Tournayre Jeanne Groupe B
Tissot Benoit

TP Informatique

Question 1:

Dans notre fonction évaluation nous testons tout d'abord si la longueur de notre solution est de la même longueur que celle de la combinaison proposée. En effet, si elle n'est pas de la même longueur, nous ne pouvons pas évaluer.
Ensuite, nous cherchons le nombre de bien placé nommé "bp". C'est-à-dire lorsque la couleur est bonne et à la bonne place. C'est pour cela que nous comparons dans la boucle uniquement les mêmes indices de la combinaison et de la solution. Lorsque la couleur est la même on rajoute un.
Une fois le nombre de bien placé connu nous recherchons le nombre de bonne couleur. On compare donc tous les termes de la combinaison avec ceux de la solution. Lorsque que le terme est similaire on ajoute 1 à la variable "color_ok"
et on remplace le terme de la solution par une virgule pour ne pas le recompter avec un autre un autre terme de la combinaison. Puis on met un break pour sortir de la boucle pour ne pas comparer le terme de la combinaison avec les autres termes de la solution. Une fois les deux boucles fini on soustrait la variable "bp" à "color_ok" ce qui nous donne le nombre de mal placé appelé "mp".
On a donc notre doublet bien placé, mal placé qui correspond à la solution.

Question 2:

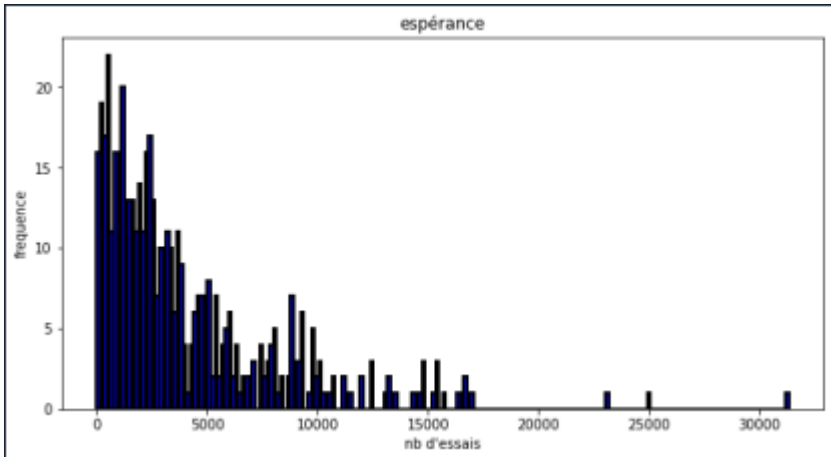
Nous faisons tout d'abord une initialisation en choisissant une combinaison aléatoire que nous nommerons "solution" que nous avons mis au préalable en variable global pour pouvoir la réutiliser dans les autres programmes.
Ensuite on utilise de façon récursive la fonction évaluation fait en question 1, en rappelant la variable "solution" fait dans la fonction init.

Question 3:

Pour calculer l'espérance du nombre d'essais avant de trouver la bonne réponse nous avons repéré que cela correspondait à une loi géométrique qui intervient lorsque l'on répète à l'infini et de façon indépendante la même expérience de Bernoulli de paramètre p et qu'on attend le premier succès.

ici la probabilité d'avoir la bonne réponse est p = $\frac{1}{4096}$ car il y a 8^4 possibilités.

Dans cette situation l'espérance vaut $E = \frac{1}{p}$ = 4096
L'espérance est donc de 4096.



Voici notre histogramme avec en abscisse le nombre d'essais et en ordonné la fréquence d'apparition. Comme dans cette fonction le codebreaker peut proposer plusieurs fois la même solution on peut monter jusqu'à plus de 30000 essais. Même si la majorité des essais sont avant les 5000.

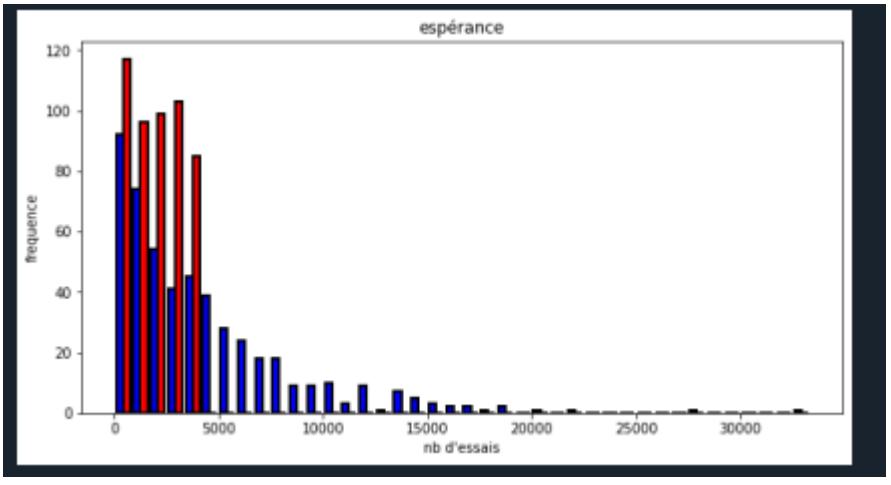
Question 4:

Cette fois ci on ne teste que les combinaisons jamais testées. Il n'y a donc pas de répétition et les essais ne sont pas indépendants. Il va donc falloir utiliser les probabilités conditionnelles.
A l'aide de cette formule de l'espérance : $\mathbb{E}(X) = \sum_{k=0}^{+\infty} x_k p_k$

avec $p_k = \frac{1}{4096}$ et $x_k = k$. On trouve donc une espérance E = 2048.5

Si on appelle le gain la différence des deux espérances on a :
G = 4096 - 2048.5 = 2047.5

Voici ce que donne notre histogramme:



En rouge, nous faisons jouer le codebreaker1 qui ne teste jamais deux fois la même combinaison alors qu'en bleu, nous faisons jouer le codebreaker0 qui peut tester plusieurs fois la même combinaison.
On remarque bien que les barres rouges ne dépassent pas 4096. C'est normal car dans le pire des cas le codebreaker teste toutes les combinaisons avant de trouver la bonne, c'est-à-dire 4096 combinaisons
Tandis que l'on trouve des barres bleues au-delà de 30000 essais.
Visuellement on repère bien que pour le codebreaker1 l'espérance va se trouver au milieu de 0 et 4096.
On remarque donc logiquement un gain entre le rouge et le bleu et l'on peut dire que le codebreaker1 est bien plus efficace.

Question 5:

Tout d'abord nous avons créé une fonction init qui va nous créer un set appelé possible réunissant toutes les 4096 combinaisons possibles.
Pour cela nous avons fait 4 boucles for pour faire passer toutes les variables selon tous nos quatres emplacements.
Ensuite nous avons créé une fonction donner_possible.
Tout d'abord, nous faisons une copie de notre liste des possible en possible2 afin de pouvoir la parcourir dans la boucle for sans avoir de problème de modification de la variable.
On compare ensuite la liste de toutes les possibilités avec la combinaison proposée, la solution est forcément présente dans les combinaisons qui ont la même évaluation que la combinaison proposée par le codebreaker. On enlève alors du set les combinaisons dont l'évaluation est différentes de la combinaison proposée car elles ne peuvent pas être solution. Pour finir, on enlève la combinaison testée puisqu'elle n'est pas juste.

Après avoir testé toutes les comparaisons, on se retrouve avec une liste possible2 contenant seulement les combinaisons ayant la même évaluation que notre combinaison testé, donc un set composé uniquement des combinaisons encore possibles après un essai.

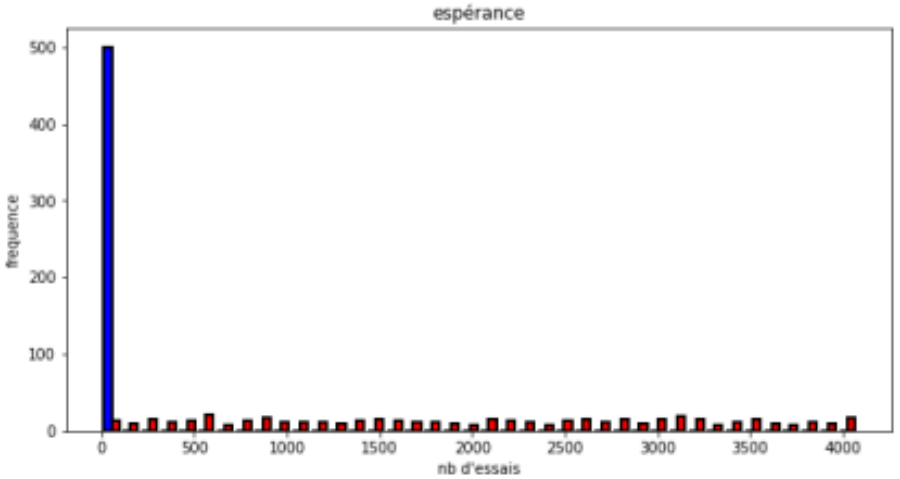
Question 6:

Le principe est le même que la question précédente, mais la fct possède un argument supplémentaire qui est la liste déjà réduite. On réduit alors encore la liste combinaison_possible en comparant l'évaluation de chaque élément restant avec l'évaluation de la combinaison testé comme précédemment.

Questions 7:

Dans cette version du codebreaker l'objectif est de tester de façon intelligente afin d'être plus efficace. Pour cela, il faut qu'il ne prenne que les combinaisons encore possible au vu des évaluations précédentes. Nous allons donc nous servir de *maj_possible* qui a chaque tour nous réduit la liste des combinaisons possibles et donc la liste dans laquelle le codebreaker doit piocher.
Tout d'abord, comme pour chaque codebreaker, nous avons fait une initialisation.
Ensuite, notre codebreaker prend aléatoirement une combinaison dans le set mis à jour par maj_possible en fonction de la combinaison proposée et de son évaluation comme dit dans la question précédente. Puis on va évaluer cette combinaison et recommencé pour réduire notre liste des possibles et trouver la bonne réponse.

Pour montrer graphiquement la différence du nombre d'essais entre codebreaker1 et codebreaker2, nous avons réalisé ce graphique a l'aide de la fonction espérance3.



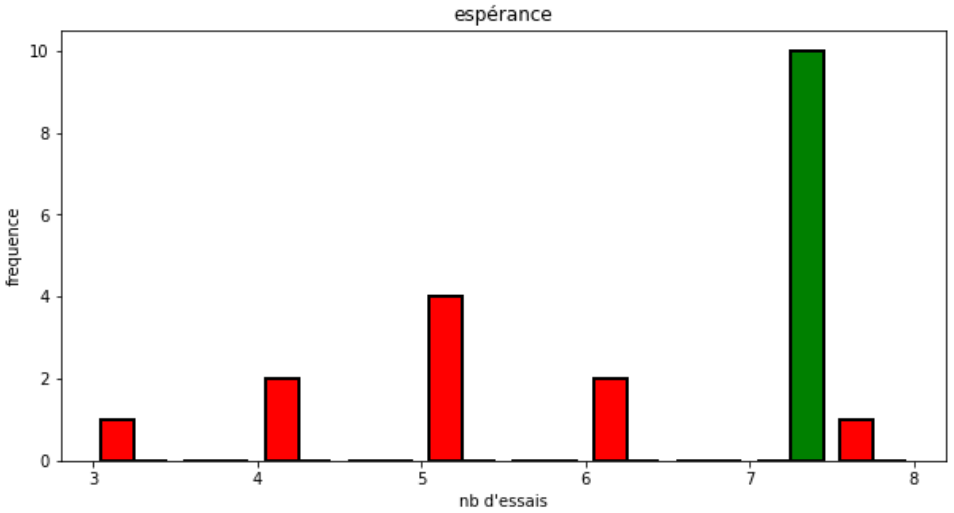
En rouge le nombre d'essais du codebreaker1 et en bleu le nombre d'essais du codebreaker2
Le codebreaker2 est donc beaucoup plus efficace et trouve la réponse à tous les coups dans les 100 premiers essais d'après le graphique (en réalité c'est moins). Alors que le codebreaker1 varie beaucoup plus en terme de nombre d'essais et peut monter comme nous l'avons déjà dit jusqu'à 4096 essais.

Question 8:

Cette fois nous allons autoriser notre codemaker à changer de combinaison au cours de la partie. Le codemaker ne peut cepandant pas prendre n'importe quelle combinaison. Son but est de choisir une combinaison telle que l'utilisateur ne se rende compte qu'il triche tout en prenant la pire possible pour lui, son objectif étant de faire durer la partie.
Pour cela nous avons créé une boucle for qui pour chaque combinaison (elt) de la liste combinaison_possible évalue cet élément par rapport à la combinaison proposé

comme si l'élément choisi était la nouvelle solution, puis on réduit la liste des combinaisons possibles comme pour une partie normale mais avec cette nouvelle solution. Il garde alors en mémoire l'élément qui réduit au moins la liste des possibles et ralentit ainsi le codebreaker.

Graphiquement voici ce que cela nous donne :



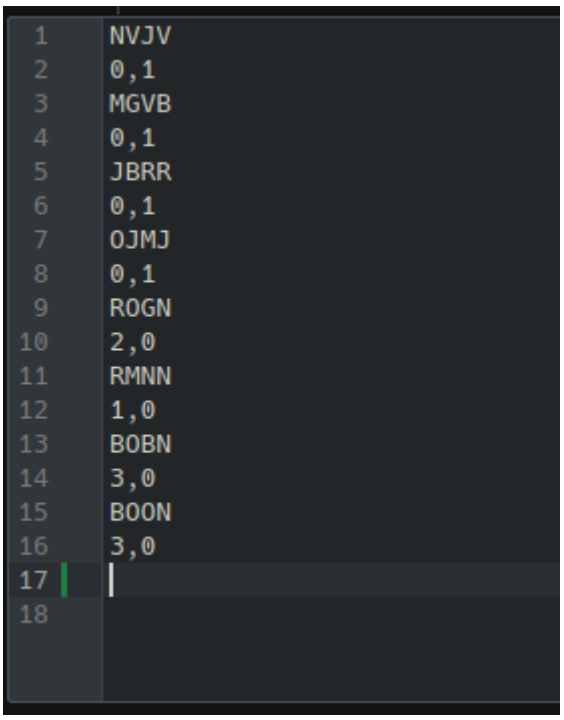
Nous avons fait le test sur seulement 10 parties car le programme pour le codemaker qui triche est très long.
En rouge nous avons les résultats du nombre d'essais de la partie sans tricher. Cela varie entre 3 et 7 essais. Si on fait la moyenne sur 10 parties, le codebreaker trouve la solution en 5 essais.
En vert les résultats de la partie avec triche. Ici la solution a été trouvée en 7 essais à chaque fois.
On remarque donc que si le codemaker triche il augmente de deux le nombre d'essais pour trouver la solution.

Questions 9 et 10 non effectuées

Question 11:

Dans cette nouvelle fonction nous réalisons une partie mais nous relevons en même temps les combinaisons proposées par le codebreaker et les évaluations données par le codemaker dans un fichier texte.
Ceci nous permet d'avoir un rendu propre de notre partie.

Voici à quoi ressemble notre fichier texte:



Question 12:

Le but ici est de vérifier que le codemaker n'a pas triché de façon visible. Nous prenons donc comme solution la dernière combinaison proposée par le codebreaker car celle-ci est correcte puis pour chaque essaie, nous comparons l'évaluation donnée par la fonction évaluation entre notre solution finale et la combinaison proposée à l'évaluation écrite sur le fichier.

Nous devons en effet trouver les mêmes évaluations car le codemaker doit changer de solution de façon à ce que ce changement n'invalide pas les évaluations précédentes.

On note par ailleurs l'utilisation d'un "try… except" qui permet de ne pas faire buguer le programme dans le fichier où le fichier serait inexistant.