Exercice 1 (E2S1)

On considère un tableau de nombres de n lignes et p colonnes.

Les lignes sont numérotées de 0 à $n-1$ et les colonnes sont numérotées de 0 à $p-1$. La case en haut à gauche est repérée par (0,0) et la case en bas à droite par ($n-1$,$p-1$).

On appelle *chemin* une succession de cases allant de la case (0,0) à la case ($n-1$,$p-1$), en n'autorisant que des déplacements case par case : soit vers la droite, soit vers le bas.

On appelle *somme* d'un chemin la somme des entiers situés sur ce chemin.

Par exemple, pour le tableau T suivant :

4	1	1	3
2	0	2	1
3	1	5	1

— Un chemin est (0,0), (0,1), (0,2), (1,2), (2,2), (2,3) (en gras sur le tableau);

— La somme du chemin précédent est 14.

— (0,0), (0,2), (2,2), (2,3) n'est pas un chemin.

L'objectif de cet exercice est de déterminer la somme maximale pour tous les chemins possibles allant de la case (0, 0) à la case ($n-1$, $p-1$).

Question 1 On considère tous les chemins allant de la case (0, 0) à la case (2, 3) du tableau T donné en exemple.

1. Un tel chemin comprend nécessairement 3 déplacements vers la droite. Combien de déplacements vers le bas comprend-il?
2. La longueur d'un chemin est égal au nombre de cases de ce chemin. Justifier que tous les chemins allant de (0,0) à (2,3) ont une longueur égale à 6.

Question 2 En listant tous les chemins possibles allant de (0, 0) à (2, 3) du tableau T, déterminer un chemin qui permet d'obtenir la somme maximale et la valeur de cette somme.

Question 3 On veut créer le tableau T' où chaque élément T'[i][j] est la somme maximale pour tous les chemins possibles allant de (0,0) à (*i,j*).

1. Compléter et recopier sur votre copie le tableau T' donné ci-dessous associé au tableau

T =

4	1	1	3
2	0	2	1
3	1	5	1

T' =

4	5	6	?
6	?	8	10
9	10	?	16

2. Justifier que si j est différent de 0, alors : T' [0] [j] = T [0] [j] + T' [0] [j-1]

Question 4 Justifier que si i et j sont différents de 0, alors : T' [i] [j] = T [i] [j] + max(T' [i-1] [j], T' [i] [j-1]).

Question 5 On veut créer la fonction récursive somme_max ayant pour paramètres un tableau T, un entier i et un entier j. Cette fonction renvoie la somme maximale pour tous les chemins possibles allant de la case (0, 0) à la case (i, j).

1. Quel est le cas de base, à savoir le cas qui est traité directement sans faire appel à la fonction somme_max? Que renvoie-t-on dans ce cas?
2. À l'aide de la question précédente, écrire en Python la fonction récursive somme_max .
3. Quel appel de fonction doit-on faire pour résoudre le problème initial?

Exercice 2 (E2S2PC)

Dans cette partie, pour une meilleure lisibilité, des espaces sont placées dans les écritures binaires des nombres. Il ne faut pas les prendre en compte dans les calculs.

Pour chiffrer un message, une méthode, dite du masque jetable, consiste à le combiner avec une chaîne de caractères de longueur comparable.
Une implémentation possible utilise l'opérateur XOR (ou exclusif) dont voici la table de vérité :

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

Dans la suite, les nombres écrits en binaire seront précédés du préfixe 0b.

1. Pour chiffrer un message, on convertit chacun de ses caractères en binaire (à l'aide du format Unicode), et on réalise l'opération XOR bit à bit avec la clé.

Après conversion en binaire, et avant que l'opération XOR bit à bit avec la clé n'ait été effectuée, Alice obtient le message suivant :

m = 0b 0110 0011 0100 0110

a. Le message m correspond à deux caractères codés chacun sur 8 bits : déterminer quels sont ces caractères. On fournit pour cela la table ci-dessous qui associe à l'écriture hexadécimale d'un octet le caractère correspondant (figure 2). Exemple de lecture : le caractère correspondant à l'octet codé 4A en hexadécimal est la lettre **J**.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	space	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Figure 2

b. Pour chiffrer le message d'Alice, on réalise l'opération XOR bit à bit avec la clé suivante :

k = 0b 1110 1110 1111 0000

Donner l'écriture binaire du message obtenu.

2.

a. Dresser la table de vérité de l'expression booléenne suivante :

(a XOR b) XOR b

b. Bob connaît la chaîne de caractères utilisée par Alice pour chiffrer le message. Quelle opération doit-il réaliser pour déchiffrer son message ?

Exercice 3 (E3S4)

Une agence immobilière développe un programme pour gérer les biens immobiliers qu'elle propose à la vente.

Dans ce programme, pour modéliser les données de biens immobiliers, on définit une classe Bim avec les attributs suivants :

- nt de type str représente la nature du bien (appartement, maison, bureau, commerces, …) ;
- sf de type float est la surface du bien ;
- pm de type float est le prix moyen par m² du bien qui dépend de son emplacement.

La classe Bim possède une méthode estim_prix qui renvoie une estimation du prix du bien. Le code (incomplet) de la classe Bim est donné ci-dessous :

```
class Bim:
    def __init__(self, nature, surface, prix_moy):
        …
    def estim_prix(self):
        return self.sf * self.pm
```

1. Recopier et compléter le code du constructeur de la classe Bim.

2. On exécute l'instruction suivante :

```
b1 = Bim('maison', 70.0, 2000.0)
```

Que renvoie l'instruction b1.estim_prix() ? Préciser le type de la valeur renvoyée.

3. On souhaite affiner l'estimation du prix d'un bien en prenant en compte sa nature :

- pour un bien dont l'attribut nt est 'maison' la nouvelle estimation du prix est le produit de sa surface par le prix moyen par m² multiplié par 1,1 ;
- pour un bien dont l'attribut nt est 'bureau' la nouvelle estimation du prix est le produit de sa surface par le prix moyen par m² multiplié par 0,8 ;
- pour les biens d'autres natures, l'estimation du prix ne change pas.

Modifier le code de la méthode estim_prix afin de prendre en compte ce changement de calcul.

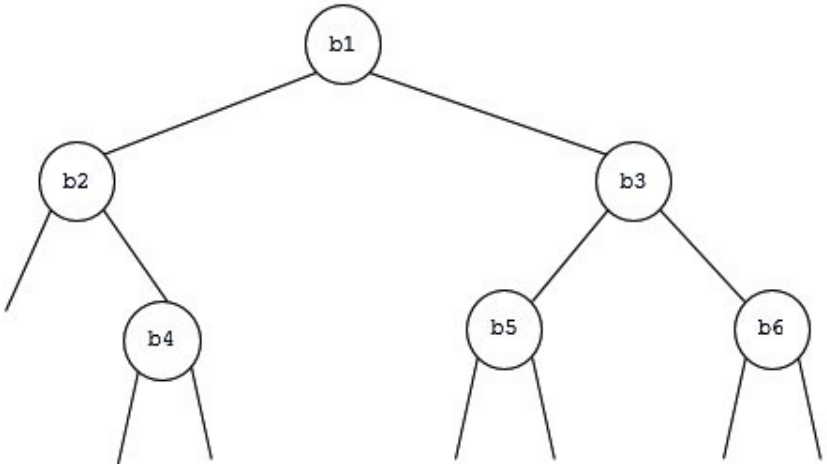
4. Écrire le code Python d'une fonction nb_maison(lst) qui prend en argument une liste Python de biens immobiliers de type Bim et qui renvoie le nombre d'objets de nature 'maison' contenus dans la liste lst.

5. Pour une recherche efficace des biens immobiliers selon le critère de leur surface, on stocke les objets de type Bim dans un arbre binaire de recherche, nommé abr. Pour tout nœud de cet arbre :

- tous les objets de son sous-arbre gauche ont une surface inférieure ou égale à la surface de l'objet contenue dans ce nœud ;
- tous les objets de son sous-arbre droit ont une surface strictement supérieure à la surface de l'objet contenue dans ce nœud.

L'objet abr dispose des méthodes suivantes :
abr.est_vide() : renvoie True si abr est vide et False sinon.
abr.get_v() : renvoie l'élément (de type Bim) situé à la racine de abr si abr n'est pas vide et None sinon.
abr.get_g() : renvoie le sous-arbre gauche de abr si abr n'est pas vide et None sinon. abr.get_d() : renvoie le sous-arbre droit de abr si abr n'est pas vide et None sinon.

3. Dans cette question, on suppose que l'arbre binaire abr a la forme ci-dessous :



Donner la liste les biens b1, b2, b3, b4, b5, b6 triée dans l'ordre croissant de leur surface.

4. Recopier et compléter le code de la fonction récursive contient donnée ci-dessous, qui prend en arguments un nombre surface de type float et un arbre binaire de recherche abr contenant des éléments de type Bim ordonnés selon leur attribut de surface sf. La fonction contient(surface, abr) renvoie True s'il existe un bien dans abr d'une surface supérieure ou égale à surface et False sinon.

```
def contient(surface, abr):
    if abr.est_vide():
        return False
    elif abr.get_v().sf >= ……. :
        return True
    else:
        return contient( surface , ……. )
```

Exercice 4 (E4S4)

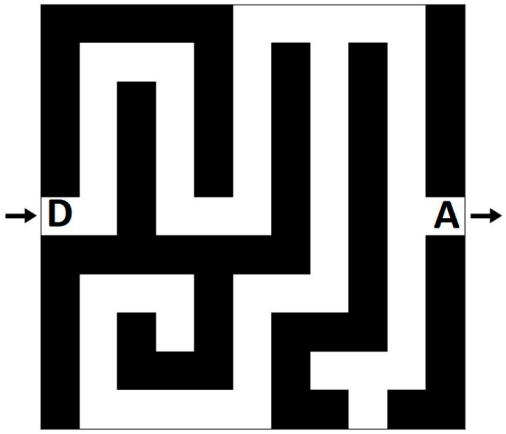
Partie A : Représentation d'un labyrinthe

On modélise un labyrinthe par un tableau à deux dimensions à lignes et colonnes avec et des entiers strictement positifs.
Les lignes sont numérotées de 0 à 1 et les colonnes de 0 à 1.
La case en haut à gauche est repérée par 0,0 et la case en bas à droite par 1; 1 .

Dans ce tableau :

- 0 représente une case vide, hors case de départ et arrivée,
- 1 représente un mur,
- 2 représente le départ du labyrinthe,
- 3 représente l'arrivée du labyrinthe.

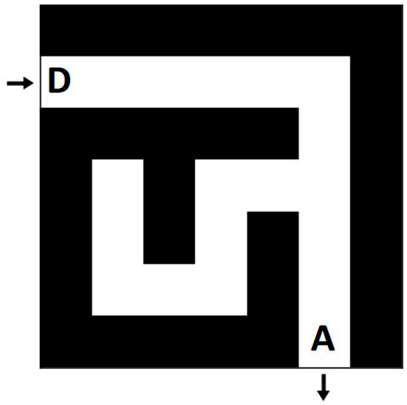
Ainsi, en Python, le labyrinthe ci-dessous est représentée par le tableau de tableaux lab1.




```
lab1 = [[1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1],
        [1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        [2, 0, 1, 0, 0, 0, 1, 0, 1, 0, 3],
        [1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1],
        [1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1],
        [1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1],
        [1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1]]
```

1. Le labyrinthe ci-dessous est censé être représenté par le tableau de tableaux lab2. Cependant, dans ce tableau, un mur se trouve à la place du départ du labyrinthe. Donner une instruction permettant de placer le départ au bon endroit dans lab2.

2.




```
lab2 = [[1, 1, 1, 1, 1, 1, 1],
        [1, 0, 0, 0, 0, 0, 1],
        [1, 1, 1, 1, 1, 0, 1],
        [1, 0, 1, 0, 0, 0, 1],
        [1, 0, 1, 0, 1, 0, 1],
        [1, 0, 0, 0, 1, 0, 1],
        [1, 1, 1, 1, 1, 3, 1]]
```

3. Écrire une fonction est_valide(i, j, n, m) qui renvoie True si le couple (;) correspond à des coordonnées valides pour un labyrinthe de taille (;), et False sinon. On donne ci-dessous des exemples d'appels.

```
>>> est_valide(5, 2, 10, 10)
True
>>> est_valide(-3, 4, 10, 10)
False
```

4. On suppose que le départ d'un labyrinthe est toujours indiqué, mais on ne fait aucune supposition sur son emplacement. Compléter la fonction depart(lab)ci-dessous de sorte qu'elle renvoie, sous la forme d'un tuple, les coordonnées du départ d'un labyrinthe (représenté par le paramètre lab). Par exemple, l'appel depart(lab1) doit renvoyer le tuple (5, 0).

```
def depart(lab) :     n = len(lab)
m = len(lab[0])     …
```

5. Écrire une fonction nb_cases_vides(lab) qui renvoie le nombre de cases vides d'un labyrinthe (comprenant donc l'arrivée et le départ).
Par exemple, l'appel nb_cases_vides(lab2) doit renvoyer la valeur 19.

Partie B : Recherche d'une solution dans un labyrinthe

On suppose dans cette partie que les labyrinthes possèdent un unique chemin allant du départ à l'arrivée sans repasser par la même case. Dans la suite, c'est ce chemin que l'on appellera solution du labyrinthe.

Pour déterminer la solution d'un labyrinthe, on parcourt les cases vides de proche en proche.
Lors d'un tel parcours, afin d'éviter de tourner en rond, on choisit de marquer les cases visitées. Pour cela, on remplace la valeur d'une case visitée dans le tableau représentant le labyrinthe par la valeur 4.

1. On dit que deux cases d'un labyrinthe sont voisines si elles ont un côté commun.
On considère une fonction voisines(i, j, lab) qui prend en arguments deux entiers i et j représentant les coordonnées d'une case et un tableau lab qui représente un labyrinthe. Cette fonction renvoie la liste des coordonnées des cases voisines de la case de coordonnées (i, j) qui sont valides, non visitées et qui ne sont pas des murs. L'ordre des éléments de cette liste n'importe pas.

Ainsi, l'appel voisines(1, 1, [[1, 1, 1], [4, 0, 0], [1, 0, 1]]) renvoie la liste [(2, 1), (1, 2)].

Que renvoie l'appel voisines(1, 2, [[1, 1, 4], [0, 0, 0], [1, 1, 0]]) ?

2. On souhaite stocker la solution dans une liste chemin. Cette liste contiendra les coordonnées des cases de la solution, dans l'ordre. Pour cela, on procède de la façon suivante.

- Initialement :
  - déterminer les coordonnées du départ : c'est la première case à visiter ; - ajouter les coordonnées de la case départ à la liste chemin.
- Tant que l'arrivée n'a pas été atteinte :
  - on marque la case visitée avec la valeur 4 ;
  - si la case visitée possède une case voisine libre, la première case de la liste renvoyée par la fonction voisines devient la prochaine case à visiter et on ajoute à la liste chemin ;
  - sinon, il s'agit d'une impasse. On supprime alors la dernière case dans la liste chemin. La prochaine case à visiter est celle qui est désormais en dernière position de la liste chemin.

a. Le tableau de tableaux lab3 ci-dessous représente un labyrinthe.

```
Lab3 = [[1, 1, 1, 1, 1, 1],
        [2, 0, 0, 0, 0, 3],
        [1,0, 1, 0, 1, 1],
        [1, 1, 1, 0, 0, 1]]
```

La suite d'instructions ci-dessous simule le début des modifications subies par la liste chemin lorsque l'on applique la méthode présentée.

```
# entrée: (1, 0), sortie (1, 5)
chemin = [(1, 0)]
chemin.append((1, 1))
chemin.append((2, 1))
chemin.pop()
chemin.append((1, 2))
chemin.append((1, 3))
chemin.append((2, 3))
```

Compléter cette suite d'instructions jusqu'à ce que la liste chemin représente la solution. *Rappel : la méthode* pop *supprime le dernier élément d'une liste et renvoie cet élément.*

b. Recopier et compléter la fonction solution(lab) donnée ci-dessous de sorte qu'elle renvoie le chemin solution du labyrinthe représenté par le paramètre lab. On pourra pour cela utiliser la fonction voisines.

```
def solution(lab):
    chemin = [depart(lab)]
    case = chemin[0]
    i = case[0]
    j = case[1]
        …
```

Par exemple, l 'appel solution(lab2) doit renvoyer [(1, 0), (1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (2, 5), (3, 5), (4, 5), (5, 5), (6, 5)].

Exercice 5 (E5S4)

Dans un tableau Python d'entiers tab, on dit que le couple d'indices (i, j) forme une inversion lorsque i<j et tab[i] > tab[j]. On donne ci-dessous quelques exemples.

- Dans le tableau [1, 5, 3, 7], le couple d'indices (1,2) forme une inversion car 5 > 3.
  Par contre, le couple (1,3) ne forme pas d'inversion car 5 < 7. Il n'y a
  qu'une inversion dans ce tableau.

- Il y a trois inversions dans le tableau [1, 6, 2, 7, 3], à savoir les couples d'indices (1, 2), (1, 4) et (3, 4).

- On peut compter six inversions dans le tableau [7, 6, 5, 3] : les couples d'indices (0, 1), (0, 2), (0, 3), (1, 2), (1, 3) et (2, 3).

On se propose dans cet exercice de déterminer le nombre d'inversions dans un tableau quelconque.

Questions préliminaires

1. Expliquer pourquoi le couple (1, 3) est une inversion dans le tableau [4, 8, 3, 7].
2. Justifier que le couple (2, 3) n'en est pas une.

Partie A : Méthode itérative

Le but de cette partie est d'écrire une fonction itérative nombre_inversion qui renvoie le nombre d'inversions dans un tableau. Pour cela, on commence par écrire une fonction fonction1 qui sera ensuite utilisée pour écrire la fonction nombre_inversion.

1. On donne la fonction suivante.

```
def fonction1(tab, i):
    nb_elem = len(tab)
    cpt = 0
    for j in range(i+1, nb_elem):
        if tab[j] < tab[i]:
            cpt += 1

    return cpt
```

   a. Indiquer ce que renvoie la fonction1(tab, i) dans les cas suivants.

   - Cas n°1 : tab = [1, 5, 3, 7] et i = 0.
   - Cas n°2 : tab = [1, 5, 3, 7] et i = 1.
   - Cas n°3 : tab = [1, 5, 2, 6, 4] et i = 1.

   b. Expliquer ce que permet de déterminer cette fonction.

2. En utilisant la fonction précédente, écrire une fonction nombre_inversion(tab) qui prend en argument un tableau et renvoie le nombre d'inversions dans ce tableau. On donne ci-dessous les résultats attendus pour certains appels.

```
>>> nombre_inversions([1, 5, 7])
0
>>> nombre_inversions([1, 6, 2, 7, 3])
3
```

```
>>> nombre_inversions([7, 6, 5, 3])
6
```

3. Quelle est l'ordre de grandeur de la complexité en temps de l'algorithme obtenu ? Aucune justification n'est attendue.

Partie B : Méthode récursive

Le but de cette partie est de concevoir une version récursive de la fonction nombre_inversion.
On définit pour cela des fonctions auxiliaires.

1. Donner le nom d'un algorithme de tri ayant une complexité meilleure que quadratique.

Dans la suite de cet exercice, on suppose qu'on dispose d'une fonction tri(tab) qui prend en argument un tableau et renvoie un tableau contenant les mêmes éléments rangés dans l'ordre croissant.

2. Écrire une fonction moitie_gauche(tab) qui prend en argument un tableau tab et renvoie un nouveau tableau contenant la moitié gauche de tab. Si le nombre d'éléments de tab est impair, l'élément du centre se trouve dans cette partie gauche. On donne ci-dessous les résultats attendus pour certains appels.

```
>>> moitie_gauche([])
[]
>>> moitie_gauche([4, 8, 3])
[4, 8]
>>> moitie_gauche ([4, 8, 3, 7])
[4, 8]
```

Dans la suite, on suppose qu'on dispose de la fonction moitie_droite(tab) qui renvoie la moitié droite sans l'élément du milieu.

3. On suppose qu'une fonction nb_inv_tab(tab1, tab2)a été écrite. Cette fonction renvoie le nombre d'inversions du tableau obtenu en mettant bout à bout les tableaux tab1 et tab2, à condition que tab1 et tab2 soient triés dans l'ordre croissant. On donne ci-dessous deux exemples d'appel de cette fonction :

```
>>> nb_inv_tab([3, 7, 9], [2, 10])
3
>>> nb_inv_tab([7, 9, 13], [7, 10, 14])
3
```

En utilisant la fonction nb_inv_tab et les questions précédentes, écrire une fonction récursive nb_inversions_rec(tab) qui permet de calculer le nombre d'inversions dans un tableau. Cette fonction renverra le même nombre que nombre_inversions(tab) de la partie A. On procédera de la façon suivante :

- Séparer le tableau en deux tableaux de tailles égales (à une unité près).
- Appeler récursivement la fonction nb_inversions_rec pour compter le nombre d'inversions dans chacun des deux tableaux.
- Trier les deux tableaux (on rappelle qu'une fonction de tri est déjà définie).
- Ajouter au nombre d'inversions précédemment comptées le nombre renvoyé par la fonction nb_inv_tab avec pour arguments les deux tableaux triés.

Exercice 6 (E4S5)

On s'intéresse dans cet exercice à un algorithme de mélange des éléments d'une liste.

1, Pour la suite, il sera utile de disposer d'une fonction echange qui permet d'échanger dans une liste lst les éléments d'indice i1 et i2.
Expliquer pourquoi le code Python ci-dessous ne réalise pas cet échange et en proposer une modification.

```
def echange(lst, i1, i2):
    lst[i2] = lst[i1]
    lst[i1] = lst[i2]
```

•

2, La documentation du module random de Python fournit les informations cidessous concernant la fonction randint(a,b) :

Renvoie un entier aléatoire N tel que a <= N <= b. Alias pour randrange(a,b+1).

Parmi les valeurs ci-dessous, quelles sont celles qui peuvent être renvoyées par l'appel randint(0, 10) ?

0 1 3.5 9 10 11

3, Le mélange de Fischer Yates est un algorithme permettant de permuter aléatoirement les éléments d'une liste. On donne ci-dessous une mise en œuvre récursive de cet algorithme en Python.

```
from random import randint

def melange(lst, ind):
    print(lst)
    if ind > 0:
        j = randint(0, ind)
        echange(lst, ind, j)
        melange(lst, ind-1)
```

- Expliquer pourquoi la fonction melange se termine toujours.

Lors de l'appel de la fonction melange, la valeur du paramètre ind doit être égal au plus grand indice possible de la liste lst.

- Pour une liste de longueur , quel est le nombre d'appels récursifs de la fonction melange effectués, sans compter l'appel initial ?

On considère le script ci-dessous :

```
lst = [v for v in range(5)]
melange(lst, 4)
```

On suppose que les valeurs successivement renvoyées par la fonction randint sont 2, 1, 2 et 0.

Les deux premiers affichages produits par l'instruction print(lst) de la fonction melange sont :

[0, 1, 2, 3, 4]

[0, 1, 4, 3, 2]

- Donner les affichages suivants produits par la fonction melange.
- Proposer une version itérative du mélange de Fischer Yates.

Exercice 7 (E5S5)

Étant donné un tableau non vide de nombres entiers relatifs, on appelle sous-séquence une suite non vide d'éléments voisins de ce tableau. On cherche dans cet exercice à déterminer la plus grande somme possible obtenue en additionnant les éléments d'une sous-séquence.

Par exemple, pour le tableau ci-dessous, la somme maximale vaut 18.
Elle est obtenue en additionnant les éléments de la sous-séquence encadrée en gras ci-dessous (6 ; 8 ; -6 ; 10).

-8	-4	**6**	**8**	**-6**	**10**	-4	-4

a. Quelle est la solution du problème si les éléments du tableau sont tous positifs ?

b. Quelle est la solution du problème si tous les éléments sont négatifs ?

- Dans cette question, on examine toutes les sous-séquences possibles.

  a. Écrire le code Python d'une fonction somme_sous_sequence(lst, i, j) qui prend en argument une liste et deux entiers i, j et renvoie la somme de la sous-séquence délimitée par les indices i et j (inclus).

  b. La fonction pgsp ci-dessous permet de déterminer la plus grande des sommes obtenues en additionnant les éléments de toutes les sous-séquences possibles du tableau lst.

```
def pgsp(lst):
    n = len(lst)
    somme_max = lst[0]
    for i in range(n):
        for j in range(i, n):
            s = somme_sous_sequence(lst, i, j)
            if s > somme_max :
                somme_max = s
    return somme_max
```

  Parmi les quatre choix suivants, quel est le nombre de comparaisons effectuées par cette fonction si le tableau lst passé en paramètre contient 10 éléments ?

  10 55 100 1055

  c. Recopier et modifier la fonction pgsp pour qu'elle renvoie un tuple contenant la somme maximale et les indices qui délimitent la sous-séquence correspondant à cette somme maximale.

- On considère dans cette question une approche plus élaborée. Son principe consiste, pour toutes les valeurs possibles de l'indice , à déterminer la somme maximale ( ) des sous-séquences qui se terminent à l'indice .
  En désignant par lst[ ] l'élément de lst d'indice , on peut vérifier que

  S(0) = lst[0]

  et pour i ≥ 1 :

  S(i) = lst[i] si S(i−1) ≤ 0 ;

  S(i) = lst[i] +S(i−1) si S(i−1) > 0.

  a. Recopier et compléter le tableau ci-dessous avec les valeurs de ( ) pour la liste considérée en exemple.

i	0	1	2	3	4	5	6	7
lst[i]	-8	-4	6	8	-6	10	-4	-4
S(i)								

b. La solution au problème étant la plus grande valeur des S(i), on demande de compléter la fonction pgsp2 ci-dessous, de sorte que la variable sommes_max contienne la liste des valeurs S(i) .

```
def pgsp2(lst):
    sommes_max = [lst[0]]
    for i in range(1, len(lst)):

        # à compléter
    return max(sommes_max)
```

c. En quoi la solution obtenue par cette approche est-elle plus avantageuse que celle de la question **2.b.** ?

Exercice 8 (E3S7)

L'objectif de l'exercice est d'étudier une méthode de cryptage d'une chaîne de caractères à l'aide du codage ASCII et de la fonction logique XOR.

Le nombre 65, donné ici en écriture décimale, s'écrit 01000001 en notation binaire. En détaillant la méthode utilisée, **donner** l'écriture binaire du nombre 89.

La fonction logique OU EXCLUSIF, appelée XOR et représentée par le symbole ⊕, fournit une sortie égale à 1 si l'une ou l'autre des deux entrées vaut 1 mais pas les deux.

On donne ci-contre la table de vérité de la fonction XOR.

E_1	E_2	$E_1 \oplus E_2$
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	**1**	**0**

Si on applique cette fonction à un nombre codé en binaire, elle opère bit à bit.

$$\begin{array}{rr} & 1100 \\ \oplus & 1010 \\ \hline = & 0110 \end{array}$$

Poser et calculer l'opération : 11001110 ⊕ 01101011

3. On donne, ci-dessous, un extrait de la table ASCII qui permet d'encoder les caractères de A à Z.
On peut alors considérer l'opération XOR entre deux caractères en effectuant le XOR entre les codes ASCII des deux caractères. Par exemple : 'F' XOR 'S' sera le résultat de 01000110 ⊕ 01010011.

Code ASCII Décimal	**Code ASCII Binaire**	**Caractère**
65	01000001	A
66	01000010	B
67	01000011	C
68	01000100	D
69	01000101	E
70	01000110	F
71	01000111	G
72	01001000	H
73	01001001	I
74	01001010	J
75	01001011	K
76	01001100	L
77	01001101	M

Code ASCII Décimal	**Code ASCII Binaire**	**Caractère**
78	01001110	N
79	01001111	O
80	01010000	P
81	01010001	Q
82	01010010	R
83	01010011	S
84	01010100	T
85	01010101	U
86	01010110	V
87	01010111	W
88	01011000	X
89	01011001	Y
90	01011010	Z

On souhaite mettre au point une méthode de cryptage à l'aide de la fonction XOR.

Pour cela, on dispose d'un message à crypter et d'une clé de cryptage de même longueur que ce message. Le message et la clé sont composés uniquement des caractères du tableau ci-dessus et on applique la fonction XOR caractère par caractère entre les lettres du message à crypter et les lettres de la clé de cryptage.

Par exemple, voici le cryptage du mot ALPHA à l'aide de la clé YAKYA :

Message à crypter	A	L	P	H	A
Clé de cryptage	Y	A	K	Y	A
Message crypté	□	□	□	□	□
	'A' XOR 'Y'	'L' XOR 'A'	'P' XOR 'K'	…	…

Ecrire une fonction **xor_crypt(message, cle)** qui prend en paramètres deux chaînes de caractères et qui renvoie la liste des entiers correspondant au message crypté.

Aide :

- On pourra utiliser la fonction native du langage Python **ord(c)** qui prend en paramètre un caractère **c** et qui renvoie un nombre représentant le code ASCII du caractère **c**.
- On considère également que l'on dispose d'une fonction écrite **xor(n1,n2)** qui prend en paramètre deux nombres **n1** et **n2** et qui renvoie le résultat de **n1** ⊕ **n2**.

c. On souhaite maintenant générer une clé de la taille du message à partir d'un mot quelconque. On considère que le mot choisi est plus court que le message, il faut donc le reproduire un certain nombre de fois pour créer une clé de la même longueur que le message.

Par exemple, si le mot choisi est YAK pour crypter le message ALPHABET, la clé sera YAKYAKYA.

Créer une fonction **generer_cle(mot,n)** qui renvoie la clé de longueur **n** à partir de la chaîne de caractères **mot**.

d. Recopier et **compléter** la table de vérité de $(E_1 \oplus E_2) \oplus E_2$.

E_1	E_2	$E_1 \oplus E_2$	$(E_1 \oplus E_2) \oplus E_2$
0	**0**	**0**	
0	**1**	**1**	
1	**0**	**1**	
1	**1**	**0**	

A l'aide de ce résultat, **proposer** une démarche pour décrypter un message crypté.

Exercice 9 (E1S9)

Partie A : Manipulation d'une liste en Python

- Donner les affichages obtenus après l'exécution du code Python suivant.

```
notes = [8, 7, 18, 14, 12, 9, 17, 3]
notes[3] = 16
print(len(notes))
print(notes)
```

- Écrire un code Python permettant d'afficher les éléments d'indice 2 à 4 de la liste notes.

Partie B : Tri par insertion

Le tri par insertion est un algorithme efficace qui s'inspire de la façon dont on peut trier une poignée de cartes. On commence avec une seule carte dans la main gauche (les autres cartes sont en tas sur la table) puis on pioche la carte suivante et on l'insère au bon endroit dans la main gauche.

1. Voici une implémentation en Python de cet algorithme. Recopier et compléter les lignes 6 et 7 surlignées (uniquement celles-ci).

```
def tri_insertion(liste):
    """ trie par insertion la liste en paramètre """
    for indice_courant in range(1,len(liste)):
        element_a_inserer = liste[indice_courant]
        i = indice_courant – 1
        while i >= 0 and liste[i] > ..... :
            liste[.....] = liste[.....…]
            i = i – 1
        liste[i + 1] = element_a_inserer
```

On a écrit dans la console les instructions suivantes :

```
notes = [8, 7, 18, 14, 12, 9, 17, 3]
tri_insertion(notes)
print(notes)
```

On a obtenu l'affichage suivant : [3, 7, 8, 9, 12, 14, 17, 18]

On s'interroge sur ce qui s'est passé lors de l'exécution de tri_insertion(notes).

2. Donner le contenu de la liste notes après le premier passage dans la boucle for.
3. Donner le contenu de la liste notes après le troisième passage dans la boucle for.

Partie C : Tri fusion

L'algorithme de tri fusion suit le principe de « diviser pour régner ».

(1) Si le tableau à trier n'a qu'un élément, il est déjà trié.

(2) Sinon, séparer le tableau en deux parties à peu près égales.

(3) Trier les deux parties avec l'algorithme de tri fusion.

(4) Fusionner les deux tableaux triés en un seul tableau.

source : Wikipedia

- Cet algorithme est-il itératif ou récursif ? Justifier en une phrase.

- Expliquer en trois lignes comment faire pour rassembler dans une main deux tas déjà triés de cartes, la carte en haut d'un tas étant la plus petite de ce même tas ; la deuxième carte d'un tas n'étant visible qu'après avoir retiré la première carte de ce tas.
  À la fin du procédé, les cartes en main doivent être triées par ordre croissant.

Une fonction fusionner a été implémentée en Python en s'inspirant du procédé de la question précédente. Elle prend quatre arguments : la liste qui est en train d'être triée, l'indice où commence la sous-liste de gauche à fusionner, l'indice où termine cette sous-liste, et l'indice où se termine la sous-liste de droite.

- Voici une implémentation de l'algorithme de tri fusion. Recopier et compléter les lignes 8, 9 et 10 surlignées (uniquement celles-ci).

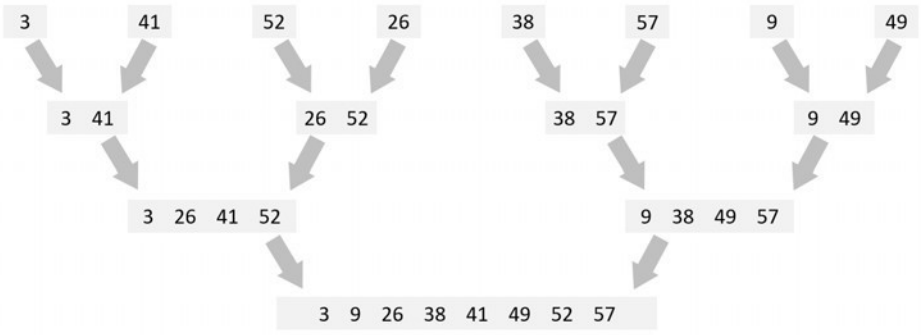
```
from math import floor
def tri_fusion (liste, i_debut, i_fin):
    """ trie par fusion la liste en paramètre depuis
    i_debut jusqu'à i_fin """
    if i_debut < i_fin:
        i_partage = floor((i_debut + i_fin) / 2)
        tri_fusion(liste, i_debut, ..............…)
        tri_fusion(liste, ....................., i_fin)
        fusionner(liste, ....................., ....................., .....................)
```

Remarque : la fonction floor renvoie la partie entière du nombre passé en paramètre.

- Expliquer le rôle de la première ligne du code de la question précédente.

Partie D : Comparaison du tri par insertion et du tri fusion

Voici une illustration des étapes d'un tri effectué sur la liste [3, 41, 52, 26, 38, 57, 9, 49].



- Quel algorithme a été utilisé : le tri par insertion ou le tri fusion ? Justifier.

- Identifier le tri qui a une complexité, dans le pire des cas, en $O(n^2)$ et identifier le tri qui a une complexité, dans le pire des cas, en $O(n \log_2 n)$.
  Remarque : n représente la longueur de la liste à trier.

- Justifier brièvement ces deux complexités.

Exercice 10 (E5S10)

Afin d'améliorer l'ergonomie d'un logiciel de traitement des inscriptions dans une université, un programmeur souhaite exploiter l'intelligence artificielle pour renseigner certains champs par auto-complétion. Il s'intéresse au descripteur « genre » (masculin, féminin ou indéterminé). Pour cela il propose d'exploiter les dernières lettres du prénom pour proposer automatiquement le genre.

Pour vérifier son hypothèse, il récupère un fichier CSV associant plus de 60 000 prénoms du monde entier au genre de la personne portant ce prénom. En utilisant seulement la dernière lettre, le taux de réussite de sa démarche est de 72,9% avec la fonction définie ci-dessous :

```
1  def genre(prenom):
2      liste_M = ['f', 'd', 'c', 'b', 'o', 'n', 'm', 'l', 'k',
3                 'j', 'é', 'h', 'w', 'v', 'u', 't', 's', 'r',
4                  'q', 'p', 'i', 'þ', 'z', 'x', 'ç', 'ö', 'ã',
5                  'â', 'ï', 'g']
6      liste_F = ['e', 'a', 'ä', 'ü', 'y', 'ë']
7 8
9 i    if prenom[len(prenom)-1].lower() in liste_M :
10 r       return "M"
11
12     elif prenom[len(prenom)-1].lower() in liste_F :
13         return "F"
       else :
           return "I"


  #  Pour rappel, C.lower() convertit le caractère C en minuscule.
```

1. Appropriation

- Expliquer ce qu'est un fichier CSV.
- Donner le type de l'argument **prenom** de la fonction **genre,** et le type de la réponse renvoyée.

2. Développement

Pour effectuer son étude sur les prénoms à partir du fichier CSV, le programmeur souhaite utiliser la bibliothèque csv.

a La bibliothèque csv est un module natif du moteur python.

Donner, dans ce cas, l'instruction d'importation.

b Au cours du développement de son projet, le programmeur souhaite insérer une assertion sur l'argument donné à la fonction.

Proposer une assertion sur le type de l'argument qui corrige une erreur lorsque le type ne correspond pas à celui attendu.

c Avant le déploiement de sa solution, le programmeur décide de rendre sa fonction plus robuste.

Pour cela il veut remplacer l'assertion proposée dans la question **2.b)** par une gestion de l'argument pour éviter toutes erreurs empêchant la poursuite du programme.

Proposer alors une ou plusieurs instructions en Python utilisant l'argument afin de s'assurer que la fonction se termine quel que soit le type de l'argument.

3. Améliorations

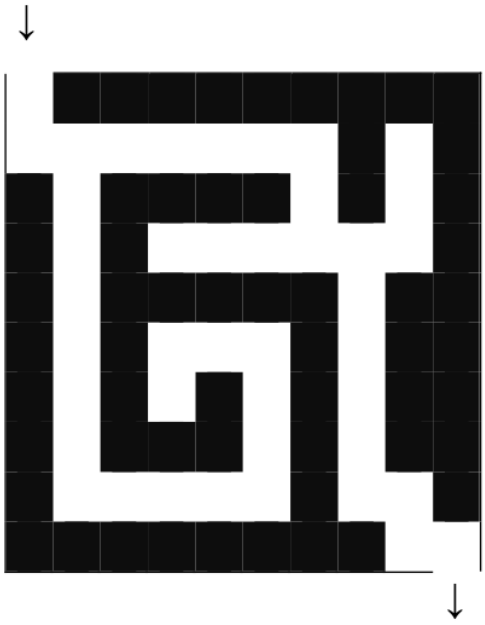
En prenant en compte les deux dernières lettres du prénom, il parvient à augmenter son taux de réussite à 74,4%. Pour cela, son étude du fichier CSV lui permet de créer deux listes : **liste_M2** pour les terminaisons de deux lettres associées aux prénoms masculins et **liste_F2** pour les prénoms féminins.

Sur votre copie, recopier et modifier la structure conditionnelle (lignes 8 à 13) de la fonction genre afin de prendre en compte les terminaisons de deux lettres des listes liste_M2 et liste_F2.

Exercice 11 (E2S7)

L'objectif de cet exercice est de mettre en place une modélisation d'un jeu de labyrinthe en langage Python.

On décide de représenter un labyrinthe par un tableau carré de taille n, dans lequel les cases seront des 0 si l'on peut s'y déplacer et des 1 s'il s'agit d'un mur. Voici un exemple de représentation d'un labyrinthe :



```
laby=[[0,1,1,1,1,1,1,1,1,1],
      [0,0,0,0,0,0,0,1,0,1],
      [1,0,1,1,1,1,0,1,0,1],
      [1,0,1,0,0,0,0,0,0,1],
      [1,0,1,1,1,1,1,0,1,1],
      [1,0,1,0,0,0,1,0,1,1],
      [1,0,1,0,1,0,1,0,1,1],
      [1,0,1,1,1,0,1,0,1,1],
      [1,0,0,0,0,0,1,0,0,1],
      [1,1,1,1,1,1,1,1,0,0]]
```

L'entrée du labyrinthe se situe à la première case du tableau (celle en haut à gauche) et la sortie du labyrinthe se trouve à la dernière case (celle en bas à droite).

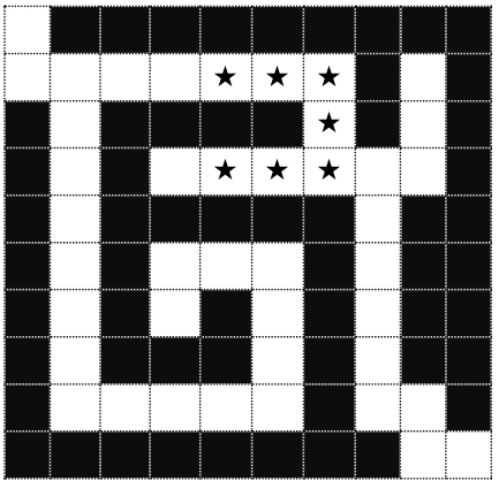
1. **Proposer**, en langage Python, une fonction mur, prenant en paramètre un tableau représentant un labyrinthe et deux entiers i et j compris entre 0 et n-1 et qui renvoie un booléen indiquant la présence ou non d'un mur. Par exemple :

```
>>mur(laby, 2, 3)
 True
>>mur(laby, 1, 8)
 False
```

Un parcours dans le labyrinthe va être représenté par une liste de **cases.** Il s'agit de couples (i,j) où i et j correspondent respectivement aux numéros de ligne et de colonne des cases successivement visitées au long du parcours. Ainsi, la liste suivante

[(1,4),(1,5),(1,6),(2,6),(3,6),(3,5),(3,4)]

correspond au parcours repéré par des étoiles ★ ci-contre :



La liste [(0,0),(1,0),(1,1),(5,1),(6,1)] ne peut correspondre au parcours d'un labyrinthe car toutes les cases parcourues successivement ne sont pas adjacentes.

2. On considère la fonction voisine ci-dessous, écrite en langage Python, qui prend en paramètres deux cases données sous forme de couple.

```
def voisine(case1, case2) :
    l1, c1 = case1
    l2, c2 = case2
    # on vous rappelle que **2 signifie puissance 2
    d = (l1-l2)**2 + (c1-c2)**2
    return (d == 1)
```

2.a Après avoir remarqué que les quantités **l1-l2** et **c1-c2** sont des entiers, **expliquer** pourquoi la fonction **voisine** indique si deux cases données sous forme de tuples **(l,c)** sont adjacentes.
2.b En déduire une fonction **adjacentes** qui reçoit une liste de cases et renvoie un booléen indiquant si la liste des cases forme une chaîne de cases adjacentes.

Un parcours sera qualifié de ***compatible avec le labyrinthe*** lorsqu'il s'agit d'une succession de cases adjacentes accessibles (non murées). On donne la fonction teste(cases, laby) qui indique si le chemin **cases** est un chemin possible compatible avec le labyrinthe **laby** :

```
def teste(cases, laby) :
    if not adjacentes(cases) :
        return False
    possible = True
    i = 0
    while i < len(cases) and possible:
        if mur(laby, cases[i][0], cases[i][1]) :
            possible = False
        i = i + 1
    return possible
```

3.1 Justifier que la boucle de la fonction précédente se termine.
3.2 En déduire une fonction echappe(cases, laby) qui indique par un booléen si le chemin **cases** permet d'aller de l'entrée à la sortie du labyrinthe **laby**.