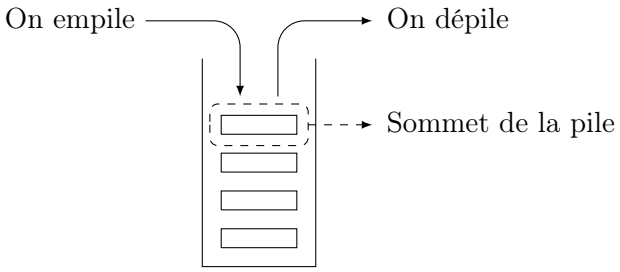
Exercice 1

Cet exercice porte sur la notion de pile et sur la programmation de base en Python.

On rappelle qu'une pile est une structure de données abstraite fondée sur le principe « dernier arrivé, premier sorti » :



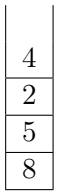
Une pile

On munit la structure de données Pile de quatre fonctions primitives définies dans le tableau ci-dessous. :

Structure de données abstraite : Pile
Utilise : Éléments, Booléen

Opérations :
— `creer_pile_vide : ∅ ⟶ Pile`
`creer_pile_vide()` renvoie une pile vide
— `est_vide : Pile ⟶ Booléen`
`est_vide(pile)` renvoie True si `pile` est vide, False sinon
— `empiler : Pile, Élément ⟶ Rien`
`empiler(pile, element)` ajoute `element` au sommet de la `pile`
— `depiler : Pile ⟶ Élément`
`depiler(pile)` renvoie l'élément au sommet de la `pile` en le retirant de la `pile`

Question 1 On suppose dans cette question que le contenu de la pile `P` est le suivant (les éléments étant empilés par le haut) :



Quel sera le contenu de la pile `Q` après exécution de la suite d'instructions suivante ?

```
1       Q = creer_pile_vide()
2       while not est_vide(P):
3           empiler(Q, depiler(P))
```

Question 2

1. On appelle *hauteur* d'une pile le nombre d'éléments qu'elle contient. La fonction `hauteur_pile` prend en paramètre une pile `P` et renvoie sa hauteur. Après appel de cette fonction, la pile `P` doit avoir retrouvé son état d'origine.

   Exemple : si `P` est la pile de la question 1 : `hauteur_pile(P) = 4`.

   Recopier et compléter sur votre copie le programme Python suivant implémentant la fonction `hauteur_pile` en remplaçant les `???` par les bonnes instructions.

```
def hauteur_pile(P):
    Q = creer_pile_vide()
    n = 0
    while not(est_vide(P)):
        ???
        x = depiler(P)
        empiler(Q,x)
    while not(est_vide(Q)):
        ???
        empiler(P, x)
    return ???
```

2. Créer une fonction `max_pile` ayant pour paramètres une pile `P` et un entier `i`. Cette fonction renvoie la position `j` de l'élément maximum parmi les `i` derniers éléments empilés de la pile `P`. Après appel de cette fonction, la pile `P` devra avoir retrouvé son état d'origine. La position du sommet de la pile est `1`.

   Exemple : si `P` est la pile de la question 1 : `max_pile(P, 2) = 1`

Question 3 Créer une fonction `retourner` ayant pour paramètres une pile `P` et un entier `j`. Cette fonction inverse l'ordre des `j` derniers éléments empilés et ne renvoie rien. On pourra utiliser deux piles auxiliaires.

Exemple : si `P` est la pile de laquestion 1(a), après l'appel de `retourner(P, 3)`, l'état de la pile `P` sera :

Question 4 L'objectif de cette question est de trier une pile de crêpes.

On modélise une pile de crêpes par une pile d'entiers représentant le diamètre de chaque crêpe. On souhaite réordonner les crêpes de la plus grande (placée en bas de la pile) à la plus petite (placée en haut de la pile).

On dispose uniquement d'une spatule que l'on peut insérer dans la pile de crêpes de façon à retourner l'ensemble des crêpes qui lui sont au-dessus.

Le principe est le suivant :

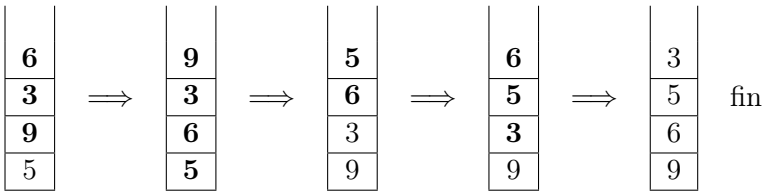
— On recherche la plus grande crêpe.

— On retourne la pile à partir de cette crêpe de façon à mettre cette plus grande crêpe tout en haut de la pile.

— On retourne l'ensemble de la pile de façon à ce que cette plus grande crêpe se retrouve tout en bas.

— La plus grande crêpe étant à sa place, on recommence le principe avec le reste de la pile.

Exemple :



Créer la fonction `tri_crepes` ayant pour paramètre une pile `P`. Cette fonction trie la pile `P` selon la méthode du tri crêpes et ne renvoie rien. On utilisera les fonctions créées dans les questions précédentes.

Exemple : Si la pile `P` est

7
14
12
5
8

, après l'appel de `tri_crepes(P)`, la pile `P` devient

5
7
8
12
14

.

Exercice 2

Cet exercice porte sur la programmation en général et la récursivité en particulier.

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 4 (4 points)

Cet exercice porte sur l'algorithmique et la programmation en Python. Il aborde les notions de tableaux de tableaux et d'algorithmes de parcours de tableaux.

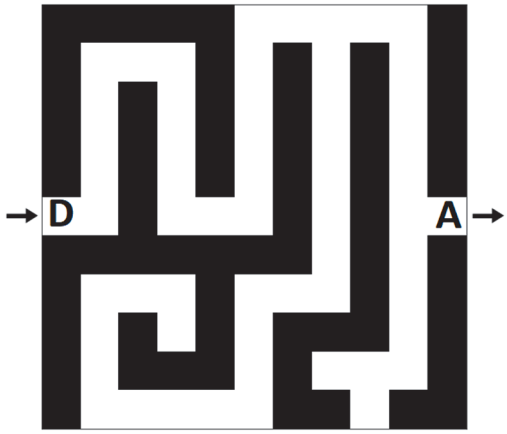
Partie A : Représentation d'un labyrinthe

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

Dans ce tableau :

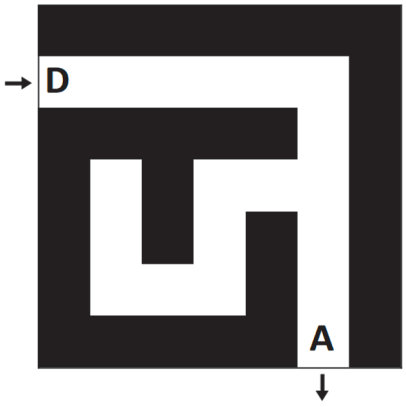
- 0 représente une case vide, hors case de départ et arrivée,
- 1 représente un mur,
- 2 représente le départ du labyrinthe,
- 3 représente l'arrivée du labyrinthe.

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`.




```
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]]
```

2. Écrire une fonction `est_valide(i, j, n, m)` qui renvoie `True` si le couple (i, j) correspond à des coordonnées valides pour un labyrinthe de taille (n, m), 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
```

3. 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])
    …
```

4. 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)]`.