# La réussite des alliances

Florian Berger - Lucas Lesage
Binôme 5

Printemps 2022

# Table des matières

# Avant-propos

Le projet présenté dans ce document est la réalisation d'un programme en Python permettant de jouer à une forme de réussite nommée « réussite des alliances », aussi appelée « la poussette ».

## Règles du jeu

Cette réussite peut être jouée avec un jeu de 32 ou de 52 cartes.

La partie commence en alignant trois cartes, de gauche à droite. On observe ensuite la carte située en avant-dernière position.

- Si elle est entourée de cartes ayant la même valeur ou la même couleur (on appellera une telle configuration une « alliance »), on prend la carte située au milieu et on la met sur la carte située immédiatement à sa gauche (on parle de « saut ») ;
- Sinon, on pioche une nouvelle carte, que l'on pose à droite de la dernière carte.

Une fois la carte posée, on observe l'avant-dernière carte, et on reproduit le schéma décrit ci-dessus.

Si un saut a pu être fait, on vérifie si d'autres alliances apparaissent. On peut alors effectuer les différents sauts possibles, consécutivement. La partie se termine une fois que la dernière carte de la pioche est posée et qu'aucun saut n'est possible.

Traditionnellement, la partie est gagnée lorsqu'il ne reste plus que deux tas de cartes, condition assez difficile à atteindre, et plus particulièrement avec un paquet de 52 cartes.

## Réalisation du projet

Ce projet a été réalisé tout au long du second semestre de l'année universitaire 2021-2022. Le premier cours nous a permis d'identifier les caractéristiques principales du projet, et d'établir quelles extensions nous souhaitions réaliser.

Chaque séance débutait par un récapitulatif de nos avancées respectives sur le programme. Nous présentions les fonctions que nous avions réalisées, et discutions des éventuelles modifications à apporter. A ce titre, bien que le concepteur de chaque fonction sera indiqué à la suite de ce document, il est important de tenir compte du fait qu'elles ont été sujettes à discussions et ont été améliorées collectivement.

## Lien vers le projet

`https://gitlab.isima.fr/lulesage/jeu-de-la-reussite-par-florien-berger-et-lucas-lesage.git`

# 1 Fonctions de formatage

```
cree_carte(couleur, valeur)
```

***conçue par :*** *Florian Berger*

La fonction prend en argument un caractère **couleur** et un caractère ou un entier **valeur**, et renvoie une carte au format dictionnaire.

Exemple : `cree_carte('K', 10)` renvoie `{'valeur':10, 'couleur':'K'}`

```
carte_to_chaine(carte, char_special = True)
```

***conçue par :*** *Lucas Lesage*

La fonction prend en argument une carte ainsi qu'un booléen char_special. La fonction renvoie une carte au format string. Cependant, en fonction de la valeur de char_special cela peut être soit avec les symboles des couleurs des cartes ou au format valeur-couleur.

Exemple : `carte_to_chaine({'valeur': 9, 'couleur': 'K'})` renvoie 9♢ et `carte_to_chaine({'valeur': 9, 'couleur': 'K'}, False)` renvoie 9-K

```
chaine_to_dico(carte)
```

***conçue par :*** *Florian Berger*

La fonction prend en argument une chaine de caractère **carte** au format `"valeur-couleur"`, et renvoie un dictionnaire désignant une carte au format `{'valeur': valeur, 'couleur': couleur}`.

Si la valeur de la carte est un entier, la valeur contenue dans le dictionnaire retourné est un entier. S'il s'agit d'une lettre `'A'`, `'R'`, `'D'` ou `'V'`, la valeur contenue dans le dictionnaire sera un caractère.

Exemples : `chaine_to_dico('9-T')` renvoie `{'valeur':9, 'couleur':'T'}`

`chaine_to_dico('R-T')` renvoie `{'valeur':'R', 'couleur':'T'}`

```
paquet_to_liste_dico(paquet)
```

***conçue par :*** *Florian Berger*

La fonction prend en argument une liste de chaines de caractères **paquet** au format `['valeur1-couleur1', 'valeur2-couleur2',...]`, et renvoie une liste de dictionnaires au format `[{'valeur': valeur1, 'couleur': couleur1}, {'valeur': valeur2, 'couleur': couleur2},...]`.

Exemple :

`paquet_to_liste_dico(['5-K', 'D-P'])` renvoie `[{'valeur':5, 'couleur':'K'}, {'valeur':'D', 'couleur':'P'}]`

```
cree_paquet_cartes(nb_cartes)
```

***conçue par :*** *Florian Berger et Lucas Lesage*

La fonction prend en argument un entier **nb_cartes** et renvoie une liste de chaines de caractères appelée

paquet au format `['valeur1-couleur1', 'valeur2-couleur2',...]`.

Un paquet compte 32 ou 52 cartes selon l'entier mis en argument, rangées par couleur puis par valeur. Si l'entier mis en argument est différent de 32 ou 52, le paquet renvoyé est vide.

Ces différentes fonctions dites "de formatage" ont pour but, comme leur nom l'indique, de changer le type associé à une carte ou à un paquet de cartes afin de pouvoir les exploiter dans d'autres fonctions décrites ultérieurement.

A titre d'exemple, la fonction `paquet_to_liste_dico` permettra de formater une pioche conservée dans un fichier de sauvegarde au format chaine de caractères en liste de dictionnaires, afin qu'elle soit utilisable dans un mode de jeu.

## 2 Fonctions principales

```
init_pioche_alea(nb_carte = 32)
```

***conçue par :*** *Lucas Lesage*

La fonction prend en argument un entier nb_carte, et renvoie un paquet de carte mélangé.

L'objectif de la fonction init pioche alea est de prendre un paquet de carte est de le mélanger. Pour cela, nous avons utilisé la fonction shuffle de la bibliothèque random.

```
afficher_reussite(liste_carte)
```

***conçue par :*** *Florian Berger*

La fonction prend en argument une liste de cartes **liste_carte**, au format liste de dictionnaires, et affiche chaque carte de cette liste au format `'Valeur Symbole'` au moyen de la fonction `carte_to_chaine` (voir section 1).

```
alliance(carte1, carte2)
```

***conçue par :*** *Florian Berger*

La fonction prend en argument deux cartes au format dictionnaire, **carte1** et **carte2**, et renvoie :
— Vrai si elles ont la même `'valeur'` ou la même `'couleur'` ;
— Faux sinon.

```
saut_si_possible(liste_tas, num_tas)
```

***conçue par :*** *Florian Berger*

La fonction prend en arguments :
— une liste de cartes **liste_tas**, au format liste de dictionnaires, qui représente les cartes révélées au joueur,
— un entier **num_tas**, représentant l'indice de la carte pour laquelle on veut vérifier la possibilité de faire un saut,
et renvoie :
— Vrai si le saut est possible ;
— Faux sinon.
La fonction vérifie également si l'entier **num_tas** correspond à un indice inclus dans la liste de tas et permettant un saut, c'est-à-dire s'il est compris entre la deuxième carte (qui aura pour indice 1) et l'avant-dernière carte de la liste. Dans le cas contraire, la fonction renverra Faux.

```
une_etape_reussite(liste_tas, pioche, affiche = False)
```

***conçue par :*** *Florian Berger*

La fonction prend en arguments :
— une liste de dictionnaires **liste_tas**, correspondant aux cartes révélées face au joueur,
— une liste de dictionnaires **pioche**, correspondant aux cartes restant dans la pioche,
— un booléen **affiche**, en argument optionnel, initialisé par défaut à Faux,

et réalise une étape de la réussite selon le schéma suivant :

1. On pioche une carte et on l'ajoute à la liste des tas en jeu,
2. On vérifie la possibilité de réaliser un saut à l'avant-dernière carte de la liste au moyen de la fonction `saut_si_possible` :
   - Si oui :
     → On réalise le saut,
     → On se place à la deuxième carte de la liste en partant de la gauche,
     → On vérifie si un saut est possible, et on le réalise tant que cela est possible à cet emplacement,
     → Une fois qu'on ne peut plus faire de saut à cet emplacement, on passe au tas situé immédiatement à droite, et on renouvelle la vérification à partir de ce tas,
     → On réalise cette étape jusqu'à arriver à la dernière carte de la liste ;
   - Si non, la fonction s'arrête.

Si l'argument **affiche** est initialisé à Vrai, alors la fonction affichera, au moyen de la fonction `afficher_reussite`, la liste de tas en jeu à chaque modification de cette liste, c'est-à-dire à l'étape de pioche et à chaque saut réalisé.

Afin de ne manquer aucune possibilité de saut, nous avons convenu de recommencer la vérification d'existence d'alliances à partir de la deuxième carte de la liste à chaque fois qu'un saut est effectué. En effet, il arrive que la réalisation d'un saut entraine l'apparition d'alliances à des emplacements d'indice inférieur à celui en cours de vérification.

# 3 Modes de jeu

```
reussite_mode_auto(pioche, affiche = False)
```

***conçue par :*** *Florian Berger*

La fonction prend en argument une liste de cartes **pioche**, au format liste de dictionnaires, et en argument optionnel un booléen **affiche**, initialisé à Faux par défaut. Elle réalise les différentes étapes d'une partie de réussite des alliances selon les règles décrites dans la rubrique "Avant-propos".

— Tant que la pioche n'est pas vide, elle utilise la fonction `une_etape_reussite` (voir section 2) à chaque nouvelle carte piochée.

— Une fois la pioche vide, il convient de vérifier s'il reste des sauts possibles. Pour ce faire, on applique à chaque carte à partir de la deuxième carte à gauche la fonction `saut_si_possible` (voir la section 2), autant de fois qu'un saut est possible, puis on passe au tas suivant, jusqu'à atteindre le dernier tas de la liste.

Si l'argument **affiche** est initialisé à Vrai, la fonction affichera, au moyen de la fonction `afficher_reussite` :

— La pioche utilisée pour jouer la partie,

— La liste de tas en jeu à chaque modification de cette liste, c'est-à-dire à chaque pioche et à chaque saut réalisé.

Elle renvoie une liste de dictionnaires représentant les cartes visibles une fois la partie terminée.

```
reussite_mode_manuel(pioche, nb_tas_max = 2)
```

***conçue par :*** *Florian Berger*

La fonction prend en argument une liste de cartes **pioche**, au format liste de dictionnaires, et en argument optionnel un entier **nb_tas_max**, initialisé à 2 par défaut, qui représente le nombre de tas à ne pas dépasser en fin de partie pour déclarer que l'utilisateur a gagné.

Elle permet à l'utilisateur de simuler une partie de réussite des alliances. A chaque étape, un menu s'affiche et propose plusieurs options :

1. Piocher une carte, tant que la pioche n'est pas vide. Dans le cas contraire, cette option n'est plus affichée ;
2. Proposer un saut : l'utilisateur sera alors invité à saisir l'indice de la carte sur laquelle il souhaite réaliser le saut. Si le saut est possible, il est effectué. Sinon, un message d'erreur apparaît, suivi du menu du mode manuel. Si l'indice saisi n'est pas valide, l'utilisateur est invité à saisir un nouvel indice ;
3. Quitter : dans ce cas, les cartes sont tirées consécutivement, sans vérifier si un saut est possible, jusqu'à ce que la pioche soit vide.

Une fois la partie terminée, si le nombre de tas restant est inférieur à l'entier **nb_tas_max**, un message `"Reussite !"` apparaît. Sinon, le message affiché sera `"Perdu..."`.

La conception de cette fonction n'a pas été sans difficultés. La première a été la conception d'un menu permettant au joueur d'effectuer les actions de son choix. L'implémentation d'une boucle `while` prenant en compte une valeur **reponse** saisie par l'utilisateur nous a semblé être une bonne solution, qui a cependant nécessité de réaffecter une valeur neutre à la variable **reponse** (ici, 0) dans les cas où le joueur décide d'effectuer une pioche ou de proposer un saut. Cette réaffectation permet d'afficher à nouveau le menu de jeu tant que la partie est en cours.

La seconde difficulté a été la gestion de la partie une fois la pioche vide. En effet, dans sa première version, la fonction ne proposait pas d'effectuer de saut une fois la dernière carte tirée, ce qui empêchait automatiquement d'atteindre les conditions de victoire si le nombre de tas maximal était fixé à 2. Il a fallu donc prévoir la situation où la pioche est vide et où le joueur n'a pas choisi de quitter la partie en cours. Un nouveau menu, permettant uniquement de proposer un saut ou de quitter, a été mis en place sur le même principe que le précédent.

```
lance_reussite(mode, nb_carte = 32, affiche = False, nb_tas_max = 2)
```

***conçue par :*** *Lucas Lesage*

La fonction prend en argument une chaine de caractère mode (définissant le mode de jeu), un entier nombre de carte, un booléen affiche et un entier nb_tas_max. Et qui lance une partie choisie par le mode de jeu désiré.

L'objectif la fonction est de pouvoir lancer une partie en mode automatique ou en mode manuel.

# 4 Chargement / sauvegarde de partie

```
init_pioche_fichier(nom_fichier)
```

***conçue par :*** *Lucas Lesage*

La fonction prend en argument une chaine de caractère nom_fichier. Et qui renvoie une liste de carte qui aura été lue dans un fichier.

```
ecrire_fichier_reussite(nom_fichier_sauvegarde, pioche)
```

***conçue par :*** *Lucas Lesage*

La fonction prend en argument une chaine de caractère nom_fichier ainsi qu'une liste pioche. La fonction ne renvoie rien mais écris dans un fichier les cartes composant la pioche.

L'objectif de ces fonctions est de pouvoir lire et sauvegarder des parties à partir d'un fichier. Nous avons pu observer une difficulté au niveau de la fonction écrire réussite sur le point de vue du format approprié à inscrire. Pour la fonction init pioche fichier, nous avons tout d'abord transformé, le fichier en une liste de carte au format valeur-couleur (10-P). Puis nous avons transformée chacune des cartes au format de dictionnaire de carte (ex : {'valeur' : 10, 'couleur' : 'P'}) grâce à la fonction chaine to dico. Pour la fonction écrire fichier, nous avons décidé de parcourir la liste de carte au format dictionnaire pour les transformer en des cartes au format valeur-couleur au moyen de la fonction carte to chaine puis de les écrire dans le fichier désiré.

# 5 Menu du jeu

```
menu_reussite()
```

***conçue par :*** *Florian Berger*

La fonction ne prend aucun argument, et constitue l'unique fonction de notre programme principal. Elle permet à l'utilisateur d'effectuer les fonctionnalités suivantes :

— Jouer une partie ;
— Effectuer une simulation ;
— Quitter le programme.

Nous avons décidé de concevoir cette fonction suite à un constat simple : dans le cadre de la réalisation de ce projet, il nous a été proposé de mettre en place les fonctions `init_pioche_fichier` et `ecrire_fichier_reussite`, qui permettent la mise en place d'un système de chargement / sauvegarde de pioche (pour plus de précisions, voir la section 4). Cependant, celles-ci ne sont pas utilisées dans le cadre des consignes proposées. Nous avons donc réfléchi à la meilleure façon de les intégrer au jeu, tout en conservant l'accessibilité du programme.

Nous avons donc conçu une fonction réunissant la majorité des fonctionnalités prévues dans le programme au moyen d'un menu textuel. L'utilisateur est amené à faire ses choix par saisie clavier. Chaque menu décrit dans cette section a été développé au moyen de boucles `while`, afin de proposer une nouvelle saisie à l'utilisateur en cas d'erreur, mais aussi pour permettre de réaliser plusieurs actions sans avoir à quitter le programme.

## Jouer une partie

Une fois cette option sélectionnée, le joueur est amené à choisir s'il souhaite effectuer une partie automatique ou une partie manuelle, au moyen des fonctions décrites dans la section 3.

Indépendamment du choix du mode de jeu, il est demandé au joueur s'il veut utiliser une pioche sauvegardée. Dans ce cas, l'utilisateur saisi le nom du fichier (au format .txt) dans lequel est enregistrée la pioche qu'il souhaite utiliser. L'existence de cette sauvegarde est vérifiée par cette partie du code :

```
if jeu_fichier.lower()== "oui" :
    erreur = True
    while erreur :
        nom_fichier = input("Fichier: ")
        try :
            with open(nom_fichier+".txt") as file :
                print("Chargement...")
                erreur = False
            pioche = init_pioche_fichier(nom_fichier+".txt")
            if (len(pioche) not in [32, 52]) or not (verifier_pioche(pioche
                , len(pioche))):
                pioche_invalide = int(input("Attention, vous jouez avec une
                    pioche non reglementaire ! \nVoulez-vous charger un
                   autre fichier ?\n 1- Oui\n 2- Non\n 3- Charger une
                   pioche aleatoire\n "))
                if pioche_invalide == 1 :
                    erreur = True
                    pioche = []
                elif pioche_invalide == 3 :
                    pioche = []
                    erreur = False
                    jeu_fichier = "non"
```

```
            except FileNotFoundError:
                confirm = int(input("Fichier non trouve.\n1- Saisir un nouveau
                    nom de fichier\n2- Charger une pioche aleatoire\n"))
                if confirm == 2 :
                    erreur = False
                    jeu_fichier = "non"
```

Le code fonctionne comme suit :

— Si le joueur souhaite charger une partie enregistrée, un booléen `erreur` est initialisé à Vrai. S'il contient la valeur Vrai, cela signifie que le fichier que l'on tente de charger n'existe pas.

— Tant que **erreur** contient la valeur Vrai, le joueur est invité à saisir le nom du fichier qu'il souhaite charger.

— La commande `try :` essaie d'ouvrir le fichier .txt dont le nom a été saisi par l'utilisateur.

→ S'il existe, la pioche sera initialisée à partir de ce fichier, et la variable **erreur** contiendra la valeur Faux. La pioche ainsi générée est alors testée au moyen de la fonction `verifier_pioche` (décrite à la section 6.3).
Si celle-ci n'est pas valable, il sera proposé au joueur, au moyen d'un menu :

1. de saisir un autre nom de fichier (**erreur** prendra pour valeur Vrai, et la pioche sera vidée),
2. jouer avec la pioche malgré son invalidité,
3. générer une pioche aléatoire (**erreur** prendra pour valeur Vrai, et la pioche sera vidée, et la variable **jeu_fichier** prendra pour valeur "non").

→ Sinon, le programme sera confronté à une erreur de type `FileNotFoundError`. Dans ce cas, un message apparaît. Le joueur est invité, au moyen d'un menu, à saisir un nouveau nom de fichier, ou à générer une pioche aléatoire (auquel cas la variable **erreur** contiendra la valeur Faux, permettant de sortir de la boucle, et la variable **jeu_fichier** prendra pour valeur "non").

Si le joueur ne souhaite pas charger de partie enregistrée, il est alors amené à choisir la taille du paquet de cartes, qui sera généré aléatoirement.

La partie se déroule selon le mode choisi. Si la pioche a été générée aléatoirement, il est proposé au joueur de sauvegarder la pioche utilisée pour la partie. Cette fonctionnalité est gérée par cette partie du code :

```
enreg = input("Voulez vous sauvegarder cette pioche ? oui / non : ")
if enreg.lower() == "oui" :
    erreur_sauv = True
    while erreur_sauv :
        nom_fichier_sauv = input("Nom de la sauvegarde : ")
        try :
            with open(nom_fichier_sauv + ".txt", 'r') as file :
                confirm = int(input("Ce nom est deja utilise.\n 1- Saisir
                    un nouveau nom de fichier\n 2- Ne pas sauvegarder\n"))
                if confirm == 2 :
                    erreur_sauv = False
        except FileNotFoundError :
            ecrire_fichier_reussite(nom_fichier_sauv + ".txt", pioche)
            erreur_sauv = False
```

— Si le joueur souhaite sauvegarder le paquet avec lequel il vient de joueur, un booléen **erreur_sauv** est initialisé à Vrai. S'il contient la valeur Vrai, cela signifie que le fichier que l'on tente de charger existe, et empêche donc toute nouvelle sauvegarde.

— Tant que **erreur_sauv** contient la valeur Vrai, le joueur est invité à saisir le nom du fichier sous lequel il souhaite réaliser sa sauvegarde.

— La commande `try :` essaie d'ouvrir *en mode lecture* le fichier .txt dont le nom a été saisi par l'utilisateur.
  → Si le programme parvient à ouvrir un fichier à ce nom, cela signifie que ce fichier existe. Un message d'erreur avertit l'utilisateur qu'un fichier porte déjà ce nom. Il est alors invité, au moyen d'un menu, à saisir un autre nom, ou à ne pas sauvegarder la pioche (auquel cas, **erreur_sauv** prendra la valeur Faux, nous permettant de sortir de la boucle).
  → Sinon, le programme sera confronté à une erreur de type `FileNotFoundError`. Cela signifie qu'aucun fichier .txt à ce nom n'existe, et qu'une sauvegarde est donc possible. La pioche est alors enregistrée, et on quitte la boucle.

L'utilisateur se retrouve alors à nouveau face au menu principal du jeu.

# 6 Extensions

## 6.1 Statistiques

```
res_multi_simulation(nb_sim, nb_cartes = 32)
```

***conçue par :*** *Lucas Lesage*

La fonction prend en argument 2 entiers : **nb_sim** et **nb_cartes** représentant le nombre de simulations ainsi que le nombre cartes désirées. La fonction renvoie une liste du nombre de tas finaux à la fin de chaque simulation.

```
statistiques_nb_tas(nb_sim, nb_cartes = 32)
```

***conçue par :*** *Lucas Lesage*

La fonction prend en argument 2 entiers : **nb_sim** et **nb_cartes** représentant le nombre de simulations ainsi que le nombre cartes désirées. La fonction ne renvoie rien mais calcule la moyenne du nombre des tas finaux obtenu par la fonction `res_multi_simulation`, le minimum et maximum de la liste. Elle affichera le tout.

```
graphique_stats(nb_sim, nb_cartes = 32)
```

***conçue par :*** *Lucas Lesage*

La fonction prend en argument 2 entiers : **nb_sim** et **nb_cartes** représentant le nombre de simulations ainsi que le nombre cartes désirées. La fonction ne renvoie rien mais crée un affichage des tas finaux ainsi que la moyenne.

L'objectif de cette fonction est de former un nuage de points des tas obtenu grâce à la fonction res_multi_simulation, ainsi qu'une droite d'équation y=moyenne représentant la moyenne des tas. Elle obtenue grâce à la fonction moyenne qui calcule et renvoie la moyenne. La fonction a été programmé grâce à `matplotlib`, nous avons rencontré des difficultés dans la compréhension et l'utilisation du module. Nous avons alors cherché sur différents sites des tutoriels et explications du module.

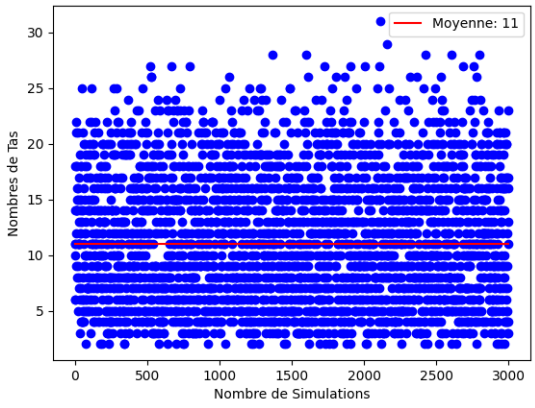


FIGURE 1 – Graphique représentant une simulation de 3000 parties avec 32 cartes

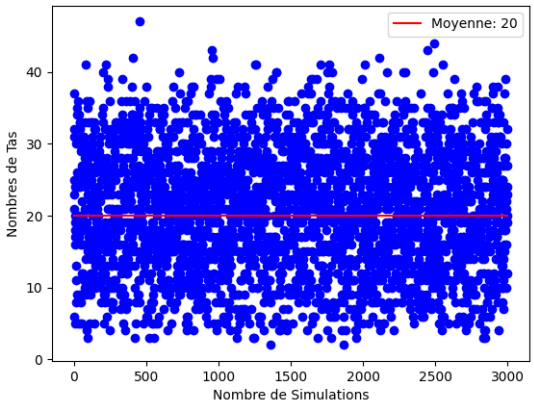


FIGURE 2 – Graphique représentant une simulation de 3000 parties avec 52 cartes

## 6.2 Probabilité

```
proba(nb_sim, nb_carte = 32)
```

***conçue par :*** *Lucas Lesage*

La fonction prend en argument 2 entiers : **nb_sim** et **nb_cartes** représentant le nombre de simulations ainsi que le nombre cartes désirées. La fonction renvoie un dictionnaire contenant 32 ou 52 clés en fonction du nombre de carte voulu.

L'objectif de cette fonction est de savoir exactement sur le nombre de simulation le nombre de fois où il y a eu 2 tas puis 3 et ainsi de suite jusqu'au nombre de carte. Ces données sont alors stockées dans un dictionnaire pour pouvoir accéder aux données facilement.

```
graph_proba(nb_sim, nb_carte = 32)
```

***conçue par :*** *Lucas Lesage*

La fonction prend en argument 2 entiers : **nb_sim** et **nb_cartes** représentant le nombre de simulations ainsi que le nombre cartes désirées. Elle ne renvoie rien mais affiche un graphique.

L'objectif de cette fonction est d'afficher la répartition des tas sur un nombre de simulations donnée.

La fonction est composée de 3 parties majeures.

Premièrement, cela va récupérer un dictionnaire généré par la fonction `proba`, puis va créer deux listes : `liste_proba` et `liste_probas`. La première est une liste de couples c'est-à-dire au format `[(clé, entier)]`. La deuxième quant à elle ne va contenir que les valeurs contenues dans le dictionnaire grâce à la fonction `dico_to_liste`, cette liste va permettre le calcule de la moyenne et de l'écart-type grâce à la bibliothèque numpy qui le permet.

Dans un deuxième temps la fonction va afficher les résultats sous la forme d'une fonction graphique. Pour cela, elle va utiliser la liste `liste_proba` pour former des couples (x,y) grâce à la fonction zip. Par la suite, elle va tracer la fonction obtenue grâce au couple (x,y).

Et enfin dans un troisième temps, la fonction va regarder si le nombre de simulation est supérieur à 325. En effet, si le nombre de simulation est inférieur à 325 cela va lancer la génération d'une fonction représentant la densité de la loi Normal, obtenu grâce à la fonction `DensiteNormal` prenant en argument la moyenne et l'écart-type. La formule a été obtenu grâce à la formule mathématiques trouvable sur internet. Il devait y

avoir aussi un histogramme représentant la loi normal associée mais malheureusement les résultats n'étaient pas satisfaisants d'un point de vue rendu.

Les problèmes obtenus sur la création de cette fonction ont été très nombreux comme la recherche autour de la génération d'une loi normal, mais aussi comment transformer un dictionnaire en tuples. Cela a nécessité énormément de recherches approfondies sur diverses plateformes montrant d'ailleurs l'immense possibilité derrière ces quelques modules que sont `matplotlib` et `numpy`.

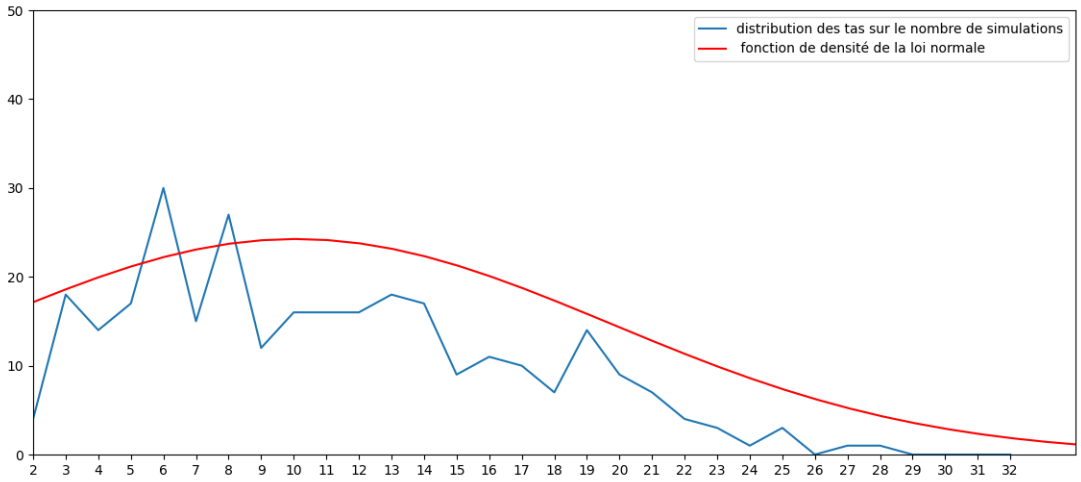


FIGURE 3 – Graphique représentant une simulation de 300 parties avec 32 cartes

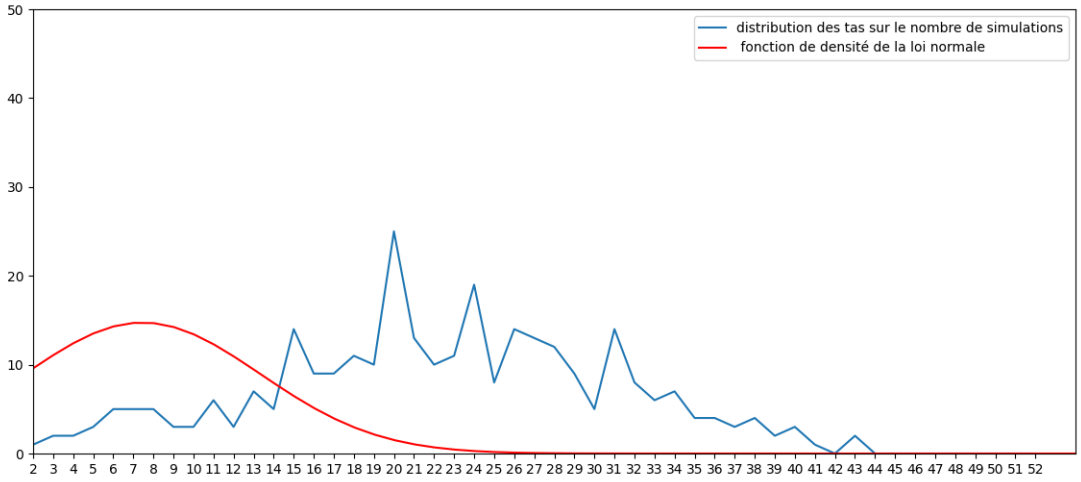


FIGURE 4 – Graphique représentant une simulation de 300 parties avec 52 cartes

### 6.3 Verifier Pioche

```
verifier_pioche(pioche, nb_cartes = 32)
```

***conçue par :*** *Lucas Lesage*

La fonction prend en argument une liste pioche et un entier **nb_cartes**. La fonction renvoie un booléen désignant si la pioche est complète et totalement juste.

L'objectif de cette fonction est de vérifier si un jeu de carte est correctement initialisé c'est-à-dire le bon nombre de carte et si elles y sont toutes. La Fonction va tout d'abord vérifier s'il y a le bon nombre de carte puis si la condition y est rentrée, elle va parcourir les cartes et vérifier les cartes une par une grâce à un compteur.

# Liste des fonctions présentées

## A - C



## E - I



## L - P



## R - V

# Bibliographie


— Daniel Diaz (15 juin 2021). *7 façons de vérifier si un fichier ou un dossier existe en Python.* `https://geekflare.com/fr/check-if-file-folder-exists-in-python/`
— Documentation Matplotlib `https://matplotlib.org/stable/users/index`