MPSI 1 : D.S. 1 d'Informatique Commune (1h15)

Tous les appareils électroniques sont interdits. Les deux parties sont complètement indépendantes (le début du 2 est plus facile que la fin du 1)).
Sauf mention du contraire, on n'utilisera pas de module comme `math`. Les *seules* méthodes sur les listes autorisées sont : l'accès à l'entrée `i` de `L` via `L[i]`, l'ajout à `L` d'une entrée ayant la valeur de `a` via `L.append(a)`, l'accès à la longueur de la liste via `len(L)`.

1 Autour du calcul des décimales du nombre e :

a) Ecrire une fonction `factorielle(n)` qui prend en argument un entier naturel `n` et renvoie la valeur de $n!$.

b) Justifier le comportement de votre fonction `factorielle` lorsqu'on lui passe l'argument 0, c'est-à-dire lorsqu'on fait l'appel : `factorielle(0)`.

c) Ecrire une fonction `SommeListe(L)` qui prend en argument une liste $\texttt{L} = [\texttt{a}_0, \ldots, \texttt{a}_{n-1}]$ dont les entrées sont supposées être des nombres, et qui renvoie la valeur de la somme : $\sum_{i=0}^{n-1} a_i$.

Exemple d'utilisation de SommeListe :

```
>>> L=[1,5,4]
>>> SommeListe(L)
10
```

d) Ecrire une fonction `ListInverseFact(n)` qui prend un argument un entier naturel `n` et retourne la liste (de longueur $n+1$) donnant les valeurs flottantes de $[1/0!, 1/1!, 1/2!, \ldots, 1/n!]$
Exemple :

```
>>>ListInverseFact(4)
[1.0, 1.0, 0.5, 0.16666666666666666, 0.041666666666666664]
```

e) On montrera (chapitre F de maths) que la suite (u_n) définie par $\forall\, n \in \mathbb{N}$, $u_n = \dfrac{1}{0!} + \dfrac{1}{1!} + \cdots + \dfrac{1}{n!}$ converge vers $e = \exp(1)$.
Ecrire une fonction `approxiE1(n)` qui prend comme argument un entier n et renvoie la valeur (flottante) de u_n correspondante, en utilisant les deux fonctions précédemment créées. Le code de cette fonction ne dépassera pas deux lignes.

f) Bien que, pour le calcul numérique de `e` avec des flottants, la fonction `approxiE1` précédente soit largement assez efficace, du point de vue strictement informatique elle présente au moins deux défauts :

- elle stocke les $1/k!$ dans un tableau (une liste python), ce qui utilise de la mémoire inutilement
- lorsqu'on l'appelle pour un entier n, elle calcule successivement $1!, 2!, \ldots k!, \ldots, n!$ en repartant à chaque fois du début pour le calcul de chaque factorielle, alors que, lorsqu'on connaît la valeur de $k!$ il suffit de la multiplier par $(k+1)$ pour avoir celle de $(k+1)!$.

Ecrire une autre fonction `approxiE2(n)` qui renvoie encore la valeur de u_n mais en éliminant les deux défauts précédents (elle n'utilisera donc pas les fonctions précédentes).

g) La différence de temps de calcul entre le deux programmes précédents n'est pas significative pour les petites valeurs de `n` qui suffisent pour avoir les seize décimales de `e` significatives dans l'écriture en flottant (obtenues à partir de `n=17`) : `2.718281828459045`
Cependant, si on veut connaître davantage de décimales exactes de `e`, on peut ne pas utiliser le type flottant, et calculer à l'aide du type `Fraction` du module `fractions`.
(i) **Question :** Quelle commande rentrer pour importer tout le contenu du module `fractions` ?
On rappelle l'utilisation de ce module avec quelques exemples :

```
>>> F=Fraction(1,3)
>>> G=Fraction(1,4)
>>> F+G
Fraction(7, 12)
>>> H=F+G
>>> H.numerator
7
>>> H.denominator
12
```

(ii) **Question :** Modifier votre fonction `approxiE2` en une fonction `approxiE2Frac` qui renvoie cette fois son résultat sous la forme d'une `Fraction`. Par exemple, on aura :

```
>>> approxiE2Frac(3)
Fraction(8, 3)
```

h) Pour `F=approxiE2Frac(100)`, on a :

```
>>> F
Fraction(266690570578313737330634132288070236461240278868834697744597737 1,
 9810997807004315497939551021311215756250852119019520000000000000)
```

On démontrera en maths que pour tout $n \in \mathbb{N}^*$, $u_n < e < u_n + \dfrac{1}{n.n!}$.

Question : Sachant qu'il est facile de calculer que $10^{157} < 100! < 10^{158}$, comment calculer *de manière exacte à l'aide de calcul sur les entiers* les 159 premiers chiffres de l'écriture du nombre e, à partir de la fraction `F`, ci-dessous ?

2 Exemple d'algorithme glouton pour le rendu de monnaie :

On veut programmer une caisse automatique pour qu'elle rende la monnaie avec le *nombre minimal* de pièces et de billets.

On se donne donc une liste de valeurs de billets ou de pièces, par exemple en euros :
`liste=[200,100,50,20,10,5,2,1]`

Une méthode possible est d'utiliser *un algorithme glouton* [1] dont l'idée est la suivante : imaginons qu'on doive rendre 47 euros. On commence par chercher la pièce ou le billet de la plus grande valeur possible inférieure à égale à la somme à rendre, donc ici 20, on déduit cette valeur de la somme à rendre qui devient 27, et on recommence jusqu'à obtenir une somme nulle. On obtiendra une liste : `monnaie =[20,20,5,2]` dont il n'est pas difficile de montrer qu'elle est ici optimale car le nombre total de solutions possibles n'est pas très grand.

a) Pour la fonction de rendu de monnaie, nous utiliserons seulement des listes écrites dans l'ordre décroissant. Ecrire une fonction `verifieDecroissant(L)` qui prend en argument une liste `L` et vérifie que les entrées de `liste` sont bien écrites dans l'ordre décroissant (au sens large) : cette fonction retourne un *booléen*.

b) Ecrire une fonction`MonMax(L)` qui prend en argument une liste `L` et retourne un couple `M,imax` où `M` est le maximum des valeurs de `L` et `imax` est un indice tel que `L[imax]` donne cette valeur maximum.

c) Ecrire une fonction `MaxEnTete(L)` qui ne retourne rien mais modifie la liste `L` en échangeant la valeur apparaissant en `L[0]` et la valeur `L[imax]` (avec les notations du b)).

d) Ecrire une fonction Python `rendre(valeur,liste)` qui prend en argument un entier `valeur` et une liste d'entiers `liste` (supposée rangée dans l'ordre décroissant) et applique l'algorithme glouton décrit ci-dessus pour renvoyer une liste qu'on notera `monnaie` décomposant l'entier `valeur` à l'aide des éléments de `liste` comme dans l'exemple précédent.

e) En supposant que l'on passe à la fonction `rendre` les arguments :
`valeur=63` et `liste=[200,100,50,20,2,1]`, (on n'a pas de billets de 5 ni de 10), quelle sera la liste `monnaie` renvoyée par la fonction ?
Montrer que cette liste `monnaie` n'est *pas* optimale.

1. D'une manière générale un algorithme glouton fait toujours le choix qui lui semble le meilleur sur le moment