joueur humain ne triche pas. Rappelez-vous, un mot valide doit respecter deux règles : i) les lettres doivent être adjacentes, ii) un dé ne peut être utilisé qu'une et une seule fois dans un mot. N'oubliez pas que dés que vous réalisez que vous ne pouvez pas former un mot à partir de cette position, vous devez passer à la position suivante.

Classe : `board_t`

Responsabilités :

- Conserve la position de chacun des 16 dés.
- Permet de mélanger les dés (la nouvelle position d'un dé est tirée aléatoirement, un nouveau lancer de ce dé est réalisé pour obtenir une nouvelle valeur).
- Permet d'afficher l'état du plateau.
- Un constructeur permet de lire un fichier et de créer les 16 dés qui y sont décrits.
- **Permet de chercher (récursivement) si l'on peut construire à un mot donné à partir de l'état du plateau (en effectuant un parcours des dés).**

Collaborateurs :

- `dice_t`

▷ **Question 5:** Modifiez votre fonction `play_human` pour qu'elle n'autorise que des mots pouvant être construits à partir du plateau. Voici son nouveau prototype:

```
void play_human(board_t* board, lexicon_t* lexicon, char*** words, int* word_count);
```

★ Étape 4 : Trouver tous les mots sur un plateau (tour du joueur artificiel)

Il est temps d'implémenter la logique du joueur artificiel et notamment l'algorithme de recherche récursive couplé à un dictionnaire. Cette récursivité est une recherche exhaustive. Vous devez donc explorer toutes les positions du plateau à la recherche des mots qui sont dans le dictionnaire. Cette étape est la plus difficile. Réfléchissez bien à toutes les étapes de la recherche avant de développer votre solution. Pensez également aux optimisations que vous pouvez mettre en place une fois que votre recherche fonctionne. Utilisez un dictionnaire plus petit lors de votre développement et pourquoi pas un plateau contenant moins de dés.

Pour réaliser l'algorithme de recherche, plusieurs approches sont envisageables :

(a) Utiliser la méthode de recherche développée à l'étape 3 pour chacun des mots du dictionnaire. Cette approche est malheureusement trop longue en pratique avec un dictionnaire de taille réaliste.

(b) Écrire une méthode récursive qui génère toutes les combinaisons de lettres présentes sur le plateau, et pour chaque combinaison, regarder si celle-ci est contenu dans le dictionnaire. Malheureusement, il y a environ 40 000 milliards de milliards de combinaisons.

La solution est d'améliorer la seconde approche pour arrêter au plus tôt la génération d'une combinaison si celle-ci ne permet pas de générer des mots du dictionnaire. Autrement dit, à chaque génération d'une combinaison, si celle-ci ne peut pas servir de préfixe à un mot du dictionnaire, alors on coupe cette branche de la génération, et on remonte à l'étape précédente. Le pseudo-code correspondant est présenté page suivante:

```
1  void play_computer(board_t* board, char*** words, int* word_count) {
2
3    int used[16] = {0};   // Tableau de boolean indiquant si chaque dé est déjà pris
4    char sofar[17] = {0}; // Mot en cours de construction
5
6    // Fonction locale (peut être fait autrement dans une version finale :)
7    void recursive_search(int board_pos, int word_pos) {
8      pour tout next_pos parmi les voisins de board_pos {
9        if (used[next_pos] != 1) { // Le même dé ne peut servir qu'une fois
10         sofar[word_pos++] = lettre du plateau à la position next_pos
11
12         Si sofar ne constitue pas un préfixe de mot valide du dictionnaire {
13           sofar[word_pos--] = '\0'; // On efface cette lettre du mot
14           continue; // Tente avec la valeur suivante de next_pos
15         }
16         Si sofar constitue un mot valide du dictionnaire
17           Ajouter une copie du mot ``sofar'' à la liste
18
19         used[next_pos] = 1;
20         recursive_search(next_pos, word_pos);
21         used[next_pos] = 0;
22         sofar[word_pos--] = '\0';
23       }
24     }
25   }
26
27   // Corps de la fonction play_computer
28   for (int pos = 0; pos<16; pos++) { // Pour chaque position initiale possible
29
30     used[pos] = 1;
31     recursive_search(pos, 0); // Fonction ci-dessous
32     used[pos] = 0;
33   }
34 }
35 }
```

▷ **Question 6:** Pourquoi n'est-il pas nécessaire de remettre à zéro les tableaux `used` et `sofar` quand on envisage une nouvelle position initiale à la ligne 29? La réponse est triviale pour la première position initiale, mais qu'en est-il du cas `pos=1` ?

▷ **Question 7:** Modifiez votre classe `lexicon_t` afin de stocker les mots dans une structure de type *PATRICIA trie*, qui est un type de *radix tree*. Il n'est pas demandé d'implémenter cette structure de donnée vous-même, mais plutôt d'utiliser une version disponible en ligne, telle que `https://github.com/armon/libart` ou une autre à votre convenance.

Classe : `lexicon_t`

Responsabilités :

- Permet de lire un fichier et d'ajouter les mots lus.
- Permet d'ajouter un mot.
- Permet de supprimer un mot.
- Permet de savoir si un mot est contenu dans le dictionnaire.
- **Permet de savoir si une chaîne de caractère est un préfixe pour un ou plusieurs mots du dictionnaire.**
- Permet de savoir combien de mots sont dans le dictionnaire.

Collaborateurs :

Fonctions standards `gettext`, `malloc`, `realloc`.

`https://github.com/armon/libart` ou équivalent.

▷ **Question 8:** Modifiez votre boucle principale pour permettre à l'ordinateur de jouer à son tour.

★ Étape 5 : Finitions.

▷ **Question 9:** Modifiez votre boucle principale pour permettre de rejouer en fin de partie.

▷ **Question 10:** Vous pouvez maintenant peaufiner votre programme : vérifiez que vous avez bien géré toutes les saisies possibles et cherchez à optimiser les différentes routines.

★ Bonus (tâches facultatives mais bienvenues)

Esthétisme *(difficulté: faible)* : Améliorez l'aspect esthétique de votre application avec un peu d'ASCII Art et voire même quelque code d'échappement ANSI pour mettre de la couleur).

Jeu à deux *(difficulté: faible)* : Offrez la possibilité à deux joueurs humains de se confronter lors d'une partie. L'ordinateur indiquera les mots non trouvés à la fin de la partie.

Hall of Fame *(difficulté: faible)* : Votre programme sauvegardera et affichera les meilleurs scores (le nom des joueurs associés).

Double lettre *(difficulté: assez faible)*: Donnez la possibilité d'avoir deux lettres sur un même dé, par exemple pour le couple `QU`. Le couple présent sur le dé compte pour 2 lettres: le mot `quota` rapportera 2 points (mot de 5 lettres) mais est formé de 4 dés seulement sur le plateau de jeu.

Taille variable *(difficulté: assez faible)* : Modifier votre programme afin que les dimensions du plateau puissent être changées facilement, afin par exemple de pouvoir jouer sur une grille (42×42).

Extension 3D *(difficulté: moyenne)* : Modifier votre programme afin que le plateau ne soit plus uniquement un carré (4×4), mais un cube ($4 \times 4 \times 4$).

Limite de temps de jeu *(difficulté: moyenne)* : Ajouter la possibilité de limiter le temps de jeu de chaque joueur. Par exemple, le joueur humain n'a que 5 minutes pour trouver et proposer ses mots.

Interface graphique *(difficulté: élevée)* : Proposer une interface utilisateur graphique (par exemple en utilisant Allegro ou SDL2).

Wordament *(difficulté: moyenne à élevée)* : Ajoutez les différents mode de jeu proposés par l'application Wordament: dés offrant deux lettres au choix, percer les ballons, gemmes, ruée vers l'or. Le mode de base de ce jeu est plus relaxant pour l'humain, puisqu'il doit simplement marquer un certain nombre de points pour débloquer le niveau et passer au suivant (notez que les points ne sont pas comptés comme au boggle). Pour faire un bon générateur de niveaux intéressants, vous aurez probablement besoin d'un résolveur automatique, afin de calculer combien de points sont possibles sur un plateau donné.

Klingon Boggle *(difficulté: déraisonnable)* : `http://bigbangtheory.wikia.com/wiki/Klingon_Boggle`.

Imaginer vos propres extensions : Vous êtes bien entendu libres d'implémenter toute autre extension ou option à ce jeu.

★ Cas d'utilisation (à titre d'exemple)

```
 ______                                  __
|   __ \.-----..-----..-----.|  |.-----.
|      <|  _  ||  _  ||  _  ||  ||  -__|
|______/|_____||___  ||___  ||__||_____|
                |_____||_____|

Please enter your name: Sheldon Cooper

        /---\/---\/---\/---\
        | r || r || g || t |
        \---/\---/\---/\---/
        /---\/---\/---\/---\
        | s || h || m || i |
        \---/\---/\---/\---/
        /---\/---\/---\/---\
        | g || a || d || d |
        \---/\---/\---/\---/
        /---\/---\/---\/---\
        | e || t || s || q |
        \---/\---/\---/\---/

Player:Sheldon Cooper        Score:0
Please enter a word: tas
Player:Sheldon Cooper        Score:1
Please enter a word: gate
Player:Sheldon Cooper        Score:2
```

```
Please enter a word: mats
Player:Sheldon Cooper        Score:3
Please enter a word: bidule
Sorry, but you cannot make this word from the board.
Please enter a word: de
Sorry, but you must enter at least a 3-characters word.
Please enter a word: gate
Sorry, but this word is already in your list.
Please enter a word:
Congratulations Sheldon Cooper, your score is 3
The computer will now play...
I just found 'git', 'sas', 'sate', 'sage', 'samit', 'hast', 'haste', 'hate', 'mit', 'mas', 'mat', 'mats', 'mate',
 'mage', 'mahdi', 'image', 'gate', 'admit', 'age', 'ami', 'date', 'dam', 'dit', 'eta', 'team', 'tas', 'tag', 'tag
s', 'stage'.
Player:computer        Score:32
Bazinga!
```

★ Rendu et attendus

Ce projet est à faire seul ou en binôme. Chaque groupe doit rendre une archive tar compressée contenant les sources du programme, un rapport en pdf d'au plus cinq pages, et les fichiers de tests que vous aurez écrits. L'archive doit être envoyée à l'adresse `martin.quinson@ens-rennes.fr`. Tout votre programme doit être écrit en C. Si vous avez utilisé git comme conseillé, vous pouvez donner l'adresse de ce git à la place de l'archive (en vous assurant que nous y avons accès).

Vous porterez un soin particulier à l'écriture du code. *Pensez à nettoyer pour ne pas rendre un brouillon.* Le code doit être bien écrit et commenté pour être facilement lisible. Il est rappelé que l'on écrit un programme pour que d'autres humains puissent le lire, et (accidentellement seulement) pour que les machines puissent l'exécuter. Nous compilerons votre programme avec quelques `-W???` bien choisis pour une première vérification syntaxique. Nous le lirons également attentivement. Quelques conseils se trouvent sur wikipédia[3].

Votre rapport doit apporter une réponse claire et détaillée à chaque question du sujet, sans reprendre trop de code. La note finale tiendra compte à la fois de votre rapport et de votre code. Relisez-vous avant de envoyer votre travail! Votre rapport doit comporter (au moins) les parties suivantes :

Introduction : quelques mots pour présenter ce projet (ce qu'on va faire et pourquoi c'est intéressant).

Une réponse construite pour chaque question : expliquer vos observations, pourquoi ça se passe comme ça, quel est le mécanisme observé, à quoi il sert, pourquoi on se pose cette question... Soyez pédagogiques!

Synthèse : une conclusion sur vos observations, expérimentations et résultats, etc. Éventuellement, si vous en avez, des commentaires sur ce qui pourrait être amélioré pour l'année prochaine ou des idées de bonus que vous auriez voulu implémenter dans ce projet.

Bibliographie: Donnez la liste de toutes les sources (sites, livres ou individus) qui vous ont aidé, avec quelques mots de ce que vous en avez retiré. Attention, la frontière est mince entre *l'oubli* de certaines sources et le plagiat. N'oubliez rien, ne trichez pas.

Par tricher, nous entendons notamment :

- Rendre le travail de quelqu'un d'autre avec votre nom dessus ;
- Obtenir une réponse sur internet ou autre et mettre votre nom dessus. Changer le nom des variables et fonctions ou leur ordre avant de mettre votre nom dessus est considéré comme une tricherie agravée bien que cela ne suffise pas à tromper un système anti-plagiat comme MOSS[4];
- Permettre à un collègue de *s'inspirer* de votre travail. Assurez vous que votre dépot est privé.

En cas de litige grave, seul un historique progressif de vos travaux (comme en offre git) constitue une preuve de votre innocence. *Commit soon, commit often.*

En revanche, il est possible (voire conseillé) de discuter du projet et d'échanger des idées avec vos collègues. Mais vous ne pouvez rendre que du code écrit par vous-même, et vous devez détailler brievement l'intégralité de vos sources inspirations dans la partie bibliographie de votre rapport. La ligne jaune à ne pas dépasser est la lecture du code : on peut discuter, et même faire des schémas au tableaux, mais **vous ne devez jamais lire du code écrit par un autre groupe. Sous aucun prétexte.**

Votre travail est à rendre pour le **vendredi 20 décembre 2019 avant 19h CEST**.

[3]Notions basiques sur la lisibilité d'un code C : `http://fr.wikibooks.org/wiki/Conseils_de_codage_en_C/Lisibilit%E9_des_sources`.

[4]MOSS, `http://theory.stanford.edu/~aiken/moss/`