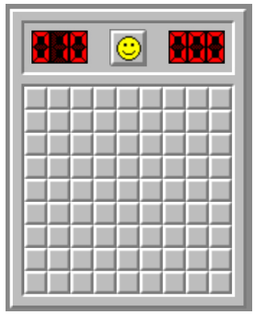
BCPST
1.1 & 1.2

TP d'informatique n° 8

— Projet du démineur —

Pour les trois dernières séances, nous vous proposons de réaliser un jeu de démineur. Le graphisme sera modeste (ce n'est plus de l'informatique mais de l'art) mais l'algorithme doit respecter les règles du jeu et déclarer la victoire ou la défaite.



On peut même imaginer que les plus rapides puissent programmer un début de résolution par un programme Python (ce n'est pas encore une IA !).

1 Les étapes d'un projet

Un projet se mène en plusieurs étapes :

- Choisir la modélisation : choix des informations que l'on veut coder et type de variables pour cela. On décide également des valeurs que l'on va attribuer aux différentes cases du jeu.
- Créer un programme principal qui est avant tout un chef d'orchestre : il ne réalise pas d'opérations complexes mais lance tour à tour plusieurs fonctions qui ont chacune un rôle très simple (une fonction = une action).
- Écrire les fonctions du jeu en réfléchissant bien aux paramètres d'entrée nécessaires et aux variables de sortie. Il est très important de bien tester chaque fonction avant de passer à la rédaction de la suivante.
- Il ne reste qu'à jouer au jeu que vous aurez programmé !

De nombreux choix de modélisations sont possibles. De même que la structure du programme principal peut avoir des formes très variées. Comme nous ne disposons que de 3 semaines nous vous proposons une modélisation et une structure du programme principal.

2 La modélisation du jeu du démineur

Nous vous proposons la modélisation suivante :

- Le plateau de jeu sera modélisé par un tableau numpy carré (de taille `(n,n)`)
- En réalité il y a deux tableaux : `Tc`, le **T**ableau **c**aché, qui contient l'emplacement des bombes et `Ta`, le **T**ableau **a**ffiché, que verra le joueur et sur lequel il cliquera avec la souris.
- Les bombes seront codées par la valeur 9 (dans `Tc`), les cases vides par la valeur 0, les cases non découvertes par la valeur -1 (dans `Ta`) et les autres cases contiendront le nombre de bombes dans les cases voisines.

3 La structure du projet

Commençons par l'importation des modules dont nous aurons besoin :

```python
from math import *
import random as rd
import numpy as np
import matplotlib.pyplot as plt
```

Attention `randint(a,b)` renvoie un entier aléatoire dans l'intervalle [a,b] avec b inclu. C'est une exception : en général dans Python les bornes supérieures des intervalles sont exclues (`range`, `L[2:5]`,...).

3.1 Le programme principal

Nous vous proposons la structure suivante pour le programme principal :

```python
def demineur(n,n_b):
    """ Entrée : taille du plateau de jeu : n et nb de bombes cachées : n_b
    Attention il faut n_b < n**2"""
    Tc = placement_bombes(n,n_b)
    print(Tc) # on triche ! A enlever à la fin.
    Tc = remplissage(Tc)
    Ta = creation_affichage(n)
    affichage(Ta)
    while fin_de_jeu(...) == False:
        l,c = clic(Ta)
        Ta[l,c] = Tc[l,c]
        # if Tc[l,c] == 0:
            # Ta = tache_huile(...) # à faire en dernier
        affichage(Ta)
    if Tc[l,c] == 9:
        plt.title('Perdu...')
        return 'perdu...'
    else:
        plt.title('Gagné !')
        return 'gagné !'
```

On voit que le programme principal ne contient pas beaucoup d'instructions : une boucle `while` et un `if`. Par contre il lance plusieurs fonctions et récupère les variables de sortie pour alimenter les fonctions qui suivent.

3.2 Présentation des fonctions du programme principal

- La fonction `placement_bombes` renvoie un tableau numpy de dimension `(n,n)` initialement rempli de 0 et y place aléatoirement `n_b` bombes qui sont des valeurs 9.
- La fonction `remplissage` remplace les 0 (cases vides) par le nombre de bombes voisines (entre 0 et 8 puisqu'on regarde aussi en diagonale). Cette fonction utilisera une fonction `decompte_bombes_voisines` qui renvoie le nombre de bombes voisines d'une case donnée en faisant attention aux cases du bord.
- La fonction `creation_affichage` renvoie un tableau numpy de dimension `(n,n)` rempli de -1 (cases non découvertes).
- La fonction `fin_de_jeu` dont les paramètres d'entrée seront à choisir, renvoie un booléen : `True` si la partie est finie et `False` sinon. Attention : il y a deux raisons pour lesquelles la partie est terminée : le joueur a découvert une bombe (perdu !) ou il ne reste plus de cases à découvrir à part les bombes (c'est la victoire !).
- La fonction `clic` récupère les coordonnées cartésiennes du clic sur la figure et renvoie le numéro de ligne et de colonne de la case choisie par le joueur.
- La fonction `affichage` crée la figure associée au tableau `Ta`.
- La fonction `tache_huile` sera programmée en dernier et assure que si le joueur clique sur une case contenant un 0, toutes les cases contigües contenant un 0 sont également découvertes. Les cases contiguës à celles contenant un zéro sont également découvertes. Elle renvoie le tableau `Ta` modifié.

3.3 Présentation de la fonction clic

Le fonction `clic` doit renvoyer les coordonnées (ligne, colonne) de la case sur laquelle le joueur a cliqué avec la souris. Elle utilise la fonction `plt.ginput` dont voici quelques spécifications :

- `plt.ginput(n,timeout)` attend que l'utilisateur réalise `n` clics sur la figure (il faut bien sûr qu'une figure soit déjà ouverte) en une durée inférieure à `timeout`,
  - `n` est un entier qui vaut 1 par défaut,
  - `timeout` est un entier qui vaut 30 par défaut (30 secondes). Si `timeout` vaut 0 alors le temps de saisie n'est pas limité.
- en sortie, la fonction renvoie une liste de tuple `(x,y)` correspondant aux coordonnées de chaque clic,
- le clic est repéré en coordonnées cartésiennes : l'axe des `x` est horizontal et correspond donc au numéro de colonne de la case visée. Il faut arrondir à l'entier le plus proche : `c = int(round(x))` et `l = int(round(y))`.

D'où le code proposé pour l'utilisation de `plt.ginput` :

```
def clic(Ta):
    [(x,y)] = plt.ginput(1,30)
    l,c = int(round(y)),int(round(x))
    return (l,c)
```

Il faut compléter la fonction pour que le clic ne soit pas validé (on attend un autre clic) si la case choisie est déjà découverte.

3.4 Présentation de la fonction affichage

La fonction `affichage` est en partie écrite. Elle vous permettra de tester les autres fonctions au fur et à mesure de leur écriture.

```
def affichage(Ta):
    palette = plt.matplotlib.colors.ListedColormap(['black','white','cyan','blue','green','
        greenyellow','yellow','orange','tomato','deeppink', 'red'])
    n,n=np.shape(Ta)
    plt.close()
    plt.imshow(Ta,cmap=palette,clim=(-1,9))
    for i in range(n+1):
        plt.plot([-0.5,n-0.5],[-0.5+i,-0.5+i],'w',lw=2)
        plt.plot([-0.5+i,-0.5+i],[-0.5,n-0.5],'w',lw=2)
    plt.show()
```

La fonction affiche le tableau `Ta` avec la carte de couleurs nommée `palette` définie par nos soins et des valeurs limites comprises entre -1 et 9.
La boucle `for` permet de tracer des traits horizontaux et verticaux afin de délimiter les cases. Comme le code de couleur des cases n'est pas connu du joueur, il vous reste à ajouter l'affichage du nombre de bombes voisines dans chaque case découverte (sauf pour les cases n'ayant pas de bombes voisines).
La fonction `plt.text(x,y,'2')` affiche le texte '2' au point de coordonnées cartésiennes `(x,y)` dans la figure ouverte. Attention, l'axe des `x` est horizontal et correspond donc au numéro de colonne de la case visée. On rappelle que la fonction `str(2024)` transforme l'entier 2024 en une chaine de caractères `'2024'`.

4 Les étapes à suivre

Voici le travail que vous avez à faire pour mener à bien le projet :

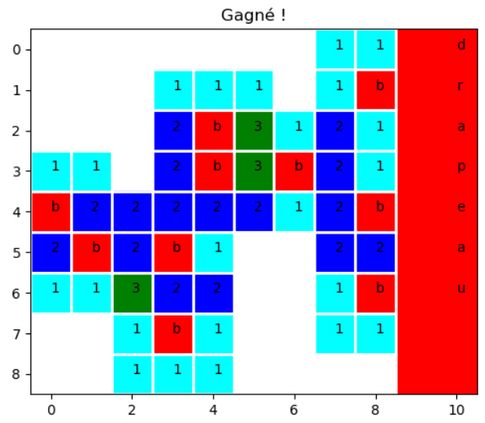
1. Récupérer le fichier `Projet démineur.py` sur cahier de prépa.
2. Écrire la fonction `placement_bombes` et la tester correctement. Attention, il ne faut pas qu'il y ait plusieurs bombes sur la même case. Lors de la création d'un tableau (fonctions `np.zeros` ou `np.ones`) les valeurs sont par défaut de type `float`. Afin que ce soit des entiers (pour l'affichage sur la figure) il faut ajouter l'argument `dtype=int` lors de la création du tableau. Voici un exemple obtenu en lançant `placement_bombes(3,5)` :

```
>>> placement_bombes(3,5)
array([[9, 0, 9],
       [0, 0, 9],
       [0, 9, 9]])
```

3. Écrire la fonction `remplissage` et la tester correctement.
   Voici un exemple pour le lancement de `remplissage(placement_bombes(4,4))` :

```
>>> remplissage(placement_bombes(4,4))
array([[0, 1, 1, 1],
       [1, 2, 9, 2],
       [2, 9, 4, 9],
       [2, 9, 3, 1]])
```

4. Écrire la fonction `creation_affichage`.
5. Écrire la fonction `fin_de_jeu` et la tester correctement.
6. Terminer la fonction `clic` et la tester correctement (cf. **3.3**).
7. Terminer la fonction `affichage` et la tester correctement (cf. **3.3**).
8. Compléter la ligne du `while` du programme principal et la modifier en exploitant le fait que la fonction `fin_de_jeu` renvoie un booléen. On pourra utiliser l'opérateur logique `not`.
9. Jouer un peu pour vérifier que tout fonctionne correctement.
10. Pour les plus rapides écrire la fonction `tache_huile`.
11. Ajouter sur la figure une case qui sert à désigner les emplacements supposés des bombes (l'équivalent du clic droit dans le jeu en ligne). Il faut alors modifier plusieurs fonctions pour arriver à intégrer cette option de jeu.



12. Proposer une résolution du démineur par Python.