

1 Introduction

Bulls and Cows est un jeu où s'affrontent deux joueurs. Au début, chaque joueur choisit une combinaison secrète de quatre chiffres entre 0 et 9.



Dans la variante que nous étudierons, la répétition d'un même chiffre dans une combinaison est autorisée. Lors d'un tour, chaque joueur essaie de deviner la combinaison de son adversaire en lui faisant une proposition. Celui-ci l'examine et lui transmet deux indices :

- le premier nommé *cows* représente le nombre de chiffres identiques et à la même place dans sa configuration secrète et dans la proposition
- le second nommé *bulls* représente le nombre de chiffres présents dans sa configuration secrète et dans la proposition mais pas à la même place

Chaque joueur peut noter les indices successifs sur une feuille et le gagnant est le premier à trouver la configuration secrète de son adversaire. Ce mini-projet va se décliner en deux parties :

- Programmer une partie en mode console où un joueur humain doit deviner une configuration secrète choisie aléatoirement par l'ordinateur.
- Programmer une partie en mode console où l'ordinateur doit deviner à l'aide d'un algorithme de résolution, une configuration secrète choisie aléatoirement. Dans cette partie on va aussi conjecturer statistiquement à travers des expériences sur un très grand nombre de parties des propriétés de l'algorithme de résolution : nombre moyen de tours nécessaires pour deviner la combinaison secrète, nombre maximal de tours ...

Un prolongement de ce mini-projet sera la réalisation d'une interface graphique pour une partie où un joueur humain doit deviner une configuration secrète (dans un prochain Mini-Projet). On donne ci-dessous un exemple d'interface graphique. La colonne *Guess* contient les propositions successives et la colonne *Result* contient les indices successifs. Par exemple 1A0B signifie que la proposition comportait 1 *bulls* (chiffre bien placé) et 0 *cows* (chiffre mal placé).



2 Cahier des charges

- Les programmes de chaque partie doivent être rassemblés dans des fichiers différents nommés *partieA.py* et *partieB.py*.

2. Chaque fonction de `partieA.py` et `partieB.py` doit passer le test inclus correspondant dans les programmes de tests `test-partieA.py` et `test-partieB.py` fournis sinon cela doit être mentionné dans le code source sous forme de commentaires.
3. Les réponses aux questions qui ne nécessitent pas de code doivent être fournies dans le code source sous forme de commentaire en les préfixant par le numéro de la question.
4. Chaque fonction doit être documentée avec une docstring.
5. Les parties les moins évidentes du code doivent être commentées de façon pertinente.
6. Un code client permettant de tester le programme ou les fonctions principales doit être fourni.

3 Partie A : le joueur est un humain

1. Créer un répertoire `BullsCows_Eleve1_Eleve2_Eleve3` avec les noms des membres du groupe séparés par des tirets bas (surtout pas d'espace). Par la suite on appellera ce répertoire le répertoire de base du projet.
2. Copier dans le répertoire de base le programme de tests `test-partieA.py` fourni.
3. Ouvrir un éditeur Python et créer dans le répertoire de base un fichier `partieA.py` avec l'en-tête ci-dessous :

```
1 import random
```

La première ligne permet d'importer le module (ou bibliothèque) `random`.

Exécuter le programme et afficher dans la console la documentation de la fonction `randint` avec `help(random.randint)`

Quelle expression permet de simuler le choix aléatoire d'un entier entre 0 et 9?

4. a. Ajouter à `partieA.py` le code complété de la fonction `combinaison_secrete()` en respectant la spécification donnée dans la docstring.

```
1 def combinaison_secrete():
2     """Renvoie un tableau de 4 entiers choisis
3     aléatoirement entre 0 et 9"""
4     return .....
```

- b. Vérifier si la fonction passe le jeu de tests en exécutant `test_combinaison_secrete(10000)` dans `test-partieA.py` (décommenter l'instruction). Si les tests sont passés, un message "Tests réussis pour `combinaison_secrete`" devrait s'afficher.

Méthode

Une assertion est une instruction qui vérifie si une condition (à valeur booléenne) est vérifiée dans l'état courant du programme.

L'exécution d'une assertion est silencieuse si elle est vérifiée et elle lève une exception de type `AssertionError` qui interrompt l'exécution sinon.

Les conditions vérifiées au début du programme sont appelées **préconditions**, à la fin du programme ce sont des **postconditions**.

Les postconditions sont souvent rassemblées dans des jeux de tests.

La syntaxe est `assert condition`.

```
>>> assert 1 == 1
>>> assert 1 == 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>> def test():
...     assert 1 == 1
...     assert 1 == 2
...
>>> test()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in test
AssertionError
```

5. Par la suite, on appelle configuration un tableau de quatre entiers entre 0 et 9 représentant une combinaison choisie ou proposée par un joueur au cours du jeu.

a. Ajouter à `partieA.py` le code complété de la fonction `nombre_bien_places()` en respectant la spécification donnée dans la docstring.

```
1
2 def nombre_bien_places(configuration1, configuration2):
3     """Renvoie le nombre de chiffres identiques et
4     en même position pour deux configurations"""
5     assert len(configuration1) == len(configuration2)
6     n = 0
7     .....
8     .....
9     .....
10    return n
```

b. Vérifier si la fonction passe le jeu de tests en exécutant `test_nombre_bien_places()` dans `test-partieA.py` (décommenter l'instruction).

6. a. Ajouter à `partieA.py` le code complété de la fonction `minimum(a, b)` en respectant la spécification donnée dans la docstring.

```
1 def minimum(a, b):
2     """Renvoie le minimum de deux nombres"""
3     .....
4     .....
5     .....
```

b. Vérifier si la fonction passe le jeu de tests en exécutant `test_minimum()` dans `test-partieA.py` (décommenter l'instruction).

7. a. Ajouter à `partieA.py` le code complété de la fonction `histo()` en respectant la spécification donnée dans la docstring.

```
1 def histo(configuration):
2     """Renvoie un histogramme du nombre d'occurrences
3     de chaque chiffre entre 0 et 9 dans un tableau configuration"""
4     h = [0] * 10
5     for c in configuration:
6         .....
7     return h
```

- b. Compléter le code de la fonction `test_histo()` dans `test-partieA.py` avec des assertions permettant de réaliser des tests pertinents de la fonction `histo`.

```
1 def test_histo():
2     assert histo([3,6,3,1]) == [0,1,0,2,0,0,1,0,0,0]
3     #à compléter avec des assert
4     print('Tests réussis pour histo')
```

8. a. Ajouter à `partieA.py` le code complété de la fonction `nombre_communs(configuration1, configuration2)` en respectant la spécification donnée dans la docstring.

```
1 def nombre_communs(configuration1, configuration2):
2     """Renvoie le nombre de chiffres communs pour les deux
3     configurations passées en paramètre"""
4     assert len(configuration1) == len(configuration2)
5     h1 = histo(configuration1)
6     h2 = histo(configuration2)
7     n = 0
8     for i in range(10):
9         .....
10    return n
```

- b. Compléter le code de la fonction `test_nombre_communs()` dans `test-partieA.py` avec des assertions permettant de réaliser des tests pertinents de la fonction `nombre_communs`.

```
1 def test_nombre_communs():
2     assert nombre_communs([5,5,5,5], [5,5,5,5]) == 4
3     assert nombre_communs([4,4,4,4], [3,3,3,3]) == 0
4     assert nombre_communs([4,6,4,1], [4,4,4,6]) == 3
5     #à compléter avec des assert
6     print('Tests réussis pour nombre_communs')
```

9. a. Ajouter à `partieA.py` le code complété de la fonction `nombre_bulls_cows(configuration1, configuration2)` en respectant la spécification donnée dans la docstring.

```
1 def nombre_bulls_cows(configuration1, configuration2):
2     """Renvoie un tableau [nombre de bulls, nombre de vaches]
3     pour [nombre de chiffres bien placés, nombre de chiffres
4     communs mais mal placés] pour les configurations passées
5     en paramètres"""
6     .....
```

- b. Compléter le code de la fonction `test_nombre_bulls_cows()` dans `test-partieA.py` avec des assertions permettant de réaliser des tests pertinents de la fonction `nombre_bulls_cows`.

```

1 def test_nombre_bulls_cows():
2     assert nombre_bulls_cows([5,5,5,5], [5,5,5,5]) == [4,0]
3     assert nombre_bulls_cows([4,4,4,4], [3,3,3,3]) == [0,0]
4     assert nombre_bulls_cows([4,2,4,2], [2,4,2,4]) == [0,4]
5     #à compléter avec des assert
6     print('Tests réussis pour nombre_bulls_cows')
```

10. a. Ajouter à `partieA.py` le code complété de la fonction `conversion_saisie(chaine)` en respectant la spécification donnée dans la docstring. On a intégré des préconditions dans la fonction pour vérifier si la chaîne peut être convertie en tableau d'entiers.

```

1 def conversion_saisie(chaine):
2     """Convertit une chaîne de caractères
3     en tableau d'entiers"""
4     #Précondition 1 : chaine de longueur 4
5     assert len(chaine) == 4
6     #Précondition 2: tous les caractères de chaine sont des chiffres
7     for c in chaine:
8         assert c.isdigit()
9     #renvoi d'un tableau d'entiers défini par compréhension
10    return .....
```

- b. Ajouter à `partieA.py` le code complété de la fonction `partie_humain_vs_ordinateur(nbmax_essai, debug = False)` en respectant la spécification donnée dans la docstring.



Cette fonction comporte un paramètre nommé, c'est une façon de définir des valeurs par défaut pour des paramètres. Les paramètres nommés doivent toujours être définis après les autres paramètres. Le paramètre `debug` permet d'afficher un traçage de l'exécution s'il est positionné à `True`.

```

1 def partie_joueur_humain(nbmax_essai, debug = False):
2     """Lance une partie de Bulls and cows : l'utilisateur doit
3     deviner la combinaison secrète en au plus nbmax_essai
4     essais. Ses propositions sont des chaines de caractères
5     comme '1234' et sont converties en tableaux d'entiers.
6     """
7     secret = combinaison_secrete()
8     essai = 0
9     bulls = 0
10    if debug:
11        print("Secret : ", secret)
12    while .....:
13        saisie = input("Proposition sous la forme '1234' : ")
14        .....
15        if debug:
16            print('essai=', essai, ' bulls=', bulls, ' cows=', cows)
17            print('-' * 80)
```

```
17     if .....:
18         return "Secret trouvé en " + str(essai) + " essais"
19     else:
20         return "Secret pas trouvé en " + str(nbmax_essai) + " essais"
```

On donne ci-dessous un exemple de partie :

```
In [51]: 1 partie_joueur_humain(15, debug = True)

Secret :  [3, 2, 7, 9]
Proposition sous la forme '1234' : 2369
essai : 1  bulls : 1 cows : 2
-----
Proposition sous la forme '1234' : 3259
essai : 2  bulls : 3 cows : 0
-----
Proposition sous la forme '1234' : 3279
essai : 3  bulls : 4 cows : 0
-----

Out[51]: 'Secret trouvé en 3 essais'
```

- c. Ajouter à `partieA.py` le code client ci-dessous qui est exécuté uniquement si `partieA.py` n'est pas importé sous forme de module dans un autre programme Python.

```
1  ## Code client exécuté uniquement si le programme n'est pas importé
2  if __name__ == '__main__':
3      partie_joueur_humain(15)
```

4 Partie B : le joueur est l'ordinateur

L'objectif de cette partie est d'implémenter un algorithme qui détermine une combinaison secrète. L'algorithme va utiliser deux dimensions physiques de l'ordinateur : l'espace avec sa mémoire et le temps avec sa capacité à effectuer très rapidement des calculs. L'utilisation de l'espace et du temps seront d'ailleurs les critères que nous utiliserons pour comparer les performances d'algorithmes qui résolvent un même problème.



Définition 1

Un algorithme est une séquence d'instructions permettant de résoudre un problème et qui peut être exécuté par une machine, par exemple un ordinateur.

Pour cela, il faut traduire l'algorithme dans un langage compris par une machine. Un ordinateur manipulant l'information uniquement sous forme binaire (0 ou 1), il existe des langages intermédiaires entre un langage humain dans un lequel est formulé un algorithme et le langage machine : ce sont des langages de programmation.

Notre algorithme de résolution peut être qualifié de *brute force* car il explore tout l'espace des possibles jusqu'à trouver la combinaison secrète :

- L'ordinateur mémorise d'abord toutes les combinaisons possibles de 4 chiffres entre 0 et 9 dans un tableau à deux dimensions (ou liste de listes en Python).
- L'index de parcours du tableau de toutes les combinaisons est initialisé à 0.

- Lors de chaque tour, l'ordinateur explore le tableau de toutes les combinaisons à partir de l'index courant, jusqu'à ce qu'il trouve une combinaison compatible avec toutes les propositions déjà effectuées : c'est-à-dire que la comparaison de la combinaison candidate avec chaque proposition doit donner le même nombre de chiffres bien placés (bulls) et de chiffres mal placés (cows) que la comparaison de cette proposition avec la combinaison secrète. L'ordinateur fait alors sa nouvelle proposition, reçoit la réponse (bulls, cows) pour la comparaison de sa proposition avec la combinaison secrète et fait pointer l'index de parcours sur la combinaison suivante dans le tableau de toutes les combinaisons possibles.
- La partie se termine lorsque le nombre d'essais maximal est atteint ou lorsque la combinaison secrète a été devinée, ce qui se produit presque sûrement en moins de 15 essais comme on va le vérifier expérimentalement.

On donne ci-dessous un exemple de partie :

```
Secret : [6, 0, 4, 0]
Proposition ordi : [0, 0, 0, 0]
essai= 1 bulls= 2 cows= 0
```

```
Proposition ordi : [0, 0, 1, 1]
essai= 2 bulls= 1 cows= 1
```

```
Proposition ordi : [0, 2, 0, 2]
essai= 3 bulls= 0 cows= 2
```

```
Proposition ordi : [3, 0, 3, 0]
essai= 4 bulls= 2 cows= 0
```

```
Proposition ordi : [4, 0, 4, 0]
essai= 5 bulls= 3 cows= 0
```

```
Proposition ordi : [4, 0, 5, 0]
essai= 6 bulls= 2 cows= 1
```

```
Proposition ordi : [6, 0, 4, 0]
essai= 7 bulls= 4 cows= 0
```

1. Copier dans le répertoire de base le programme de tests `test-partieB.py` fourni.
2. Ouvrir un éditeur Python et créer dans le répertoire de base un fichier `partieB.py` avec l'en-tête ci-dessous :

```
1 import random
2 from partieA import *
```

```
3
4 def toutes_combinaisons():
5     """Retourne un tableau de toutes les combinaisons possibles
6     de 4 chiffres entre 0 et 9"""
7     tab = []
8     for a in range(10):
9         for b in range(10):
10            for c in range(10):
11                for d in range(10):
12                    tab.append([a,b,c,d])
13     return tab
```

3.
 - a. Quel est le nombre de combinaisons possibles renvoyé par `toutes_combinaisons()` ?
 - b. Ajouter à `partieB.py` une fonction `toutes_combinaisons2()` qui renvoie le même tableau que `toutes_combinaisons()` mais en une seule instruction avec un tableau défini en compréhension.
4.
 - a. Ajouter à `partieB.py` le code complété de la fonction `compatible(proposition, tab_proposition, tab_reponse)` en respectant la spécification donnée dans la docstring.

```
1 def compatible(proposition, tab_proposition, tab_reponse):
2     """Retourne un booléen indiquant si proposition
3     donne les mêmes reponses [bulls, cows] pour toutes
4     les propositions et toutes les réponses enregistrées
5     dans les tableaux tab_proposition et tab_reponse
6     """
7     for k in range(len(tab_proposition)):
8         p, r = tab_proposition[k], tab_reponse[k]
9         if nombre_bulls_cows(proposition, p) != r:
10             return .....
11     return .....
```

- b. Vérifier si la fonction passe le jeu de tests en exécutant `test_compatible()` dans `test-partieB.py` (décommenter l'instruction). Si les tests sont passés, un message "Tests réussis pour compatible" devrait s'afficher.
5.
 - a. Ajouter à `partieB.py` le code complété de la fonction `joueur_ordinateur(index_courant, tab_combi, tab_proposition, tab_reponse)` en respectant la spécification donnée dans la docstring.

```
1 def joueur_ordinateur(index_courant, tab_combi, tab_proposition,
2     tab_reponse):
3     """Choix du joueur ordinateur
4     Renvoie un tableau constitué :
5     - de l'index de la prochaine combinaison à tester
6     - de la proposition choisie (tableau de 4 entiers) pour ce tour
7     """
8     proposition_ordi = tab_combi[index_courant]
9     while not compatible(proposition_ordi, tab_proposition,
10         tab_reponse):
11         .....
```



```
9 .....
10 .....
11 .....
12 return [index_courant, proposition_ordi]
```

- b. Vérifier si la fonction passe le jeu de tests en exécutant `test_joueur_ordinateur()` dans `test-partieB.py` (décommenter l'instruction). Si les tests sont passés, un message "Tests réussis pour joueur_ordinateur" devrait s'afficher.

6. Ajouter à `partieB.py` le code complété de la fonction `partie_joueur_ordinateur(nbmax_essai, debug = False)` en respectant la spécification donnée dans la docstring.

```
def partie_joueur_ordinateur(nbmax_essai, debug = False):
    """Lance une partie de Bulls and cows :
    l'ordinateur doit deviner la combinaison secrète
    en au plus nbmax_essai essais.
    Ses propositions sont des tableaux d'entiers.
    Renvoie le nombre d'essais ou -1 en cas d'échec
    """
    secret = combinaison_secrete()
    .....
    .....
    .....
    #il y a plus que 4 lignes à compléter !
    if bulls == 4:
        return essai
    else:
        return -1
```

7. a. Ajouter à `partieB.py` le code complété de la fonction `moyenne(tab)` en respectant la spécification donnée dans la docstring.

```
def moyenne(tab):
    """Retourne la moyenne des éléments d'un tableau"""
    .....
```

- b. Vérifier si la fonction passe le jeu de tests en exécutant `test_moyenne()` dans `test-partieB.py` (décommenter l'instruction). Si les tests sont passés, un message "Tests réussis pour moyenne" devrait s'afficher.

8. Ajouter à `partieB.py` le code complété de la fonction `generer_echantillon(n, nbmax_essai)` en respectant la spécification donnée dans la docstring. On importera le module `time` qui rassemble des fonctions de mesure du temps.

```
import time

def generer_echantillon(n, nbmax_essai):
    """Retourne un tableau contenant :
    - un tableau représentant les nombres d'essais
    pour un échantillon de n parties jouées par l'ordinateur.
    - le temps mis pour générer l'échantillon
    """
```

```
debut = time.perf_counter()
echantillon = .....
duree = time.perf_counter() - debut
return [echantillon, duree]
```

9. Copier le code fourni dans cadeau-partieB.py et l'ajouter à partieB.py.

Quelle instruction aurait permis d'utiliser la fonction `diagramme` dans `partieB.py` sans effectuer de copier/coller.

Quel sont les rôles des instructions `plt.title`, `plt.bar` et `plt.savefig`?

On s'appuiera sur la documentation du module `matplotlib.pyplot` à l'adresse :

https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.bar.html.

```
import matplotlib.pyplot as plt

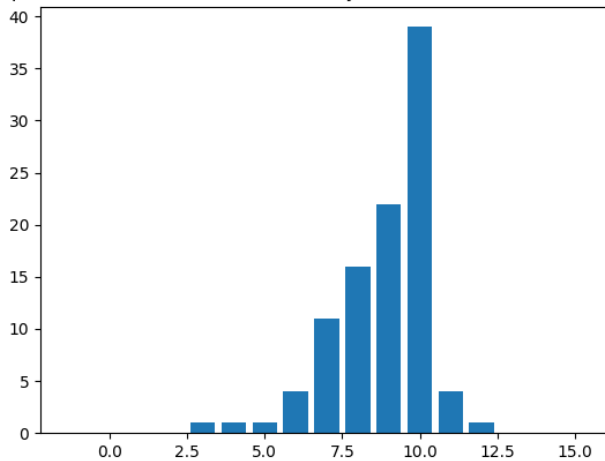
def diagramme(n, nbmax_essai):
    """Réalise un échantillon de nombres d'essais pour n parties
    Génère un diagramme batons et affiche la moyenne
    et la durée de création de l'échantillon
    """
    echantillon, duree = generer_echantillon(n, nbmax_essai)
    batons = [echantillon.count(k) for k in range(-1, nbmax_essai + 1)]
    plt.bar(list(range(-1, nbmax_essai + 1)), batons)
    plt.title("{} parties avec {} essais, en moyenne {} essais max, Duré
             e : {:.4} s".format(
                n, nbmax_essai, moyenne(echantillon), duree))
    plt.savefig("diagramme-{}.png".format(n))
```

10. Ajouter à `partieB.py` le code client ci-dessous qui est exécuté uniquement si `partieB.py` n'est pas importé sous forme de module dans un autre programme Python.

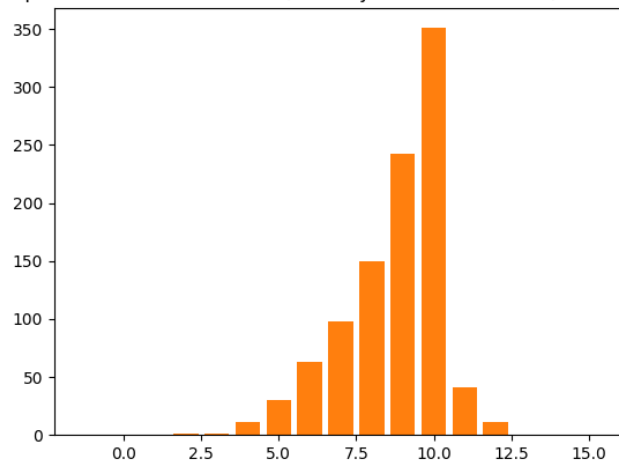
```
1  ## Code client exécuté uniquement si le programme n'est pas importé
2  if __name__ == '__main__':
3      partie_joueur_ordinateur(15, debug = True)
4      for n in [100, 1000, 10000]:
5          print("Diagramme avec un échantillon de taille {}".format(n))
6          diagramme(n, 15)
```

11. a. Commenter les résultats obtenus dans les diagrammes pour des échantillons de taille croissante : nombre moyen d'essais et temps d'exécution.

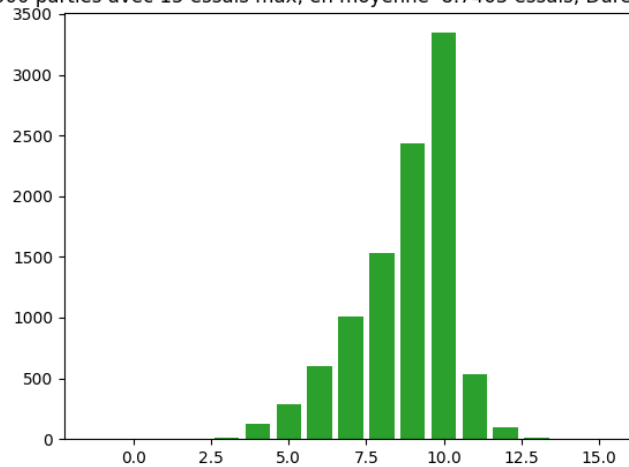
100 parties avec 15 essais max, en moyenne 8.85 essais, Durée : 5.648 s



1000 parties avec 15 essais max, en moyenne 8.743 essais, Durée : 57.18



10000 parties avec 15 essais max, en moyenne 8.7463 essais, Durée : 583.6



- b.** Il existe un autre algorithme, permettant de déterminer la combinaison secrète en au plus $10 + 3 + 2 + 1 = 16$ essais. La première étape coûte 10 propositions et consiste à proposer les 10 combinaisons de 4 chiffres identiques : 0000, 1111, ..., 9999. Quelle serait la deuxième étape?