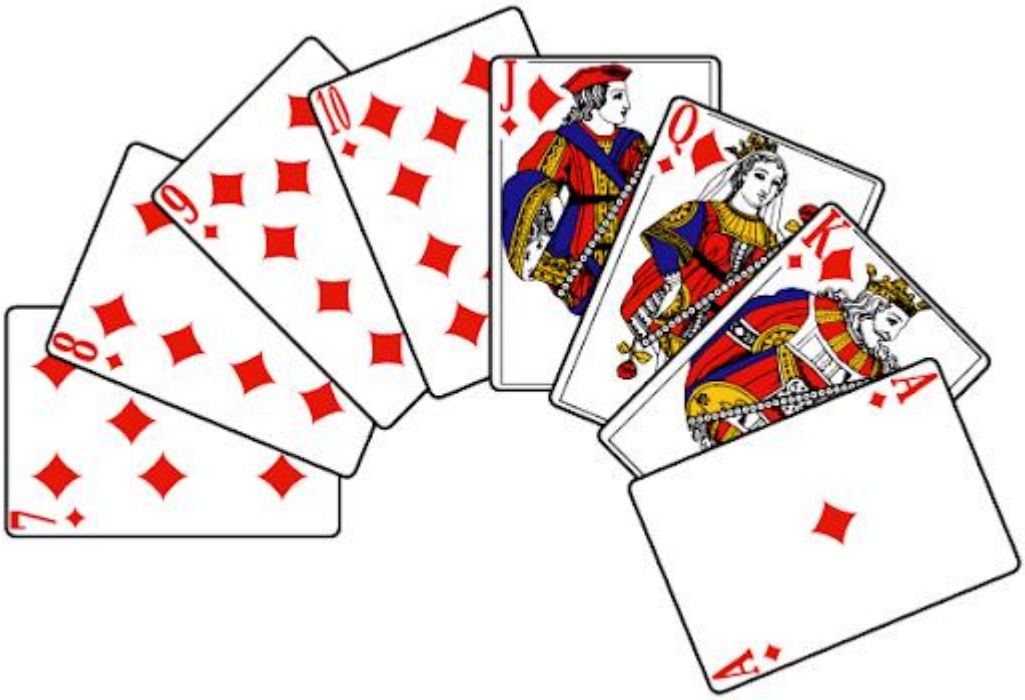
Mars 2024

Rapport *(version 1)*

Projet n°2 de Première NSI

La Belote



Irène Pasquignon,

Avigdor Olender,

Nadine Heiba

Sommaire :

1. Consignes

Ces consignes s'intègrent dans l'esprit de la rédaction d'un journal de bord. Ainsi, nous avons décidé de les intégrer dans notre rapport pour mettre au clair les éléments à travailler dans notre groupe au cours de l'avancement sur le projet, et pour nous permettre de nous y référencer durant la rédaction du rapport.

Réaliser un ou plusieurs jeux fonctionnels avec la base du projet bataille, avec éventuellement une interface graphique (TKinter à utiliser dans un deuxième temps). Les recherches et avec IA sont autorisées, à deux conditions :

- Compréhension de tous les éléments (tout sourcer si nous sommes allé chercher ailleurs).
- Compréhension de la logique indispensable du "class" en python.

Le code final doit être optimisé. Il est possible également d'essayer d'en faire un ".exe", pour le rendre exportable sur pc comme une application (dans un deuxième temps)
Il y a donc 2 versions (relativement) du code à indiquer sur ce rapport :

1. Version initiale/intermédiaire
2. Version finale avec commentaires et lignes de code pour s'assurer du fonctionnement du code (second temps, avec ajout de l'interface graphique et la possibilité d'application)

Le rapport doit contenir une déclaration d'intention au début du projet (introduction), puis un journal de bord (principale partie du rapport), puis avec une conclusion qui explique tout ce qu'on a fait, par rapport à notre intention au début du projet.
Le format du rapport doit être un PDF, et le code sous la forme d'un fichier (.py), ou sinon avec un lien vers un projet Replit partagé, ou GitHub.
Il est possible d'ajouter dans la version finale du rapport une partie « statistiques », avec les résultats de différents tests sur les différents jeux (nous l'ajouterons surtout après l'ajout complet de l'interface graphique)

A rendre pour le 26/03/2024 : état d'avancement, rapport non définitif, sous forme de carte de bord. **Faire une conclusion sur comment faire pour être meilleur : élément clé de la note**

Après les vacances : projet réellement fini, avec l'ajout de tous les éléments pour avoir une interface graphique, et éventuellement un projet exportable

2. Introduction

Lors de notre première séance, nous avons rapidement convergé vers le choix du projet portant sur le jeu de la belote. Cette décision a été motivée par le fait que deux des trois membres du groupe pratiquent activement ce jeu, et que nous avons jugé son développement potentiellement intéressant. Notre objectif est donc de concevoir un jeu, potentiellement transformable en application, doté d'une interface graphique.

Nous avons rapidement réparti les tâches entre nous trois. Nadine se concentrera principalement sur le développement de l'interface graphique, tandis qu'Irène collaborera avec elle, en prenant en charge la programmation proprement dite. De son côté, Avigdor se chargera de développer le projet Bataille et tentera de rendre la Belote jouable sur 4 appareils différents en local. Nous avons convenu de communiquer régulièrement lors des séances de classe pour mettre en commun nos avancements respectifs, et nous effectuerons un bilan oral à la fin de chaque séance afin de nous aligner sur nos objectifs pour les séances suivantes.

L'avancement du projet sera structuré en deux parties distinctes :

La première partie, (qui s'étendra jusqu'aux vacances), consistera à établir un code fonctionnel pour nos jeux (la belote et la bataille), à amorcer la mise en place de l'interface graphique, et à rédiger un journal de bord tout au long du processus.

La seconde partie, (qui prendra place après les vacances), sera dédiée à la finalisation du jeu avec une interface graphique attrayante et à la possibilité de rendre le code exportable.

Ce rapport rend compte de l'avancement du projet jusqu'au 26/03/2024, date de la veille des vacances, correspondant ainsi à la première partie du projet.

3. Code du jeu « Belote »

a. Version 1 (base, départ)

```
import random  # #import #aléatoire

cartes = ['7', '8', '9', '10', 'Valet', 'Dame', 'Roi', 'As']  # #définition #cartes

valeurs = {'7': 7, '8': 8, '9': 9, '10': 10, 'Valet': 11, 'Dame': 12, 'Roi': 13, 'As': 14}  # #dictionnaire
#valeurs

paquet = [(v, couleur) for v in cartes for couleur in ['Coeur','Carreau', 'Trèfle', 'Pique']]  #
#génération #paquet #cartes

random.shuffle(paquet)  # #mélange #aléatoire

print(paquet)  # #affichage #paquet

main_joueur1 = paquet[:8] # prend toutes les cartes de 0 à 7 et les atribue à la variable  #
#attribution #main_joueur1
main_joueur2 = paquet[8:16]  # #attribution #main_joueur2
main_joueur3 = paquet[16:24]  # #attribution #main_joueur3
main_joueur4 = paquet[24:32]  # #attribution #main_joueur4

print("Main du Joueur 1:", main_joueur1)  # #affichage #main_joueur1
print("Main du Joueur 2:", main_joueur2)  # #affichage #main_joueur2
print("Main du Joueur 3:", main_joueur3)  # #affichage #main_joueur3
print("Main du Joueur 4:", main_joueur4)  # #affichage #main_joueur4

def valeur_carte(carte):  # #définition #fonction #valeur_carte
    return valeurs[carte[0]]  # #retourne #valeur #carte

def gagnant_pli(plis, atout):  # #définition #fonction #gagnant_pli
    couleur_demandee = plis[0][1]  # #couleur #demandée
    plis_couleur_demandee = [pli for pli in plis if pli[1] == couleur_demandee]  # #filtrage #plis
#couleur_demandee
    if plis_couleur_demandee:  # #si #plis_couleur_demandee
        return max(plis_couleur_demandee, key=lambda x: valeur_carte(x))  # #retourne #maximum
#plis_couleur_demandee
    else:  # #sinon
        return max(plis, key=lambda x: valeur_carte(x) if x[1] == atout else 0)  # #retourne #maximum
#plis

plis_tour = []  # #initialisation #plis_tour
for _ in range(8):  # #boucle #8 #fois
    plis_tour.append([main_joueur1.pop(0), main_joueur2.pop(0), main_joueur3.pop(0),
main_joueur4.pop(0)])  # #ajout #plis_tour
```

```
for i, pli in enumerate(plis_tour):  # #boucle #enumerate #plis_tour
    print(f"\nTour {i + 1}:")  # #affichage #tour
    print("Joueur 1 a joué:", pli[0])  # #affichage #joueur1
    print("Joueur 2 a joué:", pli[1])  # #affichage #joueur2
    print("Joueur 3 a joué:", pli[2])  # #affichage #joueur3
    print("Joueur 4 a joué:", pli[3])  # #affichage #joueur4
    gagnant = gagnant_pli(pli, 'Coeur')  # L'atout est défini ici comme le cœur pour l'exemple  #
#détermination #gagnant
    print("Le gagnant du pli est:", gagnant)  # #affichage #gagnant
```

Version 2 (finale)

```
import random

# Définition des constantes

COULEURS = ['Coeur', 'Carreau', 'Trèfle', 'Pique']

VALEURS = ['7', '8', '9', 'Valet', 'Dame', 'Roi', '10', 'As']

POINTS = {'7': 0, '8': 0, '9': 0, '10': 10, 'Valet': 2, 'Dame': 3, 'Roi': 4, 'As': 11}

POINTS_ATOUT = {'7': 0, '8': 0, '9': 14, '10': 10, 'Valet': 20, 'Dame': 3, 'Roi': 4, 'As': 11}

PLIS = 8

plis_memoire = []


# Fonction pour créer un jeu de cartes

def creer_jeu():
    return [(v, c) for v in VALEURS for c in COULEURS]


# Fonction pour mélanger les cartes

def melanger_cartes(jeu):
    random.shuffle(jeu)


# Fonction pour distribuer les cartes dans le sens des aiguilles d'une montre

def distribuer_cartes_sens_horaire(jeu, donneur):
    mains = {i: [] for i in range(4)}

    joueur_actuel = (donneur + 1) % 4

    for _ in range(5):

        for _ in range(4):
            mains[joueur_actuel].append(jeu.pop(0))

            joueur_actuel = (joueur_actuel + 1) % 4

    # Placer une carte face visible

    # carte_visible = jeu.pop(0)

    return mains


def distribuer_cartes_finales_sens_horaire(jeu, donneur, preneur, mains):
    joueur_actuel = (preneur - 1) % 4
```

```
    print(jeu)

    for k in range(3):

        for i in range(4):

            # print("preneur = ",k,preneur)

            # print("joueur = ",joueur_actuel)

            if k == 0:

                # print("ok")

                mains[joueur_actuel].append(jeu.pop(0))  # carte visible

            else:

                mains[joueur_actuel].append(jeu.pop(0))

            joueur_actuel = (joueur_actuel + 1) % 4

        print("k=", k, jeu)

    return mains


# Fonction pour afficher les mains des joueurs

def afficher_mains(mains):
    print("on affiche les mains")

    for joueur, main in mains.items():

        print(f"Main du joueur {joueur + 1}:")

        for i, (v, c) in enumerate(main, start=1):
            print(f"{i}. {v} de {c}")

        print()


# Fonction pour déterminer le vainqueur du pli


# Fonction pour déterminer le vainqueur du pli

def determiner_vainqueur(pli, atout):
    pli_atout = [carte for carte in pli if carte[1] == atout]

    pli_non_atout = [carte for carte in pli if carte[1] != atout]

    if pli_atout:  # S'il y a des cartes atout dans le pli

        pli_atout_trie = sorted(pli_atout, key=lambda x: (VALEURS.index(x[0]), COULEURS.index(x[1])))

        vainqueur = pli_atout_trie[-1]

        index_vainqueur = pli.index(vainqueur) + 1

    else:  # S'il n'y a pas de cartes atout dans le pli

        pli_non_atout_trie = sorted(pli_non_atout,
                                    key=lambda x: (x[1] != atout, VALEURS.index(x[0]), COULEURS.index(x[1])))

        vainqueur = pli_non_atout_trie[-1]

        index_vainqueur = pli.index(vainqueur) + 1

    print(f"Le vainqueur du pli est le joueur {index_vainqueur} avec la carte {vainqueur[0]} de {vainqueur[1]}")

    return vainqueur, index_vainqueur


# Fonction pour gérer le contrat
```

```
def gerer_contrat(donneur, carte_visible):
    print(f"La carte retournée est : {carte_visible[0]} de {carte_visible[1]}")

    joueur_actuel = (donneur + 1) % 4

    while joueur_actuel != donneur:

        decision = input(

            f"Joueur {joueur_actuel + 1}, souhaitez-vous prendre (P) ou passer (A) ? "

        ).lower()

        if decision == "p":

            print(

                f"Le joueur {joueur_actuel + 1} a pris l'atout : {carte_visible[1]}")

            preneur = joueur_actuel + 1

            return preneur

            # return carte_visible[1]  # Retourner la couleur de l'atout

        elif decision == "a":

            print(f"Le joueur {joueur_actuel + 1} passe.")

            joueur_actuel = (joueur_actuel + 1) % 4

        else:

            print("Choix invalide, veuillez réessayer.")

            print("Personne n'a pris. Nouvelle donne.")

            preneur = NONE

    return preneur  # Si personne n'a pris, retourner None pour refaire une nouvelle donne


# Fonction pour vérifier les annonces de chaque joueur

def verifier_annonces(joueur):
    annonces = []

    cartes = JOUEURS[joueur]

    annonces.append(verifier_carres(cartes))

    annonces.append(verifier_suites(cartes))

    return annonces


# Fonction pour vérifier les carrés

def verifier_carres(cartes):
    carres = {'Roi': 0, 'Dame': 0, 'Valet': 0, '10': 0, 'As': 0}

    for valeur, couleur in cartes:

        if valeur in carres:
            carres[valeur] += 1

    annonces_carres = []

    for valeur, occurences in carres.items():

        if occurences >= 4:

            if valeur == 'Valet':

                annonces_carres.append(('carré de valets', 200))
```

```
        elif valeur == '9':
            annonces_carres.append(('carré de 9', 150))
        else:
            annonces_carres.append(('carré de ' + valeur.lower(), 100))
    return annonces_carres

# Fonction pour vérifier les suites
def verifier_suites(cartes):
    suites = {'Coeur': [], 'Carreau': [], 'Trèfle': [], 'Pique': []}
    for valeur, couleur in cartes:
        suites[couleur].append(VALEURS.index(valeur))
    annonces_suites = []
    for couleur, valeurs in suites.items():
        valeurs.sort()
        longueur_suite = 1
        suite_precedente = None
        for valeur in valeurs:
            if suite_precedente is not None and valeur == suite_precedente + 1:
                longueur_suite += 1
            else:
                longueur_suite = 1
            if longueur_suite >= 3:
                points = 20 if longueur_suite == 3 else (
                    50 if longueur_suite == 4 else 100)
                annonces_suites.append(('suite à ' + couleur.lower(), points))
            suite_precedente = valeur
    return annonces_suites

# Fonction pour afficher les annonces d'un joueur
def afficher_annonces(annonces):
    for annonce in annonces:
        print(f"Annonce: {annonce[0]} - Points: {annonce[1]}")

# Fonction pour jouer un tour
def jouer_tour(pli, joueur, carte_maitre, couleur_demandee):
    print(f"Tour du joueur {joueur}")
    print("Cartes jouées jusqu'à présent : ", pli)
    print("Votre main :")
    joueur = joueur % 4
    afficher_cartes(JOUEURS[joueur])
    choix = int(
        input("Choisissez le numéro de la carte que vous voulez jouer : ")) - 1
```

```
    carte_jouee = JOUEURS[joueur].pop(choix)

    if couleur_demandee is None:
        couleur_demandee = carte_jouee[1]

    if carte_jouee[1] == couleur_demandee:

        print(f"Vous avez suivi avec {carte_jouee[0]} de {carte_jouee[1]}")

    elif carte_jouee[1] == ATOUT:

        print(f"Vous avez coupé avec {carte_jouee[0]} de {carte_jouee[1]}")

    else:

        print(f"Vous avez joué {carte_jouee[0]} de {carte_jouee[1]}")

    pli.append(carte_jouee)

    if carte_jouee[1] == ATOUT:

        if carte_maitre is None or obtenir_points(

                carte_jouee, True) > obtenir_points(carte_maitre, True):
            carte_maitre = carte_jouee

    return carte_maitre, couleur_demandee


# obtenir points


def obtenir_points(carte, atout=False):
    valeur, couleur = carte

    if atout:

        points = POINTS_ATOUT.get(valeur, 0)

    else:

        points = POINTS.get(valeur, 0)

    return points


# Fonction pour afficher cartes d'un joueur

def afficher_cartes(cartes):
    for i, (v, c) in enumerate(cartes, start=1):
        print(f"{i}. {v} de {c}")


def compter_points(pli):
    points = 0

    # print("essai",POINTS['Valet'])

    for card in pli:

        if card[1] in ATOUT:

            # print("toto",card)

            # print("carte",card[0])

            points += POINTS_ATOUT[card[0]]

            # print("points = ", points)

        else:

            # print("toto",card)

            # print("carte",card[0])
```

```
            points += POINTS[card[0]]

            # print("points = ", points)

    return points


# Fonction principale pour jouer une partie de belote

def jouer_belote():
    print("Bienvenue dans la partie de belote !")

    score = [0] * 4

    # Création et mélange du jeu de cartes

    jeu = creer_jeu()

    melanger_cartes(jeu)

    # Distribution des cartes aux joueurs

    global JOUEURS, ATOUT, carte_visible

    donneur = 0

    carte_visible = jeu[20]

    mains = distribuer_cartes_sens_horaire(jeu, donneur)

    afficher_mains(mains)

    preneur = gerer_contrat(donneur, carte_visible)

    ATOUT = carte_visible[1]

    # choisir_atout()

    # Si aucun atout n'est choisi, recommencer une nouvelle partie

    if preneur is None:
        jouer_belote()

        return

    # chaque joueurs à 8 cartes

    JOUEURS = distribuer_cartes_finales_sens_horaire(jeu, donneur, preneur, mains)

    # les joueurs

    # Initialisation des variables

    plis_joues = 0

    carte_maitre = None

    couleur_demandee = None

    # Boucle principale pour jouer les plis

    joueur_actuel = (donneur + 1) % 4

    while plis_joues < 8:

        print("score", score)

        pli = []

        for i in range(4):
            print("qui joue", joueur_actuel)

            carte_maitre, couleur_demandee = jouer_tour(pli, joueur_actuel,

                                        carte_maitre,
```

```
                                couleur_demandee)
            joueur_actuel = (joueur_actuel + 1) % 4
        vainqueur_pli, index_vainqueur = determiner_vainqueur(pli, ATOUT)
        joueur_actuel = index_vainqueur
        print("vainqueur", joueur_actuel)
        point = compter_points(pli)
        print("point = ", point)
        score[joueur_actuel - 1] = point + score[joueur_actuel - 1]
        plis_joues += 1
    print("score final", score)

# Lancement du jeu
jouer_belote()
```

TKinter

main.py

```
import tkinter as tk
from tkinter import messagebox
import subprocess
from PIL import Image, ImageTk
import os
import random

# Définition des noms des cartes et leurs chemins d'accès aux images
cartes = {
    '7 de carreau': 'images/7karo.png',
    '7 de coeur': 'images/7kupa.png',
    '7 de piques': 'images/7pika.png',
    '7 de trèfle': 'images/7spatiq.png',
    '8 de carreau': 'images/8karo.png',
    '8 de cœur': 'images/8kupa.png',
    '8 de pique': 'images/8pika.png',
    '8 de trèfle': 'images/8spatiq.png',
    '9 de carreau': 'images/9karo.png',
    '9 de cœur': 'images/9kupa.png',
    '9 de pique': 'images/9pika.png',
    '9 de trèfle': 'images/9spatiq.png',
    '10 de carreau': 'images/10karo.png',
    '10 de cœur': 'images/10kupa.png',
    '10 de pique': 'images/10pika.png',
    '10 de trèfle': 'images/10spatiq.png',
    'Valet de carreau': 'images/11karo.png',
    'Valet de cœur': 'images/11kupa.png',
    'Valet de pique': 'images/11pika.png',
    'Valet de trèfle': 'images/11spatiq.png',
    'Dame de carreau': 'images/12karo.png',
    'Dame de cœur': 'images/12kupa.png',
    'Dame de pique': 'images/12pika.png',
    'Dame de trèfle': 'images/12spatiq.png',
    'Roi de carreau': 'images/13karo.png',
    'Roi de cœur': 'images/13kupa.png',
    'Roi de pique': 'images/13pika.png',
    'Roi de trèfle': 'images/13spatiq.png',
    'As de carreau': 'images/14karo.png',
    'As de cœur': 'images/14kupa.png',
    'As de pique': 'images/14pika.png',
    'As de trèfle': 'images/14spatiq.png'
}

# Mélanger les cartes
liste_cartes = list(cartes.keys())
random.shuffle(liste_cartes)

# Distribuer les cartes aux joueurs
joueur_nord = liste_cartes[:5]
joueur_sud = liste_cartes[5:10]
joueur_est = liste_cartes[10:15]
joueur_ouest = liste_cartes[15:20]

# Initialiser les scores des équipes
score_equipe_nord_sud = 0
score_equipe_est_ouest = 0

# Variable pour suivre quel joueur doit prendre la carte au milieu
joueur_actuel = "nord"

# Variable pour stocker la carte retournée
carte_retournee = None

# Variable pour suivre le nombre de clics sur le bouton
```

```
nombre_clics = 0


# Fonction pour afficher les cartes d'un joueur
def afficher_cartes_joueur(joueur):
    fenetre = tk.Toplevel()
    fenetre.title("Cartes du joueur")

    for i, carte in enumerate(joueur):
        chemin_image = cartes[carte]
        image = Image.open(chemin_image)
        image = image.resize((96, 130), Image.BILINEAR)
        photo = ImageTk.PhotoImage(image)

        label = tk.Label(fenetre, image=photo)
        label.image = photo
        label.grid(row=0, column=i, padx=5, pady=5)


def mettre_a_jour_scores():
    global score_equipe_nord_sud, score_equipe_est_ouest
    # Mettre à jour les scores selon la logique du jeu


# Variable pour suivre le nombre total de clics sur le bouton "Retourne"
nombre_total_clics_retourne = 0

# Variable pour suivre le nombre de clics sur le bouton "Retourne"
nombre_clics_retourne = 0


# Fonction pour afficher une carte au milieu de l'interface
def afficher_carte_milieu():
    global nombre_clics_retourne
    global nombre_clics
    global joueur_actuel
    global carte_retournee

    if nombre_clics < 2 and nombre_clics_retourne < 2:
        nombre_clics += 1
        nombre_clics_retourne += 1

        fenetre = tk.Toplevel()
        fenetre.title("Carte au milieu")

        # Créer une liste de cartes restantes
        cartes_restantes = [carte for carte in liste_cartes if carte not in joueur_nord +
joueur_sud + joueur_est + joueur_ouest]

        # Sélectionner aléatoirement une carte parmi les cartes restantes
        carte_milieu = random.choice(cartes_restantes)

        carte_retournee = carte_milieu  # Mettre à jour la carte retournée

        chemin_image = cartes[carte_milieu]
        image = Image.open(chemin_image)
        image = image.resize((96, 130), Image.BILINEAR)
        photo = ImageTk.PhotoImage(image)

        label = tk.Label(fenetre, image=photo)
        label.image = photo
        label.pack(padx=10, pady=10)

        # Afficher les boutons "oui" et "non" pour le joueur actuel
        bouton_oui = tk.Button(fenetre, text="Oui", command=lambda: prendre_carte_milieu(True,
fenetre))
        bouton_oui.pack(side=tk.LEFT, padx=5, pady=5)

        bouton_non = tk.Button(fenetre, text="Non", command=lambda:
prendre_carte_milieu(False, fenetre))
        bouton_non.pack(side=tk.LEFT, padx=5, pady=5)
```

```
        if nombre_clics_retourne == 2:
            bouton_carte_milieu.config(state=tk.DISABLED)  # Désactiver le bouton après 2
clics
            if nombre_clics == 2:
                messagebox.showinfo("Fin du jeu temporaire", "Aucun joueur n'a pris la
carte.")  # Afficher un message
                nombre_clics = 0  # Réinitialiser le nombre de clics
                nombre_clics_retourne = 0  # Réinitialiser le nombre de clics sur le bouton
"Retourne"

            if nombre_clics == 2:
                if joueur_actuel == "nord":
                    joueur_actuel = "est"
                elif joueur_actuel == "est":
                    joueur_actuel = "sud"
                elif joueur_actuel == "sud":
                    joueur_actuel = "ouest"
                elif joueur_actuel == "ouest":
                    joueur_actuel = "nord"

                messagebox.showinfo("Fin du jeu temporaire", "Aucun joueur n'a pris la
carte.")  # Afficher un message
                nombre_clics = 0  # Réinitialiser le nombre de clics


def retour_menu():
    # Fonction pour retourner au menu principal
    racine.destroy()
    # Lancer le script menu.py
    subprocess.run(["python", "menu.py"])


# Variable pour suivre le nombre de fois où le bouton "Non" a été cliqué consécutivement
nombre_non_consecutifs = 0


# Fonction pour gérer le choix du joueur sur la carte au milieu
def prendre_carte_milieu(accepter, fenetre):
    global joueur_actuel
    global nombre_clics
    global nombre_non_consecutifs

    if accepter:
        if joueur_actuel == "nord":
            joueur_nord.append(carte_retournee)
        elif joueur_actuel == "est":
            joueur_est.append(carte_retournee)

        bouton_carte_milieu.config(state=tk.DISABLED)  # Désactiver le bouton "Retourne"
        fenetre.destroy()  # Fermer la fenêtre de la carte au milieu
    else:
        nombre_non_consecutifs += 1  # Incrémenter le compteur de non consécutifs

        if nombre_non_consecutifs >= 4:
            fenetre.destroy()  # Fermer la fenêtre de la carte au milieu
            nombre_non_consecutifs = 0  # Réinitialiser le compteur de non consécutifs

        else:
            # Passer au joueur suivant
            if joueur_actuel == "nord":
                joueur_actuel = "est"
            elif joueur_actuel == "est":
                joueur_actuel = "sud"
            elif joueur_actuel == "sud":
                joueur_actuel = "ouest"
            elif joueur_actuel == "ouest":
                joueur_actuel = "nord"

    nombre_clics = 0  # Réinitialiser le nombre de clics
```

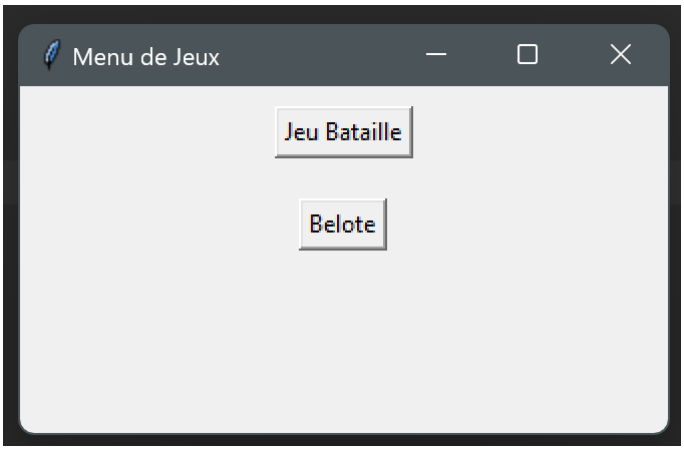
```
# Créer la fenêtre principale
racine = tk.Tk()
racine.title("Jeu de cartes")

# Charger l'image d'arrière-plan
background_image = Image.open("images/verdebelote.jpg")
background_image = background_image.resize((1300, 840), Image.BILINEAR)
background_photo = ImageTk.PhotoImage(background_image)

# Afficher l'arrière-plan
background_label = tk.Label(racine, image=background_photo)
background_label.image = background_photo
background_label.place(x=0, y=0, relwidth=1, relheight=1)

# Boutons pour afficher les cartes de chaque joueur
bouton_nord = tk.Button(racine, text="Joueur Nord", command=lambda:
afficher_cartes_joueur(joueur_nord))
bouton_nord.place(relx=0.5, rely=0.25, anchor="center")

bouton_sud = tk.Button(racine, text="Joueur Sud", command=lambda:
afficher_cartes_joueur(joueur_sud))
bouton_sud.place(relx=0.5, rely=0.75, anchor="center")

bouton_est = tk.Button(racine, text="Joueur Est", command=lambda:
afficher_cartes_joueur(joueur_est))
bouton_est.place(relx=0.75, rely=0.5, anchor="center")

bouton_ouest = tk.Button(racine, text="Joueur Ouest", command=lambda:
afficher_cartes_joueur(joueur_ouest))
bouton_ouest.place(relx=0.25, rely=0.5, anchor="center")

# Bouton pour afficher une carte au milieu
bouton_carte_milieu = tk.Button(racine, text="Retourne", command=afficher_carte_milieu)
bouton_carte_milieu.place(relx=0.5, rely=0.5, anchor="center")

# Bouton pour revenir au menu principal
bouton_menu = tk.Button(racine, text="Menu", command=retour_menu)
bouton_menu.place(relx=0.05, rely=0.05)

racine.mainloop()
```

menu.py

```
import tkinter as tk
import subprocess

def jouer_bataille():
    # Fonction à exécuter lorsque le bouton "Jeu Bataille" est cliqué
    print("Bataille lancé")

def jouer_belote():
    # Fonction à exécuter lorsque le bouton "Belote" est cliqué
    print("Belote lancé")
    # Lancer le script main.py
    subprocess.run(["python", "main.py"])


# Création de la fenêtre principale
root = tk.Tk()
root.title("Menu de Jeux")

# Création des boutons pour les jeux
btn_bataille = tk.Button(root, text="Jeu Bataille", command=jouer_bataille)
btn_bataille.pack(pady=10)
```

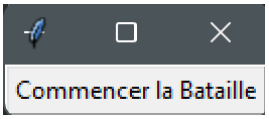
```
btn_belote = tk.Button(root, text="Belote", command=jouer_belote)
btn_belote.pack(pady=10)

# Exécution de la boucle principale
root.mainloop()
```

Image d'une exécution du menu :

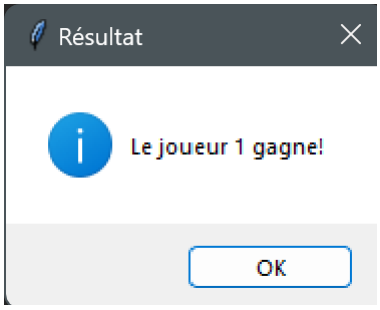


Une fois bouton belote cliqué avec les cartes affichées :

La retourne proposée aux joueurs dans le sens des aiguilles d'une montre :



Affichage jeu Bataille



b. Jeu bataille

```
import tkinter as tk
from tkinter import messagebox
import random
from PIL import Image, ImageTk

# Dictionnaire des chemins d'accès aux images des cartes
chemins_cartes = {
    '7 de Pique': 'images/7pika.png',
    '7 de Coeur': 'images/7kupa.png',
    '7 de Carreau': 'images/7karo.png',
    '7 de Trèfle': 'images/7spatiq.png',
    '8 de Pique': 'images/8pika.png',
    '8 de Coeur': 'images/8kupa.png',
    '8 de Carreau': 'images/8karo.png',
    '8 de Trèfle': 'images/8spatiq.png',
    '9 de Pique': 'images/9pika.png',
    '9 de Coeur': 'images/9kupa.png',
    '9 de Carreau': 'images/9karo.png',
    '9 de Trèfle': 'images/9spatiq.png',
    '10 de Pique': 'images/10pika.png',
    '10 de Coeur': 'images/10kupa.png',
    '10 de Carreau': 'images/10karo.png',
    '10 de Trèfle': 'images/10spatiq.png',
    'Valet de Pique': 'images/11pika.png',
    'Valet de Coeur': 'images/11kupa.png',
    'Valet de Carreau': 'images/11karo.png',
    'Valet de Trèfle': 'images/11spatiq.png',
    'Dame de Pique': 'images/12pika.png',
    'Dame de Coeur': 'images/12kupa.png',
    'Dame de Carreau': 'images/12karo.png',
    'Dame de Trèfle': 'images/12spatiq.png',
    'Roi de Pique': 'images/13pika.png',
    'Roi de Coeur': 'images/13kupa.png',
    'Roi de Carreau': 'images/13karo.png',
    'Roi de Trèfle': 'images/13spatiq.png',
    'As de Pique': 'images/14pika.png',
    'As de Coeur': 'images/14kupa.png',
    'As de Carreau': 'images/14karo.png',
    'As de Trèfle': 'images/14spatiq.png'
}

class BatailleGame:
    def __init__(self, master):
        self.master = master
        self.master.title("Bataille Game")
```

```
        self.start_button = tk.Button(
            self.master, text="Commencer la Bataille", command=self.start_game)
        self.start_button.pack()

    def start_game(self):
        self.game_window = tk.Toplevel(self.master)
        self.game_window.title("Jeu de Bataille")

        self.deck = list(chemins_cartes.keys())
        random.shuffle(self.deck)
        self.player1, self.player2 = self.deck[:len(self.deck)//2], self.deck[len(self.deck)//2:]

        self.canvas = tk.Canvas(self.game_window, width=300, height=200)
        self.canvas.pack()

        self.play_battle()

    def play_battle(self):

            gagnant = 0

            tab= []
        if self.player1 and self.player2:
            carte_j1 = self.player1.pop(0)
            carte_j2 = self.player2.pop(0)
            self.display_card(carte_j1, 50, 100)
            self.display_card(carte_j2, 250, 100)
            valeur_carte_j1 = self.get_card_value(carte_j1)
            valeur_carte_j2 = self.get_card_value(carte_j2)
            if valeur_carte_j1 < valeur_carte_j2:
                messagebox.showinfo("Résultat", "Le joueur 2 gagne!")

                        self. player2.append(carte_j1)

                        self. player2.append(carte_j2)

                        gagnant = 2
            elif valeur_carte_j2 < valeur_carte_j1 :
                messagebox.showinfo("Résultat", "Le joueur 1 gagne!")

                        self. player2.append(carte_j1)

                        self. player2.append(carte_j2)
            else:
                messagebox.showinfo("Résultat", "Bataille!")

                        tab = [carte_j1, carte_j2]

                        tab.append(self.player1.pop(0))

                        tab.append(self.player.pop(0))

                        play.battle(self)
```

```
                if gagnant == 2:
                        for I in tab:
                                self.player2.append(I)
                else:
                        for I in tab:
                                self.player.append(I)


                messagebox.showinfo('Nombre de cartes,f"Joueur 1: {len(self.player1)} cartes
\nJoueur 2: {len(self.player2)} cartes")

    def display_card(self, card, x, y):
        image_path = chemins_cartes[card]
        image = Image.open(image_path)
        image = image.resize((100, 150), Image.BILINEAR)
        photo = ImageTk.PhotoImage(image)
        self.canvas.create_image(x, y, image=photo, anchor=tk.CENTER)
        self.canvas.image = photo

    def get_card_value(self, card):
        valeurs = {
            '7': 7,
            '8': 8,
            '9': 9,
            '10': 10,
            'Valet': 11,
            'Dame': 12,
            'Roi': 13,
            'As': 14
        }
        nom_carte = card.split()[0]
        return valeurs[nom_carte]

def main():
    root = tk.Tk()
    bataille_game = BatailleGame(root)
    root.mainloop()

if __name__ == "__main__":
    main()
```

4. Explication du programme (version 2)

a. La Belote

Ce programme utilise plusieurs fonctionnalités de Python, notamment :

- Les structures de données telles que les listes, les dictionnaires et les tuples pour stocker et manipuler les cartes, les mains des joueurs, etc.
- Les boucles et les conditions pour contrôler le flux du programme et exécuter des actions spécifiques en fonction des situations.
- Les fonctions pour organiser le code en blocs réutilisables et modulaires, ce qui facilite la maintenance et la compréhension du programme.
- Les entrées/sorties standard pour permettre à l'utilisateur d'interagir avec le programme en fournissant des entrées via la console et en affichant des informations pertinentes.

De plus, le programme utilise la bibliothèque standard de Python :

- **random** : Cette bibliothèque est utilisée pour mélanger les cartes dans le jeu, ce qui garantit un mélange aléatoire et équitable.

En ce qui concerne les étapes du programme :

1. Définition des constantes :

- Les constantes définies incluent les couleurs (COULEURS), les valeurs des cartes (VALEURS), les points associés à chaque carte (POINTS), les points spécifiques pour les atouts (POINTS_ATOUT), et le nombre de plis dans une partie (PLIS).

2 . Création et mélange du jeu de cartes :

- La fonction **creer_jeu()** génère un jeu complet de 32 cartes en combinant toutes les valeurs et couleurs.
- La fonction **melanger_cartes(jeu)** mélange le jeu de cartes généré.

3 . Distribution des cartes :

- La fonction **distribuer_cartes_sens_horaire(jeu, donneur)** distribue les cartes dans le sens des aiguilles d'une montre à partir du donneur spécifié.
- La fonction **distribuer_cartes_finales_sens_horaire(jeu, donneur, preneur, mains)** distribue les cartes finales dans le sens des aiguilles d'une montre, en incluant une phase où chaque joueur prend une carte visible.

4. Gestion du contrat :

- La fonction **gerer_contrat(donneur, carte_visible)** permet aux joueurs de décider s'ils veulent prendre l'atout ou passer.

5. Vérification des annonces des joueurs :

- La fonction **verifier_annonces(joueur)** vérifie les annonces de chaque joueur, telles que les carrés et les suites.
- Les fonctions **verifier_carres(cartes)** et **verifier_suites(cartes)** vérifient respectivement s'il y a des carrés ou des suites dans la main d'un joueur.

6. Affichage des annonces :

- La fonction **afficher_annonces(annonces)** affiche les annonces faites par les joueurs.

7. Déroulement d'un tour :

- La fonction **jouer_tour(pli, joueur, carte_maitre, couleur_demandee)** simule le tour d'un joueur, où il choisit une carte à jouer.

8. Détermination du vainqueur du pli :

- La fonction **determiner_vainqueur_pli(pli)** détermine le vainqueur du pli en fonction des règles de la Belote.

9. Fonction principale pour jouer une partie de Belote :

- La fonction **jouer_belote()** orchestre le déroulement de la partie, en initialisant le jeu, en distribuant les cartes, en gérant le contrat, en jouant les plis et en déterminant.

b. TKinter

Pour les explications détaillées, cf. les annotations directement # sur le code.

L'interface graphique permet pour le moment différentes étapes, mais pas encore l'intégralité du jeu. En effet, Nous continuerons à travailler sur le raccord entre le programme (version 2) et la conception de l'interface graphique autour de celui-ci. Ainsi, voici une synthèse de ce que le programme permet de faire :

- L'affichage d'un menu donnant le choix entre **Belote** et **Bataille**
- La distribution des cartes (avec un mélange aléatoire puis une division de liste)
- La création de quatre boutons **« Nord » « Sud » « Est » « Ouest »** qui permettent à un joueur d'afficher ses cartes dans une fenêtre à part (afin de pouvoir la refermer lorsque c'est au tour d'un autre joueur éventuellement, puisque nous n'avons pas encore intégré le mode multijoueur sur différents appareils)
- Un bouton **« retourne »** (pour lequel nous souhaitions présenter initialement une explication un peu plus détaillée dans le cadre d'une présentation orale, qui aura donc lieu à la rentrée comme convenu)
- Le choix de la prise de la retourne avec deux boutons **« oui »** et **« non »**

c. La Bataille

1. Imports :

- `tkinter`: Utilisé pour créer l'interface graphique.
- `messagebox`: Utilisé pour afficher des messages d'information dans des boîtes de dialogue.
- `random`: Utilisé pour mélanger les cartes.
- `PIL`: Utilisé pour manipuler les images, notamment pour afficher les cartes.

2. Dictionnaire des chemins d'accès aux images des cartes :

- chemins_cartes = {...}

Ce dictionnaire contient les chemins d'accès aux images des cartes, où les clés sont les noms des cartes et les valeurs sont les chemins d'accès aux fichiers images correspondants.

3. Classe `BatailleGame` :

- `__init__(self, master)`: Le constructeur de la classe, qui initialise la fenêtre principale du jeu.
- `start_game(self)`: Méthode pour commencer le jeu, elle crée une nouvelle fenêtre pour le jeu de bataille.
- `play_battle(self)`: Méthode pour jouer une bataille, elle compare les cartes des joueurs, affiche les résultats et actualise l'affichage des cartes.

- `display_card(self, card, x, y)`: Méthode pour afficher une carte à une position spécifiée sur le canvas.
- `get_card_value(self, card)`: Méthode pour obtenir la valeur numérique d'une carte.

4. Fonction `main()` :

La fonction principale qui instancie la classe `BatailleGame` et lance l'application.

5. Conditions dans `play_battle(self)`:

- Les joueurs se partagent les cartes.
- Les cartes sont comparées et le résultat est affiché.
- Si une bataille se produit, les cartes sont placées en jeu et une nouvelle bataille est déclenchée récursivement.
- À la fin de chaque bataille, les cartes sont distribuées au vainqueur.
- On affiche le nombre de cartes de chaque joueur

6. Self :

le mot-clé self est crucial car il permet de faire référence à l'instance de la classe BatailleGame à l'intérieur de ses méthodes. Voici comment self est utilisé et pourquoi il est important dans ce contexte :

Accès aux attributs de l'instance : Dans le constructeur __init__, self.master est utilisé pour faire référence à l'instance de la fenêtre principale passée lors de la création de l'objet BatailleGame. Cela permet d'accéder à la fenêtre principale et de la modifier si nécessaire.

Appel des méthodes de l'instance : Dans les méthodes telles que start_game et play_battle, self est utilisé pour appeler d'autres méthodes de l'instance de la classe. Par exemple, dans self.start_game(), self permet d'appeler la méthode start_game de l'instance actuelle de BatailleGame.

Accès aux attributs de la classe : Les attributs de la classe, tels que self.deck, self.player1 et self.player2, sont accessibles et modifiables à l'intérieur des méthodes grâce à self. Cela permet de stocker et de manipuler les données spécifiques à chaque instance de la classe.

Utilisation dans les boucles et les conditions : Dans la méthode play_battle, self est utilisé pour accéder aux listes self.player1 et self.player2 et itérer à travers elles pour jouer une bataille. Les conditions telles que if self.player1 and self.player2 font référence aux attributs de l'instance actuelle.

Utilisation dans la récursion : Dans le cas où une bataille se termine par une bataille, la méthode play_battle est appelée récursivement en utilisant self.play_battle(). Cela permet de répéter le processus de jeu jusqu'à ce qu'une condition de fin soit atteinte.

5. Journal de bord

Jour 1 : mardi 12 mars : début du projet :

Aujourd'hui, nous avons entamé notre projet de développement d'un jeu de Belote en équipe. L'ambiance était bien , avec une dynamique de groupe très positive. Chacun était motivé et prêt à contribuer pour ce projet. Nous avons commencé par discuter de la répartition des rôles dans ce projet et discuté de la manière dont nous allions procéder.

Répartition des Rôles :

- Irene s'est occupée du moteur de jeux, en lui même en python
- Nadine s'est occupée de l'interface graphique, en téléchargeant des images soumises aux contraintes de TKinter (choix d'images .png)
- Avigdor a tenté de rendre le jeu multijoueur

Définition des Objectifs :

Nous avons convenu que notre objectif principal était de créer un jeu de Belote fonctionnel, où les joueurs pourraient jouer à 4, à 2 contre 2 sur 2 ordinateurs différents

Jour 2 : mardi 13 mars :Premiers Défis

Aujourd'hui, nous avons rencontré nos premiers défis. Malgré notre dynamique de groupe et notre bonne répartition des rôles, nous avons réalisé que nous n'étions pas tous sur la même longueur d'onde en ce qui concerne les règles de la Belote. Cela nous a fait perdre un peu de temps au départ, car nous avons dû clarifier certaines règles et nous mettre d'accord sur leur implémentation dans notre projet. De son cote Avigdor en a découvert beaucoup plus sur le multijoueur, nous en parlerons plus en détails dans la 6^{e} partie

Communication et Collaboration :

Cependant, grâce à une bonne communication , nous avons rapidement surmonté cet obstacle. Nous avons passé du temps à discuter des règles, à consulter des sources fiables et à échanger nos points de vue. Cette expérience nous a également permis de mieux comprendre les attentes de chacun et de renforcer notre esprit d'équipe.

Jour 3 : mardi 19 mars :Progression et Cohésion

Aujourd'hui, nous avons repris notre progression avec une nouvelle énergie. Nous avons consolidé notre compréhension des règles de la Belote et avons continué à travailler sur les

différentes parties du jeu. Chaque membre de l'équipe a contribué activement à son domaine de responsabilité, et il y avait une bonne coordination entre nous.

Développement du Jeu :

- Nadine : le début de l'interface graphique a été mis en place, et maintenant je vais développer un menu qui donne le choix entre le jeu bataille et la belote.
- Irene: programme de la belote presque finalise mais certain bug comme pour le choix du vainqueur à chaque plis
- Avigdor, a du abandonner git et revenir à Socket, réécriture du projet de bataille

Jour 4: mercredi 20 mars

Aujourd'hui, nous avons dû travailler avec la présence réduite d'un membre de l'équipe. Malgré cela, nous avons réussi à maintenir un bon rythme de travail et à réaliser des avancées significatives dans le développement de notre jeu de Belote.

Optimisation des Fonctionnalités :

Irène : examination de chaque aspect du jeu, identifiant les zones où des améliorations pouvaient être apportées pour une expérience utilisateur plus fluide et agréable. Que ce soit en réduisant les temps de chargement, en optimisant les algorithmes.

+ travaille sur la réduction de la consommation de mémoire, l'optimisation des temps de réponse et la minimisation des ralentissements.

Jour 5-6 : Consolidation et Tests

Nous avons consacré les deux derniers jours à la consolidation de notre travail et aux tests. Nous avons passé en revue chaque fonctionnalité, corrigé les éventuels bugs et amélioré l'expérience utilisateur. Chaque membre de l'équipe a joué un rôle important dans l'identification et la résolution des problèmes.

Le dernier problème dans notre programme est le numéro qui caractérise le joueur qui peut aller de 0 à 3 ou de 1 à 4 et ce décalage créé quelques problèmes.Pour résoudre ce problème, nous devons ajuster les indices pour qu'ils commencent à partir de 0 plutôt que de 1. Cela nécessitera des modifications dans plusieurs parties du code, notamment dans la détermination du vainqueur du pli puis nous effectuerons des tests approfondis pour nous assurer que la numérotation des joueurs est correctement gérée dans toutes les situations de jeu possibles.

6. Les défis du multijoueur

Il nous est paru évident en fin de première heure de cours que joué à 4 sur un seul ordinateur n'était pas quelque chose de très ergonomique. Ainsi nous avons eu l'idée de faire une sorte de multijoueur local qui fonctionne de tel sorte que 3 ordinateurs se connecte à un ordinateur jouant le rôle de serveur sur lequel sont placés les principaux fichiers sur lequel les machines connectées vont agir et faire des actions. Ainsi, apres quelques recherche j'ai decouvert le module socket.

- **Socket**

Les sockets sont un mécanisme essentiel pour la communication entre applications sur un réseau informatique. Ils fournissent une interface de programmation pour l'échange de données entre des machines distantes ou des processus sur la même machine. Cette communication est basée sur un modèle client-serveur, où un programme client envoie des requêtes à un programme serveur, qui répond en conséquence.

En Python, la création et la gestion des sockets sont réalisées à l'aide du module socket de la bibliothèque standard. Ce module permet de créer des sockets TCP ou UDP, de les lier à une adresse IP et un port, d'écouter les connexions entrantes sur le côté serveur et d'établir des connexions sortantes sur le côté client.

Une fois qu'une connexion est établie entre un client et un serveur, ils peuvent s'échanger des données en utilisant les méthodes send() et recv() pour envoyer et recevoir des données respectivement. Ces données sont généralement transférées sous forme d'octets, mais peuvent être encodées et décodées en chaînes de caractères si nécessaire.

Il est important de noter que les sockets doivent être correctement fermés une fois la communication terminée, afin de libérer les ressources réseau. Cela se fait en appelant la méthode close() sur les objets de socket correspondants, tant du côté client que du côté serveur.

Dans l'ensemble, les sockets offrent une flexibilité et une puissance considérables pour la mise en œuvre de communications réseau dans les applications Python, qu'il s'agisse d'échanger des données entre processus sur la même machine ou de communiquer à travers un réseau local ou Internet.

Le probleme que l'on peut rencontrer avec socket c'est les pare feux.
Lors de l'utilisation du module socket en Python, les pare-feu peuvent poser des problèmes en bloquant certains ports, en filtrant les protocoles ou les adresses IP, en inspectant le contenu des paquets réseau, ou en ayant des règles strictes qui peuvent perturber la communication. Pour résoudre ces problèmes, il est nécessaire de configurer correctement les règles du pare-feu pour autoriser le trafic nécessaire, en ouvrant les ports requis, en autorisant les protocoles pertinents, et en surveillant les journaux du pare-feu pour détecter et ajuster les règles bloquantes. Cependant je ne connaissais pas cette derniere étape je me suis alors tournés vers un autre outil : Git cependant il s'avère que Git n'est qu'un outil de travail collaboratif, permettant de voir et de constater les modifications que fait un appareil A sur un fichier avec un appareil B. Je suis donc revenu à Socket, et crée alors une règle grâce à des personnes rencontre sur discord sur mon ordinateur afin de laisser passer le protocole TCP dans le cas présent.

- **Mise en relation et défi**

Apres avoir compris et mis en place le Socket il s'est avéré que le programme ne permettait pas du tout de jouer avec socket car pas adapter nous avons donc compris que l'utilisation de Socket nécessitait un code 'fait pour ' c'est-à-dire qui devait être tres modulable afin de correspondre à nos attentes. Nous avons donc conclu 2 choses, premièrement, nous devions mettre d'une certaine manière de coté la partie multijoueur de notre jeu et se concentrait sur comment adapter le moteur de jeu et l'interface graphique à un jeu sur 4 ordinateurs. Dans un second temps, finir avant le début des vacances le code du moteur du jeu, et qu'il soit particulièrement clair et lisible puis véritablement comprendre comment adapter socket à ce mode de jeu. Voici ci-dessous le code permettant la creation d'une sorte de messagerie instantanée entre un ordi serveur et un ordi client :

- **Code serveur**

```
> informatique > snt > Projet python > serveur.py > ...
import socket
import threading

def send_response():
    while True:
        response = input("Réponse au client : ")
        client_socket.sendall(response.encode())

def receive_message():
    while True:
        message = client_socket.recv(1024).decode()
        print("Message du client:", message)

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_address = ('172.20.10.3', 12345)
server_socket.bind(server_address)
server_socket.listen(1)

print("Serveur en attente de connexion...")

client_socket, client_address = server_socket.accept()
print("Connexion établie avec", client_address)

send_thread = threading.Thread(target=send_response)
receive_thread = threading.Thread(target=receive_message)

send_thread.start()
receive_thread.start()
```

- **Code client**

client .py > ...

```python
import socket
import threading


def send_message():
    while True:
        message = input("Message au serveur : ")
        client_socket.sendall(message.encode())


def receive_message():
    while True:
        data = client_socket.recv(1024).decode()
        print("Réponse du serveur:", data)


client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_address = ('172.20.10.3', 12345)
client_socket.connect(server_address)

```

```python

send_thread = threading.Thread(target=send_message)
receive_thread = threading.Thread(target=receive_message)

send_thread.start()
receive_thread.start()

```

7. Conclusion

La réalisation de notre projet de développement d'un jeu de Belote a été une expérience enrichissante, mettant en lumière notre capacité à collaborer efficacement en équipe tout en surmontant les obstacles rencontrés tout au long du processus. En tenant compte des critères énoncés dans le rapport, nous pouvons constater les points suivants :

1. Respect des consignes : Nous avons respecté les consignes en établissant un journal de bord détaillé, en présentant les différentes versions du code du jeu Belote, en expliquant le programme dans sa version finale et en intégrant une interface graphique avec Tkinter.
2. Répartition des rôles et objectifs clairs : Dès le début du projet, nous avons défini les rôles de chaque membre de l'équipe et nos objectifs principaux, ce qui nous a permis de travailler de manière organisée et efficace tout au long du processus de développement.
3. Communication et collaboration : Malgré les défis initiaux liés à la compréhension des règles de la Belote, nous avons maintenu une communication ouverte et une collaboration étroite, ce qui nous a aidés à surmonter ces obstacles et à progresser efficacement.
4. Progression et cohésion : Nous avons constaté une progression constante dans le développement du jeu, grâce à une bonne coordination entre les membres de l'équipe et à une répartition équilibrée des tâches. La cohésion de l'équipe a été renforcée tout au long du projet, ce qui a favorisé notre productivité.

Pour aborder un œil plus critique sur le résultat obtenu : Nous n’avons pas pu faire ce que l’on aurait aimé faire. Le code n’est pas complètement fini. Nous pensons avoir besoin des vacances pour finir nos objectifs personnels c’est-à-dire une interface graphique impeccable quelque soit le jeu auquel on joue, et si possible esthétique. Et enfin le graal, la mise en place d’un multijoueur local qui ne peut malheureusement comme nous l’avons vu se faire que si le code est impeccable, modulable et structure. Ainsi nous pensons finir avec à peu près 700lignes – 800 lignes confondues de codes, tout fichier confondue cependant ce nombre de ligne est multipliable par 4 vue que le code est sur 4 ordis en même temps