École Nationale d'Économie Appliquée et de Management.
(ENEAM)

Section : ISE1 **Année 2022/2023**
Travaux pratiques d'algorithmique / programmation **Durée : 02 semaines**
Enseignant : KANFON Y. Credo Auguste, ISE **30/01/2023 à 15h 00**

Pour tous les exercices, les algorithmes seront écrits en utilisant la notation algorithmique et le langage python.

Exercice 1 : Diviseurs d'un nombre

1. Écrire une procédure *calcule_diviseurs(n :entier, var diviseurs Tab(1 .. nmax) d'entier, var d : entier)* qui identifie tous les diviseurs de l'entier *n* (y compris *n* lui-même) et les range dans le tableau diviseurs, dans l'ordre croissant. La procédure renvoie le nombre de diviseurs identifiés, grâce au paramètre *d*. On supposera que $n \geq 2$.
2. Pour tout entier naturel *n*, on note *s(n)* la somme des diviseurs propres de *n*, i.e. la somme de tous ses diviseurs, sauf *n* lui-même. Ecrire une fonction *addition_diviseurs_propre(n : entier)* qui calcule la somme des diviseurs propres de l'entier *n*.
3. On notera *σ(n)* la somme des diviseurs de n, i.e. la somme de tous ses diviseurs (y compris n lui-même). Ecrire une fonction *addition_diviseurs (n : entier)* qui calcule la somme des diviseurs de l'entier n. Vous devez utiliser la fonction *addition_diviseurs_propre*.
4. On appelle **abondance** d'un nombre n≥2 la valeur *a(n)=s(n)-n*. Ecrire une fonction *calcul_abondance(n :entier)* qui calcule l'abondance du nombre passé en paramètre. Vous devez utiliser la fonction *addition_diviseurs_propre*.
5. Un entier n≥2 est dit **parfait** s'il est égal à la somme de ses diviseurs propres *s(n)*, i.e. : *n=s(n)*. Autrement dit, un entier est parfait si son abondance est nulle, i.e. : *a(n)=0*. Écrivez une fonction *est_parfait(n : entier)* qui renvoie 1 si l'entier n est parfait, et 0 sinon. Vous devez utiliser la fonction *calcul_abondance*.
6. Deux entiers n,p ≥2 sont dits **amicaux** lorsque la somme des diviseurs propres de chacun des deux est égal à l'autre nombre, c'est-à-dire s'ils vérifient à la fois *s(n)=p* et *s(p)=n*. Ecrire une fonction *sont_amicaux(n,p : entier)* qui retourne 1 si les entiers n et p sont amicaux, et 0 sinon.
7. Un entier n≥2 est dit **sublime** lorsque le nombre de ses diviseurs et la somme de ses diviseurs sont tous les deux des nombres parfaits. Écrivez une fonction *est_sublime(n : entier)* qui détermine si le nombre n est sublime. La fonction renvoie 1 si c'est le cas, et 0 sinon.
8. On appelle **nombre abondant** un nombre $n \geq 2$ dont l'abondance est strictement positive : *a(n)>0*. Au contraire, on appelle **nombre déficient** un nombre dont l'abondance est strictement négative : *a(n)<0*. Ecrire les fonctions *est_abondant(n : entier)* et *est_deficient(n : entier)* qui indiquent si le nombre n passé en paramètre est respectivement abondant ou déficient. Chaque fonction doit renvoyer 1 si c'est le cas (nombre abondant ou déficient) et 0 si ce n'est pas le cas.
9. Écrivez une procedure *affiche_nombres(max ,mode : entier)* qui reçoit en paramètres une valeur maximale *max* et un mode de fonctionnement *mode*. Ce mode a trois valeurs possibles : *-1, 0* et *1*. En fonction du mode, la fonction doit réaliser l'une des trois tâches suivantes :
    - Pour -1 : afficher tous les entiers déficients compris entre 2 et max ;
    - Pour 0 : afficher tous les entiers parfaits compris entre 2 et max ;
    - Pour 1 : afficher tous les entiers abondants compris entre 2 et max.

Exercice 2 : anagrammes

Ecrire un programme qui détermine si deux mots saisis au clavier sont des anagrammes. Deux mots sont des anagrammes s'ils contiennent exactement les mêmes lettres (mêmes lettres et même nombre d'occurrences de chaque lettre).

Exemple : GARE et RAGE, NICHE et CHIEN. Par contre SALLE n'est pas une anagramme de LASSE. On note que le programme s'arrête dès qu'une lettre du premier mot n'est pas trouvée dans le deuxième. Le résultat du programme doit être affiché à l'écran.

Exercice 3 : Convergence d'une suite de Newton

Une valeur approchée de $\sqrt[3]{A}$ peut-être donnée par la formule de récurrence suivante :

$$x_{i+1} = \frac{1}{3}\left(2x_i + \frac{A}{x_i^2}\right) \; avec \; x_0 = 1 \qquad \text{Test d'arrêt : } |x_i - x_{i-1}| < \varepsilon$$

Ecrire un programme permettant de saisir une valeur *A* avec la précision *epsilon* souhaitée et qui permet de calculer et d'afficher les termes successifs x_i jusqu'à ce que le test d'arrêt soit vérifié.

Exercice 4 : Calcul d'une intégrale

Soit la fonction $f(x) = x^3$. sur un intervalle [a, b]. On rappelle que la somme de Darboux de f sur [a, b] avec une subdivision de pas 1/n est définie par :

$$S_n = \frac{b-a}{n}\sum_{k=1}^{n} f\left(a + k\frac{b-a}{n}\right) \qquad \text{et} \qquad I(f) = \int_a^b f(x)dx = \frac{b^4-a^4}{4}$$

Ecrire un programme qui, pour les bornes *a* et *b* donnée, permet de :

- Calculer et afficher $S_n(f)$ pour *n* donné.
- Déterminer *n* de façon à avoir $|S_n(f) - I(f)| < \varepsilon$ avec ε donné

NB : pour ce problème, on peut utiliser un opérateur puissance noté ** (exemple : x^3 = x**3).

Exercice 5 : Rotation dans un tableau

Soit un tableau de *n* caractères. On appelle rotation à gauche de *k* caractères, l'opération qui consiste à placer les *k* premiers caractères du tableau aux *k* dernières positions. Par exemple, pour *n* = 8 et *k* = 3, le tableau

A	B	C	D	E	F	G	H

Devient, après rotation

D	E	F	G	H	A	B	C

Ecrire un programme permettant de

- saisir un tableau de 100 caractères maximum (le nombre effectif n de caractères devra être saisi par l'utilisateur)
- saisir un nombre entier k (0 < k < n).
- effectuer la rotation à gauche de k positions sans utiliser de tableau intermédiaire.
- Afficher le tableau après cette rotation.

Exercice 6 : la Divine proportion !

La suite de Fibonacci est célèbre non seulement parce que chacun des chiffres qui la composent est la somme des deux précédents mais aussi parce que les quotients entre deux chiffres adjacents forment une suite qui converge vers un nombre appelé *nombre d'or* en raison de ces propriétés remarquables au point que les savants de l'Antiquité l'on appelé *la Divine proportion* !

Soit les suites (*u*) et (*v*) définies par $u_1 = 1$, $u_2 = 2$, $u_n = u_{n-1} + u_{n-2}$ et $v_i = \frac{u_i}{u_{i-1}}$.

La suite (*v*) converge vers le nombre d'or.

On supposera que le $n^{ième}$ terme de la suite (v) soit v_n, donne une valeur approchée du nombre d'or avec une précision e, dès que $|v_n - v_{n-1}| < e$

Ecrire un programme qui, au choix de l'utilisateur, permet soit de :

- Calculer et afficher v_n pour n donné.
- Déterminer n de façon à avoir $|v_n - v_{n-1}| < e$ avec une précision e donnée.

Exercice 7 : Manipulation de polynômes

Soit un polynôme $P(x)$ de degré n, défini par ses coefficients $a_0, a_2, a_3, \ldots, a_n$:

$P(x) = a_0 + a_1x^1 + a_2x^2 + \ldots + a_nx^n$

Ecrire un algorithme permettant selon le choix, de calculer la valeur du polynôme en un point x_0 lu au clavier ou de construire la table des valeurs de $P(x)$ pour x variant de b à c (b et c étant des nombres entiers).

Le programme devra permettre à l'utilisateur d'entrer son choix (1 ou 2). Selon le choix effectué, ce dernier devra alors entrer soit la valeur x_0, soit les valeurs b et c.

Pour le calcul des valeurs du polynôme en un point x_0, on pourra utiliser le schéma de Horner suivant :

$P(x_0) =((((\ldots(a_n x_0 + a_{n-1}) x_0 + a_{n-2}) x_0 + a_{n-3})\ldots)x_0 + a_0)$

On utilisera un tableau COEFF pour stocker les coefficients du polynôme.

Exercice 8 : Séquence

Ecrire un programme pour :

Lire une séquence de longueur variable d'éléments entiers non nuls $e_1, e_2, \ldots, e_{i-1}, e_i, \ldots$

Déterminer et afficher le nombre d'éléments, leur moyenne, leur minimum, leur maximum, le pourcentage d'éléments positifs, le pourcentage d'éléments négatifs et le rang i ($i \geq 2$) tel que $|e_{i-1}-e_i|$ soit maximal. ***On s'interdira d'utiliser un tableau dans ce problème.***

Exercice 9 : Tri par permutation simple ou "tri bulle"

On désigne par tri l'opération consistant à ordonner un ensemble d'éléments en fonction de clés sur lesquelles est définie une relation d'ordre. Il existe plusieurs méthodes de tri des éléments d'un tableau parmi lesquelles la méthode de tri par permutation simple dite de la bulle. Le sobriquet de tri bulle vient d'une interprétation imagée selon laquelle l'algorithme fait "remonter" petit à petit les éléments "légers" vers la surface.

L'algorithme de tri bulle se définit ainsi (n représentant le nombre d'éléments du tableau) :

On parcourt l'ensemble du tableau, depuis sa fin jusqu'à son début, en comparant deux éléments consécutifs, en les inversant s'ils sont mal classés. On se retrouve ainsi avec le plus petit élément placé en tête du tableau.

On renouvelle une telle opération (dite "passe") avec les $n-1$ éléments restants, puis avec les $n-2$ éléments restants, ainsi de suite … jusqu'à ce que :

- soit l'avant-dernier élément ait été classé (le dernier étant alors obligatoirement à sa place),
- soit qu'aucune permutation n'ait eu lieu pendant la dernière passe (ce qui prouve alors que l'ensemble du tableau est convenablement ordonné).

Travail à faire

Ecrire un programme permettant de réaliser le tri par valeurs croissantes d'un tableau d'entiers, en utilisant l'algorithme du tri bulle. Le programme affichera tous les résultats intermédiaires, c'est-à-dire les valeurs du tableau après chaque passe. Préciser pour chaque variable utilisée, son type et sa signification.

Exercice 10 : tri par extraction simple

Réaliser un programme de tri par valeurs décroissantes d'un tableau d'entiers, en utilisant l'algorithme dit « par extraction simple » qui se définit de la manière suivante :

- On recherche le plus grand élément des n éléments du tableau ;
- On échange cet élément avec le premier du tableau ;
- Le plus grand élément du tableau se trouve alors en première position. On peut alors appliquer les deux opérations précédentes aux n-1 éléments restants, puis aux n-2, … et ceci jusqu'à ce qu'il ne reste plus qu'un seul élément (le dernier, lequel est alors le plus petit).

Exercice 11 : Cristal magique : 1re version

Un cristal d'un ordre donné n est une succession de '+' et de '–' empilé en triangle définie selon l'algorithme suivant : la première ligne est donnée et est constituée de n caractères '+' ou '-'. Pour une ligne l donnée, si les éléments [l-1,j] et [l-1,j+1] sont les mêmes caractères, alors [l,j] recoit le caractère '+'. Sinon [l,j] recoit '-'. Le cristal est dit magique s'il y a autant de caractère '+' que de caractère '-'.
Ecrire un algorithme qui demande à l'utilisateur de lire les n éléments de la première ligne et qui génère le reste du cristal selon la description ci-dessus. L'algorithme dira à la fin si le cristal est magique ou non.

Exercice 12 : Cristal magique : 2e version

En vous inspirant du précédant cas, ecrire un algorithme qui demande l'ordre du cristal, propose de façon aléatoire la première ligne et génère le reste du cristal. L'algorithme dira à la fin si le cristal est magique ou non.

Exercice 13 : Cristal magique : 3e version

Ecrire un algorithme qui recoit l'ordre du cristal et renvoie un cristal magique. L'algorithme notifiera au cas où l'obtention d'un cristal magique de cet ordre est impossible.

Exercice 14 : Manipulation de caractères et de fonctions

1. Écrivez une procédure *affiche_ligne(n : entier, car : car)* qui affiche n fois le caractère car suivi du caractère espace ' ', et finit par un retour à la ligne.
   exemples : L'appel *affiche_ligne(3,*)* produira l'affichage suivant :
   ```
   * * *
   ```
   L'appel *affiche_ligne(5,+)* produira l'affichage suivant :
   ```
   + + + + +
   ```

2. En utilisant la procédure *affiche_ligne*, écrivez une procédure *affiche_carre(cote :entier, car : car)* qui affiche un carré plein à l'aide du caractère car, comme dans l'exemple ci-dessous.
   exemple : l'appel *affiche_carre(5,*)* provoque l'affichage suivant :
   ```
   *  *  *  *  *
   *  *  *  *  *
   *  *  *  *  *
   *  *  *  *  *
   *  *  *  *  *
   ```

3. En utilisant la procédure *affiche_ligne*, écrivez une procédure *affiche_triangle(cote : entier, car : car)* qui affiche un triangle plein à l'aide du caractère car comme dans l'exemple ci-dessous.
   exemple : l'appel *affiche_triangle(5,*)* provoque l'affichage suivant :
   ```
   *
   * *
   * * *
   * * * *
   * * * * *
   ```

4. On voudrait maintenant afficher un triangle, mais en contrôlant son orientation. Écrivez la procédure *affiche_triangle2*, qui a le même en-tête que a*ffiche_triangle*, avec en plus un nouveau paramètre direction. Si ce paramètre vaut 0, l'hypoténuse est tournée vers le haut, comme dans l'exemple précédent. Si le paramètre vaut 1, elle est tournée vers le bas, comme dans l'exemple ci-dessous. Vous devez réutiliser la procédure *affiche_triangle*.

exemple : l'appel *affiche_triangle2(6,*,1)* provoque l'affichage suivant :

```
******
*****
****
***
**
*
```

5. En utilisant uniquement la procédure *affiche_triangle2*, écrivez la procédure *affiche_grand_triangle(cote : entier, car : car)*, qui affiche un triangle du même type que celui donné en exemple ci-dessous.
   exemple : l'appel *affiche_grand_triangle(5,*)* provoque l'affichage suivant :

```
*
**
***
****
*****
****
***
**
*
```

6. En utilisant la procédure *affiche_ligne*, écrivez une procédure *affiche_ligne2(n1 : entier, n2 : entier, car : car)* qui affiche n1 fois le caractères espace, puis n2 fois le caractère car suivi du caractère espace ' '.
   exemples : (les caractères espace sont représentés sous forme d'underscores ('_') pour que les exemples soient bien compréhensibles)
   L'appel *affiche_ligne(3,4,*)* produira l'affichage suivant :

```
___****
```

   L'appel *affiche_ligne(8,2,+)* produira l'affichage suivant :

```
________++
```

   L'appel *affiche_ligne(0,3,:)* produira l'affichage suivant :

```
:::
```

7. En utilisant la procédure *affiche_ligne2*, écrivez une procédure *affiche_croix(cote : entier, car : car)* qui dessine une croix comme dans l'exemple ci-dessous. Le paramètre *cote* contrôle la longueur des branches de la croix.
   exemple : l'appel *affiche_croix(5,*)* provoque l'affichage suivant :

```
          *
          *
          *
          *
*  *  *  *  *  *  *  *  *
          *
          *
          *
          *
```

8. Écrivez une procédure *affiche_croix2(n : entier, largeur : entier, car : car)* qui prend un paramètre supplémentaire par rapport à la procédure précédente *affiche_croix* : le paramètre *largeur*, qui contrôle l'épaisseur des branches de la croix.
   exemple : l'appel *affiche_croix2(6,3,*)* provoque l'affichage suivant :

```
            * * *
            * * *
            * * *
            * * *
* * * * * * * * * * * *
* * * * * * * * * * * *
* * * * * * * * * * * *
            * * *
            * * *
            * * *
            * * *
```

Remarque : on suppose que le paramètre largeur prend une valeur impaire.

Rappel de Syntaxe de définition de fonction et procédure su python :

```
def nom_function (parametres_eventuels)
      sequence
return resultat
```

Exemple :

```
def square(number):
      return number**2
```

Le mot clé return termine l'exécution d'une fonction.
Une fonction peut ne pas avoir de return : c'est le cas des **procédures**. Dans ce cas, l'indentation permet d'identifier la fin de la fonction.

Exemple : Suite de Syracuse

```
## CC ISE1 ENEAM Exercice 1  : suite de Syracuse
#définition de la procédure
def cc_eneam_exo1_suite_Syracuse (terme_initial):
    k=terme_initial
    u=[0]*100 #Déclaration et initialisation de tableau u
    u[0]=k
    b=u[0]!=1
    i=0
    while b:
        i=i+1
        if u[i-1]%2==0:
            u[i]=u[i-1]/2
        else :
            u[i]=3*u[i-1]+1
        b=u[i]!=1
    for i in range(0,i+1):
        print(u[i],end=" ") #Affichage des éléments de u
    print("Le temps d'arret est ",i)

#appel de la procédure
cc_eneam_exo1_suite_Syracuse(5)
cc_eneam_exo1_suite_Syracuse(37)
```