Sujet 5:

Création de fichiers de tests.

(durée max: 2h)

Au fur et à mesure du développement de votre projet, il est important de réaliser des tests.

- Des **tests unitaires** après l'écriture de chaque fonctionnalité importante qui vont tester une ***fonctionnalité "unitaire"***.
- Des **tests d'intégrations** après un certain nombre d'étapes qui vont tester un ***comportement global***.

Nous avons déjà vu comment réaliser des tests au niveau du code et des fonctions dans les TP précédents. Dans cette partie, nous allons tester le comportement du programme au complet en envoyant une entrée textuelle d'un ou plusieurs déplacement, et en observant la sortie attendue.

➔ **Observez** le fichier `etat_courant.txt` qui encodent l'état du jeu après chaque déplacement. L'analyse de ce fichier permet de définir si un coup c'est correctement passé ou non en comparant celui-ci à une sortie attendue.

Mise en place des tests de l'application:

L'application permet de déplacer les pièces de manière cohérente vis à vis d'une partie.

Par exemple, la combinaison

```
> i
> n 1 0 ^
> n 1 4 v
> d 1 0 ^ ^
> fin
```

Doit aboutir à un état de jeu **valide** et connue (un éléphant et un rhinocéros sur le jeu, l'éléphant est déplacé d'une case).

Au contraire, la combinaison:

```
> i
> n 2 3 >
> fin
```

Doit aboutir à la détection d'un coup **invalide** (une pièce ne peut pas être introduite au milieu du jeu).

Notez que ces combinaisons peuvent être écrites **dès maintenant** même pour des mouvements non encore implémentés. Dans un code bien construit, ces tests de l'application ne dépendent pas du codage interne de la structure de données utilisée pour manipuler l'échiquier et son écrit avant

même d'avoir codé l'implémentation des fonctions.
Ces tests doivent donc pouvoir être exécutés, peu importe le code interne.
Par exemple, les tests des binômes voisins peuvent être par la suite appliqués sur votre code.

Exemple de fonctionnement:

Nous détaillons le fichier de test donné en exemple `test_unitaire_ok_01`:
Ce test tente d'introduire un éléphant sur le plateau.
La première ligne consiste à initialiser le jeu (dans les autres exemples, on charge un fichier à partir d'un état de jeu donné).
La seconde ligne demande l'introduction d'un éléphant en (1,0) en direction haute.
Cette introduction doit normalement être valide, et la troisième ligne demande la fin du programme.

Note:
Il s'agit bien d'un test unitaire comportant le test d'une seule action (introduction de la pièce).

Pour lancer le test, on appelle en ligne de commande (en supposant que l'on soit dans le répertoire bin/)

```
./[nom_excutable] < ../test/test_unitaire_ok/test_unitaire_ok_01.txt
```

Note: Le sigle "<" indique la redirection de l'entrée standard. C'est à dire que le programme ne va plus lire son entrée à partir du clavier, mais viens directement la lire dans le fichier indiqué.

L'analyse du résultat du test peut se faire de manière visuelle en observant le fichier de sortie `etat_courant.txt` ou en observant le résultat à l'aide du visualiseur.
Il est cependant commode d'automatiser le processus d'analyse lorsqu'on relance les tests à plusieurs reprises.

Un second fichier est alors utile dans ce cas, le fichier `test_unitaire_ok_01_sortie.txt` indique la sortie attendue de l'état du jeu après exécution du test.
Il est possible de demander la comparaison ligne à ligne entre ce fichier de sortie attendu et le fichier `etat_courant.txt` actuel.

Un script réalise cette comparaison automatiquement en parcourant tout les tests les un derrière les autres, et indique si une erreur est détectée par rapport à la sortie attendue.
Pour cela, déplacez vous dans le repertoire `script_test_automatique/` et lancez la commande:

```
$ python run_test.py
```

(ou `python3.x run_test.py` en fonction de la version de Python installée)

La sortie vous indique pour chaque test si la sortie attendue est bien la bonne.

Note: Cette démarche fonctionne également pour les tests dits de "ko". Dans ce cas, la sortie attendue ne doit pas prendre en compte le déplacement qui doit échouer.

Travail demandé:

Il vous est demandé d'établir la série de tests qui viendra **valider** que votre projet réalise bien la fonctionnalité d'un jeu d'échec en écrivant les fichiers de déplacements, ainsi que les fichiers de sorties attendues (et potentiellement les fichiers d'entrées si nécessaire).
Pour cela, vous allez écrire un ensemble de tests **unitaires** et **d'intégrations**.
Ils devront valider à la fois les coups **possibles** (à placer dans `test_unitaire_ok`), et également confirmer que les coups **invalides** (à placer dans `test_unitaire_ko`) sont bien détectés comme tels.

Vos tests devront couvrir un **maximum de cas possibles** et respecter la **structure** décrite dans l'annexe concernant les fichiers de tests.
Réfléchissez à une **stratégie** adéquate.

Vous garderez ces tests tout au long de votre projet. Ils vous permettront de valider votre projet et vous aideront à ne pas oublier de cas particuliers lorsque vous coderez. Si ces tests sont insuffisants, ils ne pourront pas valider clairement votre travail.

Quelques conseils:

- Ne passez pas l'ensemble de vos tests à observer l'introduction d'un éléphant à différents endroits de l'échiquier. Mais **couvrez** au mieux l'ensemble des actions possibles.

- Réaliser au minimum **2 tests** de validité par action (introduction, deplacement, changement orientation).
  (ex. déplacement avec changement d'orientation, déplacement nécessitant une poussée, déplacement hors du cadre du plateau, etc)

- Pour les tests **unitaires** (ok et ko), réalisez une **unique action** par test. Un test unitaire ne devrait ainsi contenir idéalement qu'une seule ligne d'action (en plus du chargement et de la terminaison du jeu).
  (ex. Ne testez pas en même temps un déplacement d'éléphant et de rhinocéros, ne testez pas l'un après l'autre deux coups invalide puisque seul le premier passera à l'exécution, …)

- Réalisez quelques tests **d'intégrations** aboutissant à une configuration de plateau plus complexe.

- Pensez bien aux cas de coups **invalides**, aux cas particuliers, aux cas nécessitant des poussées, etc.
  Établir les tests des cas particuliers dès maintenant permet de ne pas les oublier plus tard au moment du développement de code.

- Utilisez les **outils** à disposition pour simplifier la mise en scène des tests: *lecture de fichiers de scènes, initialisation*.

- Tout comme le code, vos tests doivent être commentés et satisfaire aux conditions de rendus.

Travail *minimum vitale* demandé (rapporte la moyenne):

- Au minimum 2 tests valides ainsi que 2 tests invalides (avec un fichier d'entrée, et un fichier de sortie attendue) par type d'action.
- Au minimum 2 tests d'intégrations vérifiant que l'agencement de plusieurs coups fonctionnent correctement.

Attention: Tous vos fichiers de tests devront respecter l'organisation et la syntaxe décrite dans l'annexe sur les fichiers de tests.

Note: Vos scripts de tests seront notés. La notation tiendra notamment compte de l’exhaustivité de vos tests, de la précision des tests unitaires, de vos tests d'intégrations, des commentaires.

Ecriture et lectures de fichiers:

(durée max: 2h)

Etat du jeu:

A chaque tour de jeu, l'état du jeu est écrit textuellement dans un un fichier sur le disque dur (fichier `etat_courant.txt`).
La gestion des écritures/lectures dans un fichier est regroupée dans les fichiers `entree_sortie.c` et `entree_sortie.h`.

Les fonctions `entree_sortie_ecrire_plateau_pointeur_fichier` et `entree_sortie_lire_jeu_fichier` sont écrites de manière non lisible et doivent être ré-écrite.

Rappel:

L'ouverture d'un fichier se réalise par la commande `fopen`.
La fermeture d'un fichier se réalise par la commande `fclose`.
L'écriture textuelle dans un fichier se réalise par la commande `fprintf`.
La syntaxe de `fprintf` est similaire à la syntaxe de `printf` mise à part que le premier paramètre est le descripteur de fichier retournée par `fopen`.

A titre d'exemple, voici un code permettant d'écrire dans un fichier.

```
int main()
{
    const char *filename="mon_nom_de_fichier.txt";

    FILE *fid=NULL; //struct contenant un descripteur de fichier
    fid=fopen(filename,"w"); //ouverture du fichier “filename” en mode ecriture

    if(fid==NULL)
    {
        printf("Erreur ouverture du fichier %s\n",filename); exit(1);
    }

    fprintf(fid,"J'ecris dans un fichier le nombre %d \n",3);
    fprintf(fid,"Et je sais que 2+2=%d",2+2);
    fprintf(fid,"Enfin si j'ai une piece de type tour, j'ecrirai le nombre
%d\n",tour);

    int c=fclose(fid);
    if(c!=0)
    {printf("Erreur fermeture fichier %s\n",filename);exit(1);}
}
```

Notez que la fonction `entree_sortie_ecrire_plateau_pointeur_fichier` `reçoit` directement en paramètre un pointeur de fichier. Cela permet d'écrire à la fois dans un fichier, mais également de permettre l'affichage sur la ligne de commande en fonction du pointeur utilisé.

Vous devez dans la suite, réécrire la fonction `entree_sortie_ecrire_plateau_pointeur_fichier` de manière lisible. Son comportement devra être similaire à la fonction illisible originale, prenez soin de bien vous inspirer des fichiers textes écrits par ces fonctions.

- ➔ **Implémentez** l'écriture de l'état du jeu dans un fichier.
- ➔ **Testez** votre code sur différents états de l'échiquier.
- ➔ **Assurez** vous que votre fichier d'état soit bien identiques aux fichiers écrits par la fonction illisible (même ordonnancement, etc).

Note: Pour vous assurer que deux fichiers sont identiques en tout point, la commande `diff fichier_1 fichier_2` permet d'afficher toutes différences existantes.

Vous devrez vous assurer que le fichier écrit par votre fonction est en tout point identique au fichier écrit par la fonction non lisible.

- ➔ **Implémentez** la lecture d'un jeu à partir d'un fichier (cette fois, l'argument reçu est un chemin vers le fichier à lire).
- ➔ **Testez** votre code sur différents fichier de jeux.

Travail en autonomie

(durée estimée: 3h):

- ➔ Fin de la séance.
- ➔ Débutez l'écriture du rapport (référez vous à l'annexe décrivant pour vous aidez sur la structure attendue).