Dans ce TP, on aura besoin de la commande `sorted(L)` qui renvoie une liste triée contenant les mêmes éléments que `L`. Dans le cas où `L` contient des chaînes de caractères, c'est ***l'ordre lexicographique*** qui est utilisé (analogue à l'ordre alphabétique). Lorsque `L` est une liste de couples, on se sert également de l'ordre lexicographique : la fonction `sorted` trie d'abord suivant le premier élément du couple puis, en cas d'égalité, utilise le deuxième élément.

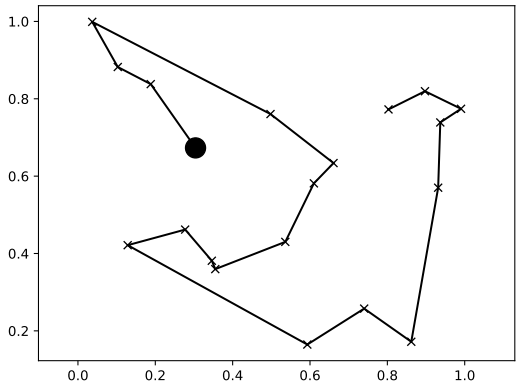
Exercice 1. Plus court chemin dans le plan

Soient $A_0, A_1, \dots, A_{n-1}$ des points du plan. Notre but est de relier tous les A_i sans jamais lever le stylo, tout en obtenant un tracé le plus court possible. Pour cela on adopte une stratégie gloutonne :

→ On part de A_0.

→ Tant que tous les points n'ont pas été visités :

- Soit A le dernier point atteint et E l'ensemble des points qui n'ont pas encore été visités.
- Soit B l'élément de E tel que la distance euclidienne entre A et B est minimale.
- On relie A et B par un segment.



Le graphique ci-contre donne un exemple où $n = 20$ (A_0 est matérialisé par le petit disque plein).

En Python, les coordonnées des points seront stockées dans une liste « `pts: list[int,int]` » telle que `pts[i]` $= (x, y)$ avec x l'abscisse de A_i et y son ordonnée. De plus, pour représenter l'ensemble E, on utilisera une liste de booléens `vis` telle que `vis[i]` vaut `True` si A_i a déjà été visité et `False` sinon.

1. Écrire une fonction qui prend en entrée la liste `pts` et renvoie un objet « `dist: list[list[float]]` » tel que pour tout `i, j` $\in [\![$`0, n - 1`$]\!]$, `dist[i][j]` est la distance euclidienne entre A_i et A_j.
2. Supposons qu'à une étape de l'algorithme, le dernier point visité soit le point A_i. Écrire une fonction

   `prochain_point(dist: list[list[float]], i: int, vis: list[bool]) -> int`

   qui renvoie l'indice `j` du point à visiter après A_i. On pourra supposer qu'il reste au moins un point non visité.
3. En déduire une fonction

   `relier_glouton(pts: list[int,int]) -> list[int]`

   qui renvoie une liste contenant les indices des points, en suivant l'ordre dans lequel ils sont visités.
4. Donner les complexités des différentes fonctions.

Afin de vérifier visuellement que les programmes écrits sont corrects, on place des points aléatoires sur un graphique, puis on les relie en suivant l'ordre donné par la fonction `relier_glouton`. Pour cela on utilisera le module `matplotlib.pyplot` :

```
import matplotlib.pyplot as plt

X = [...] # Liste des abscisses des points
Y = [...] # Liste des ordonnées des points

plt.figure()
plt.plot(X[0],Y[0],'o') # Dessine un disque sur le point initial.
plt.plot(X,Y,'x-')      # Dessine une croix sur chaque point
                        # et les relie par des segments.
plt.axis('equal')       # Fixe la même échelle pour l'axe des abscisses
                        # et des ordonnées.
plt.show()
```

5. Écrire une fonction qui prend en entrée l'entier n, génère n points aléatoires et les relie sur un graphique grâce à la fonction `relier_glouton`. On pourra utiliser la fonction `random` du module `random` qui renvoie un flottant aléatoire compris entre 0 et 1. Vérifier que l'ordre des points est bien celui attendu.
6. Trouver un exemple pour lequel l'algorithme glouton ne renvoie pas une solution optimale (c'est à dire que le tracé n'est pas de longueur minimale). On fera en sorte que le nombre de points n soit le plus petit possible.

Exercice 2. Programme TV

Homer souhaite regarder un maximum de programmes à la télévision avec les contraintes suivantes :

- Il ne peut regarder qu'un seul programme à la fois.
- Un programme doit être regardé en entier pour être comptabilisé.

Les horaires des programmes sont stockés dans une liste de couples d'entiers `L`. Chaque couple est de la forme `(d,f)` avec `d < f` où `d` est l'heure de début du programme et `f` l'heure de fin. Par exemple, si `L = [(1,4), (7,17), (4,6), (2,4), (3,4)]`, cela signifie qu'il y a cinq programmes à la télévision (voir le tableau ci-contre). On propose deux stratégies pour sélectionner les programmes.

	Début	Fin	Durée
Programme 0	1	4	3
Programme 1	7	17	10
Programme 2	4	6	2
Programme 3	2	4	2
Programme 4	3	4	1

Stratégie 1. Homer sélectionne les programmes en choisissant en priorité les plus courts.

Avec l'exemple du tableau, si Homer suit la stratégie 1 alors il choisit d'abord de regarder le programme 4 puisque c'est le plus court. Il ne pourra donc pas regarder les programmes 0 et 3 qui sont diffusés en même temps que le programme 4. Parmi les programmes restant (1 et 2), il décide de regarder le programme 2 puisque c'est le plus court. Finalement, le seul programme restant est le programme 1 et Homer pourra le regarder. Au total, Homer regardera trois programmes (4, 2 et 1) s'il suit la stratégie 1.

Stratégie 2. Homer sélectionne les programmes en choisissant en priorité ceux qui commencent le plus tôt.

Avec l'exemple du tableau, si Homer suit la stratégie 2 alors il choisit d'abord de regarder le programme 0 puisque c'est celui qui commence le plus tôt. Il ne pourra donc pas regarder les programmes 3 et 4 qui sont diffusés en même temps que le programme 0. Parmi les programmes restant (1 et 2), il décide de regarder le programme 2 puisque c'est celui qui commence le plus tôt. Finalement, le seul programme restant est le programme 1 et Homer pourra le regarder. Au total, Homer regardera trois programmes (0, 2 et 1) s'il suit la stratégie 2.

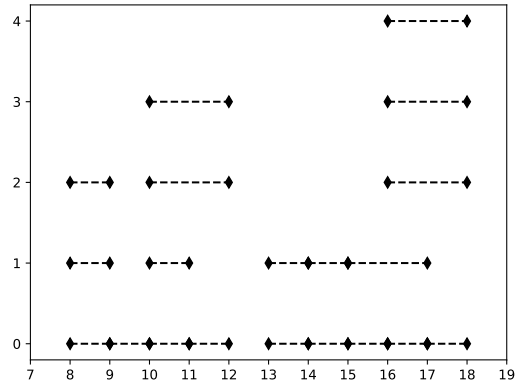
1. Montrer que les stratégies 1 et 2 ne sont pas optimales (c'est à dire que dans certains cas, il est possible de regarder plus de programmes que ne l'indiquent ces stratégies).
2. Déterminer une stratégie optimale et montrer qu'elle est bien optimale. On pourra s'inspirer de la preuve d'optimalité vue en cours.
3. Écrire en Python une fonction qui implémente la stratégie optimale. Pour tester la fonction, on pourra générer des exemples aléatoires.

Exercice 3. Allocation de salles de cours

Un lycée souhaite déterminer le nombre de salles nécessaires pour que tous les cours prévus puissent avoir lieu. On dispose initialement d'une liste de couples d'entiers notée `cours`. Chaque couple (d, f) de cette liste correspond à l'un des cours, d étant l'heure de début et f l'heure de fin. Pour chaque salle, on souhaite créer une liste « `L: list[int,int]` » contenant les horaires des cours qui y auront lieu. Les listes `L` seront stockées dans une variable « `salles: list[list[int,int]]` ».

Dans l'exemple ci-dessous, la liste `salles` indique que 5 salles sont utilisées. La première salle est utilisée pour 9 cours de 1 heure ; elle est occupée de 8h à 12h, puis de 13h à 18h. La deuxième salle est utilisée pour 4 cours de 1 heure et 1 cours de 2 heures etc ...

```
salles = [
    [(8,9), (9,10), (10,11), (11,12), (13,14),
     (14,15), (15,16), (16,17), (17,18)],
    [(8,9), (10,11), (13,14), (14,15), (15,17)],
    [(8,9), (10,12), (16,18)],
    [(10,12), (16,18)],
    [(16,18)]
]
```



Afin de créer la liste `salles` à partir de la liste `cours`, on utilise un algorithme glouton. Pour cela, on commence par trier les cours par ordre croissant d'heure de fin grâce à la fonction `sorted` de Python. La liste ainsi obtenue est notée `cours_tries`. Ensuite, pour chaque élément de `cours_tries`, on détermine la première salle libre toute la durée du cours, puis on ajoute le cours dans la sous-liste correspondante.
Par exemple, si la liste `cours` vaut

`[(16,18), (15,16), (14,17), (9,11)]`,

alors la liste `cours_tries` vaut

`[(9,11), (15,16), (14,17), (16,18)]`.

Voici l'évolution de la liste `salles` au cours de l'algorithme :

	`salles`
Initialisation	`[]`
Étape 1	`[[(9,11)]]`
Étape 2	`[[(9,11), (15,16)]]`
Étape 3	`[[(9,11), (15,16)], [(14,17)]]`
Étape 4	`[[(9,11), (15,16), (16,18)], [(14,17)]]`

1. Écrire une fonction `trier_cours` qui prend en entrée la liste `cours` et renvoie la liste `cours_tries`.
2. Écrire une fonction `trouver_salle` qui prend en entrée la liste `salles` ainsi que deux entier `d,f`, et indique l'indice de la première salle qui peut accueillir un nouveau cours dont l'heure de début est `d` et dont l'heure de fin est `f`. Si aucune salle de la liste n'est compatible, la fonction renverra `len(salles)`.
3. Écrire une fonction `allocation_glouton` qui prend en entrée la liste `cours` et renvoie la liste `salles`.

Afin de tester nos fonctions, nous allons générer aléatoirement la liste `cours` en suivant les règles suivantes :

- En moyenne, 7 cours sur 10 durent une heure, les autres durent deux heures.
- Aucun cours n'a lieu entre 12h et 13h.
- Les cours commencent au plus tôt à 8h et terminent au plus tard à 18h.

4. Écrire une fonction `generer_cours` qui prend en entrée un entier `n` et renvoie une liste `cours` de taille `n` en suivant les règles données ci-dessus. On pourra utiliser la commande `random.choice(L)` qui renvoie un élément aléatoire de la liste `L` ; ainsi que la commande `random.random()` qui renvoie un flottant aléatoire compris entre 0 et 1.
5. Écrire une fonction `tracer_allocations` qui prend en entrée la liste `salles` et dessine les horaires d'occupation des salles. Un exemple est donné au début de l'énoncé. On pourra s'inspirer du programme suivant :

```
import matplotlib.pyplot as plt

plt.figure()
plt.plot([1,2], [3,4], '--', color = 'black')
plt.plot([2], [5], 'd', color = 'black')
plt.show()
```

Exercice 4. Rendu de monnaie – Optimalité de l'algorithme glouton

Dans le cours, on a vu que l'algorithme glouton de rendu de monnaie ne donne pas nécessairement une solution optimale. Dans cet exercice, on souhaite montrer que la solution de l'algorithme glouton est optimale dans le cas du système monétaire européen où les pièces prennent leurs valeurs dans l'ensemble $V = \{1, 2, 5, 10, 20, 50, 100, 200\}$.

Dans la suite, on fixe une somme $s \in \mathbb{N}$ et on suppose avoir à notre disposition une solution optimale au problème de rendu de monnaie pour s. Formellement, si on définit :

$$A = \left\{ (n_1, n_2, n_5, n_{10}, n_{20}, n_{50}, n_{100}, n_{200}) \in \mathbb{N}^8 : \sum_{v \in V} n_v \times v = s \right\}$$

alors une solution optimale au problème du rendu de monnaie est un 8-uplet $(n_1, \ldots, n_{200}) \in A$ qui minimise $\sum_v n_v$. Dans cette formalisation, l'entier n_v représente le nombre de pièces de valeur v qui sont rendues.

1. Soit $(n_1, \ldots, n_{200}) \in A$ une solution optimale au problème de rendu de monnaie pour s.
   (a) Montrer que :

   $$n_1 \in \{0, 1\}, \quad n_2 \in \{0, 1, 2\}, \quad n_5 \in \{0, 1\}, \quad n_{10} \in \{0, 1\},$$
   $$n_{20} \in \{0, 1, 2\}, \quad n_{50} \in \{0, 1\}, \quad n_{100} \in \{0, 1\}.$$

   (b) Supposons $s \geqslant 1$. On note $v \in V$ la plus grande valeur de pièce telle que $v \leqslant s$. Montrer que $n_v \geqslant 1$.
2. En utilisant une récurrence forte sur s, montrer que l'algorithme glouton renvoie une solution optimale.

Exercices à rendre au plus tard le 19/05/2024 à 20h

Exercice 5. Déménagement

On dispose de cartons de déménagement pouvant contenir des objets. Chaque carton a une capacité `c` $\in \mathbb{N}^*$ (indépendante du carton) et chaque objet à un poids `p` $\in [\![1\,;\,c]\!]$ (dépendant de l'objet). À tout instant, la somme des poids des objets se trouvant dans un même carton doit être inférieure ou égal à `c`. L'ensemble des cartons est représenté par une variable « `cartons: list[int]` » où `cartons[i]` $\in [\![1\,;\,c]\!]$ est la somme des poids des objets se trouvant dans le carton numéro `i`. Par exemple, si `cartons = [13, 20, 13, 18, 12]`, cela signifie qu'il y a cinq cartons et que la somme des poids des objets se trouvant dans le carton numéro `3` est `18`.

Pour ranger un nouvel objet de poids `p`, on adopte une stratégie gloutonne. Si l'objet ne rentre dans aucun carton, on place l'objet dans un nouveau carton vide. Sinon, on place l'objet dans le premier carton dans lequel il peut rentrer. En reprenant la liste `[13, 20, 13, 18, 12]`, supposons que la capacité des cartons soit `c = 20` et qu'on veuille y ranger un nouvel objet de poids `p = 7`. Alors le nouvel objet est ajouté dans le carton d'indice `0` et la liste `cartons` devient `[20, 20, 13, 18, 12]`. Si par la suite, on souhaite ajouter un nouvel objet de poids `9`, alors il faut utiliser un nouveau carton et la variable `cartons` devient `[20, 20, 13, 18, 12, 9]`.

1. Écrire une fonction

   ```
   etape_dem(cartons: list[int], p: int, c: int) -> NoneType
   ```

   qui prend en entrée la variable `cartons`, le poids `p` d'un nouvel objet, ainsi que la capacité `c` des cartons ; et applique l'algorithme glouton pour ranger le nouvel objet. Votre fonction doit modifier la liste `cartons` et ne rien renvoyer.

2. Écrire une fonction

```
dem_glouton(n: int, c: int) -> NoneType
```

qui prend en entrée un entier `n` ainsi que la capacité `c` des cartons. Cette fonction doit d'abord définir une liste `cartons` vide, puis créer `n` objets et les ajouter dans les cartons. Le poids de chaque objet est choisi aléatoirement entre `1` et `c` à l'aide de la fonction `randint` du module `random`. Votre fonction doit afficher dans la console l'évolution des cartons. Par exemple, avec `n = 10` et `c = 20`, on pourra obtenir l'affichage ci-contre.

```
10 -> [10]
4 -> [14]
8 -> [14, 8]
14 -> [14, 8, 14]
11 -> [14, 19, 14]
13 -> [14, 19, 14, 13]
4 -> [18, 19, 14, 13]
8 -> [18, 19, 14, 13, 8]
16 -> [18, 19, 14, 13, 8, 16]
6 -> [18, 19, 20, 13, 8, 16]
```

3. Quelle est la complexité de la fonction `dem_glouton` ? Justifier.
4. Montrer sur un exemple que le nombre de cartons obtenus avec l'algorithme glouton n'est pas forcément minimal.

Exercice 6. Le problème du sac à dos

On dispose de n objets stockés dans un dictionnaire `obj` de type `dict[str, (int,int)]`. Les valeurs du dictionnaire sont des couples (v, p) où $v \in \mathbb{N}^*$ est la valeur de l'objet et $p \in \mathbb{N}^*$ son poids. Par exemple dans le dictionnaire ci-dessous, le smartphone a une valeur de 150 et un poids de 3 :

```
obj = {
    "gourde": (10, 3), "lingot": (10000, 10), "stylo" : (1, 1),
    "montre": (90, 2), "lunettes": (40, 1), "smartphone": (150, 3),
    "ordinateur": (390, 8)
}
```

Notre but est de déterminer la valeur maximale pouvant être emportée dans un sac à dos supportant un poids maximum $P \in \mathbb{N}^*$. Avec le dictionnaire précédent et $P = 18$, la valeur maximale qu'il est possible d'emporter est 10 390 en prenant l'ordinateur et le lingot.

Algorithme glouton. Le principe de l'algorithme glouton est de sélectionner en priorité les objets dont le rapport poids/valeur est minimal. On commence donc par créer une liste triée « `P: list[float,str]` » contenant pour chaque objet s de valeur v et de poids p, le couple $(\frac{p}{v}, s)$. Afin de trier la liste, on pourra utiliser la fonction `sorted` de Python. Pour l'exemple ci-dessus, on obtient :

```
P = [
    (0.001, 'lingot'), (0.02, 'smartphone'),
    (0.020512820512820513, 'ordinateur'), (0.022222222222222223, 'montre'),
    (0.025, 'lunettes'), (0.3, 'gourde'), (1.0, 'stylo')
]
```

Supposons maintenant que la liste `P` ait été construite. Tant qu'il est possible d'ajouter au moins un objet dans le sac, l'algorithme glouton sélectionne l'objet dont le rapport poids/valeur est minimal parmi les objets ayant un poids inférieur à la capacité restante du sac. Par exemple pour $P = 18$, voici les différentes étapes de l'algorithme glouton :

- On ajoute le lingot et le smartphone dans le sac.
- L'ordinateur ne peut pas être ajouté car son poids est strictement supérieur à la capacité restante.
- On ajoute la montre et les lunettes dans le sac.
- La gourde ne peut pas être ajoutée car son poids est strictement supérieur à la capacité restante.
- On ajoute le stylo dans le sac.

Finalement, le sac contient une valeur totale de 10 281 et un poids total de 17.

1. Écrire une fonction

```
make_P(obj: dict[str, (int,int)]) -> list[float,str]
```

qui renvoie la liste triée `P`.

2. Écrire une fonction

```
sad_glouton(obj: dict[str, (int,int)], poids_max: int) -> (list[str], int, int)
```

qui prend en entrée le dictionnaire `obj` ainsi que l'entier P ; et renvoie la liste des objets sélectionnés par l'algorithme glouton ainsi que la valeur et le poids total de ces objets. On fera en sorte de ne parcourir la liste `P` qu'une seule fois.

3. Sans compter l'appel à la fonction `sorted`, quelle est la complexité de votre fonction ? Justifier.

La fin de l'exercice est facultative.

4. Dans cette question, on exécute l'algorithme glouton sur l'exemple donné au début de l'énoncé. Déterminer les 5 valeurs de P pour lesquels votre fonction ne renvoie pas une solution optimale. Justifier.
5. Trouver un exemple avec $n = 3$ objets où l'algorithme glouton ne renvoie pas une solution optimale. Justifier.

Algorithme brute force. La solution renvoyée par l'algorithme glouton n'étant pas optimale, on souhaite déterminer une solution optimale quitte à utiliser une fonction très lente. La stratégie ***brute force*** consiste à tester toutes les configurations possibles. L'idée est de générer toutes les listes « `M: list[str]` » telles que la somme des poids des objets de `M` est inférieure à P. Avec l'exemple du début de l'énoncé et $P = 3$, voici toutes les possibilités pour `M` :

`[]`	`["gourde"]`	`["smartphone"]`
`["stylo"]`	`["lunettes"]`	`["montre"]`
`["montre", "stylo"]`	`["lunettes", "stylo"]`	`["montre", "lunettes"]`

Il suffit ensuite de déterminer la liste `M` pour laquelle la somme des valeurs des objets est maximale.

6. Écrire une fonction

```
sad_BF(obj: dict[str, (int,int)], poids_max: int) -> (list[str], int, int)
```

qui résout le problème du sac à dos avec un algorithme brute force. Pour générer toutes les listes `M`, on pourra utiliser une fonction récursive et la commande « `del d[c]` » qui supprime la clé `c` dans le dictionnaire `d`.

Remarque. Une solution optimale peut être calculée de manière plus efficace en utilisant une technique de programmation appelée ***programmation dynamique***.