Récursivité: corrigé du TP

I. Algorithme de Casteljau pour le tracé de courbes de Bézier

Question 1.

```
def binom(n,p):
    return factorial(n)//factorial(p)//factorial(n-p)
```

Question 2.

```
def bernstein(n,i,t):
    return binom(n,i)*t**i*(1-t)**(n-i)
```

Question 3. Pour chaque t, on calcule le point $M(t)$. Pour faire ça, il est plus simple de convertir les points de `P` en tableau Numpy. Pour faire la somme $\sum_{i=0}^{n} B_{n,i}(t)P_i$, on peut initialiser une variable à 0, à `np.zeros(2)` ou à `np.array([0., 0.])` : le résultat sera le même (dans le premier cas, additionner un tableau Numpy et un scalaire donne un tableau Numpy ou le scalaire a été additionné à toutes les entrées, si c'est zéro c'est une copie du tableau).

```
def bezier(P):
    L=[np.array(x) for x in P]
    T=np.linspace(0,1,1000)
    n=len(P)-1
    C=[]
    for t in T:
        M=0
        for i in range(n+1):
            M+=bernstein(n,i,t)*L[i]
        C.append(M)
    return C
```

Question 4. Par exemple :

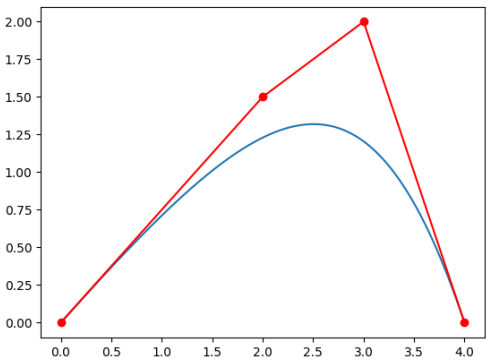
```
def trace_ligne_brisee(L):
    X=[u[0] for u in L]
    Y=[u[1] for u in L]
    plt.plot(X,Y)
```

Question 5. Une autre fonction (les points avec un gros rond rouge) :

```
def trace_ligne_brisee2(L):
    X=[u[0] for u in L]
    Y=[u[1] for u in L]
    plt.plot(X,Y,'ro-')
```

Le code suivant produit la figure de droite (la courbe est bleue, les points reliés en rouge).

```
trace_ligne_brisee(bezier([(0,0), (2,1.5), (3,2), (4,0)]))
trace_ligne_brisee2(P)
plt.show()
```



Question 6. Avec `p` et `q` des tableaux Numpy.

```
def milieu(p,q):
    return (p+q)/2
```

Question 7.

```
def etape_casteljau(P):
    p0, p1, p2, p3=P
    m=milieu
    M=m(p1, p2)
    a1=m(p0, p1)
    a2=m(a1, M)
    b2=m(p2, p3)
    b1=m(M, b2)
    a3=m(a2, b1)
    return [p0, a1, a2, a3], [a3, b1, b2, p3]
```

Question 8. D'abord une fonction donnant la norme d'un vecteur :

```
def norme(v):
    return sqrt(v[0]**2+v[1]**2)
```

Condition d'arrêt : les 3 distances entre points successifs sont inférieures à 0.1.

```
def cond_arret(P):
    for i in range(len(P)-1):
        if norme(P[i]-P[i+1])>=0.1:
            return False
    return True
```

Enfin, l'algorithme de Casteljau :

```
def bezier_casteljau(P):
    P=[np.array(x) for x in P]
    if cond_arret(P):
        trace_ligne_brisee(P)
    else:
        a,b=etape_casteljau(P)
        bezier_casteljau(a)
        bezier_casteljau(b)
```

II. Résolution de Sudoku par backtracking

Question 1. *chiffres_ ligne*. Les chiffres non nuls sur la ligne i.

```
def chiffres_ligne(L,i):
    return [x for x in L[i] if x>0]
```

Question 2. *chiffres_ colonne*. Les chiffres non nuls sur la colonne i.

```
def chiffres_colonne(L,j):
    T=[]
    for i in range(9):
        if L[i][j]>0:
            T.append(L[i][j])
    return T
```

Question 3. *chiffres_ bloc*. Les chiffres non nuls dans le bloc contenant la case (i, j).

```
def chiffres_bloc(L,i, j):
    a,b=3*(i//3), 3*(j//3) #indice de la case en haut à gauche du bloc
    T=[]
    for k in range(3):
        for p in range(3):
            x=L[a+k][b+p]
            if x>0:
                T.append(x)
    return T
```

Question 4. On en déduit une liste de chiffres que l'on ne peut pas écrire en case (i, j) (avec doublons éventuels) :

```
def chiffres_conflit(L,i,j):
    return chiffres_ligne(L,i)+chiffres_colonne(L,j)+chiffres_bloc(L,i, j)
```

Question 5. On ne traite pas à part le cas de la case $(8, 8)$ (celle en bas à droite), cela nous servira pour le cas de base dans la question suivante.

```
def case_suivante(i,j):
    if j==8:
        return i+1,0
    else:
        return i,j+1
```

Question 6.

```
def solution_sudoku(L):
    def aux(i,j):
        if i==9:
            return True
        elif L[i][j]>0:
            a,b=case_suivante(i,j)
            return aux(a,b)
        else:
            conflit=chiffres_conflit(L,i,j)
            a,b=case_suivante(i,j)
            for k in range(1,10):
                if k not in conflit:
                    L[i][j]=k
                    x=aux(a,b)
                    if x:
                        return True
            L[i][j]=0
            return False
    return aux(0,0)
```

Explications :

- La fonction `aux(i,j)` a la spécification suivante :
  - si les chiffres déja rentrés dans les cases précédentes permettent d'obtenir une solution au sudoku, l'appel `aux(i,j)` remplit entièrement la grille à partir de la case (i, j), et renvoie `True`.
  - sinon, la fonction laisse les cases à partir de (i, j) inchangées (remplies pour les conditions initiales, zéro pour les autres cases)

  L'appel initial est `aux(0,0)`, on détaille tous les cas ensuite.
- le cas $i = 9$ correspond au cas où on a réussi à remplir la grille et arriver au bout, la fonction renvoie `True`.
- le cas `L[i][j]` > 0 correspond à une donnée initiale : on appelle simplement `aux` sur la case suivante, et le retour de `aux(i,j)` est le même.
- le dernier cas est celui d'une case vide (il y a un zéro en (i, j)). On calcule les chiffres que l'on ne peut inscrire en (i, j) parce qu'ils seraient en conflit avec les précédentes hypothèses ou les données de la grille. Pour chaque chiffre que l'on peut inscrire, on le met en case (i, j), et on fait un appel récursif sur la case suivante.
  - Si un appel réussit, on laisse le chiffre en (i, j) et on renvoie `True`.
  - Si tous les appels ont raté, on efface le dernier chiffre que l'on a essayé en `(i,j)` (on remet 0 !), et on renvoie `False`.