L'Ordre, l'Ordre, l'Ordre.

Consignes

Les consignes suivantes **doivent** être respectées sous peine d'être pénalisé par des **points négatifs**.

Le code doit être clair.

- Évitez les lignes à rallonge. S'il faut introduire une variable intermédiaire, faîte le.
- Nommez vos variables avec du sens. Évitez toutefois des noms trop longs. Auquel cas décrivez le rôle de la variable en commentaire.
- Ne surchargez pas de commentaires. Les commentaires utiles sont :
  - Description du rôle d'une variable
  - Description du rôle d'une boucle complexe.

Soignez la présentation de votre code.

- Marquez clairement les indentations.
- Éviter les ratures trop nombreuses dans un code. Une ou deux, maximum.
- Écrivez votre instruction sur plusieurs lignes si elle est longue en ajoutant simplement une indentation après le retour à la ligne.

Respectez la syntaxe Python. - *0 à l'exercice si syntaxe inventée.*

- Vérifiez que ce que vous écrivez respecte la syntaxe.
- Les petites erreurs n'implique pas 0 (oublie de :, = au lieu de ==,…)
- Si vous avez une idée, mais que vous n'arrivez pas à l'écrire, n'inventez pas. Écrivez votre idée en commentaire.
- Seuls les instructions de la fiche de syntaxe sont autorisées.

Pour nommer une fonction et ses entrées, utilisez les noms donnés dans l'énoncé - *0 à l'exercice sinon*

Utilisation de `print` ou de `input` interdite - *0 à l'exercice sinon*

I Algorithmes de cours

Question 1. *Second Minimum*

Entrées : Liste d'entiers positifs, disjoints, `l` de taille supérieure à 2
Sortie : Le second minimum de `l`.

Question 2. *Recherche linéaire - version for*

Entrées : Liste `l` et un objet `x`
Sortie : L'indice de la première occurrence de `x` dans `l`, ou `-1` si `x` n'est pas dans `l`.

II L'ordre ...

Question 3. *Minimum*

Implémenter une fonction `minimum(l)` qui prend une liste d'entiers `l` et renvoie l'indice du minimum de `l`. Si `l` est vide, la fonction devra renvoyer `None`.

Question 4. *Filtrage*

Implémenter une fonction `plus_petits(l,x)` qui prend une liste d'entiers `l` et un entier `x` et renvoie la liste des entiers de `l` plus petits ou égaux à `x`.

III ... l'ordre ...

Question 5. *Nadine*

Voici une fonction `nadine(l)` qui prend en entrée une liste d'entiers `l` et renvoie en sortie une liste d'entiers.

```
1  def nadine(l) :
2      n = len(l)
3      l_res = [None] * n
4      for x in l : # Boucle 1
5          c = 0
6          for y in l : # Boucle 2
7              if y < x :
8                  c = c + 1
9          l_res[c] = x
10     return l_res
```

Soit `l = [4, 13, 0, 26, 25, 6, 10, 16]`.

1. Lister les valeurs de `x` avant chaque exécution de la ligne 5.
2. Combien de fois la ligne 7 sera-t-elle exécutée ?
3. Combien de fois la ligne 8 sera-t-elle exécutée ?
4. Que vaut `l_res` au début de la 5^e itération de la boucle 1.
5. Expliquer le but et le fonctionnement de la fonction `nadine`.

Question 6. *Est triée*

Implémenter une fonction `est_triee(l)` qui prend une liste d'entiers `l` en entrée et renvoie un booléen vrai si et seulement si `l` est triée dans l'ordre croissant.

IV … et l'ordre.

Question 7. *Ordre lexicographique*

Les chaînes de caractères peuvent être ordonnées par ordre *lexicographique*, qu'on nomme aussi *alphabétique* lorsque les chaînes ne possèdent que des caractères alphabétiques.

On peut ordonner les caractères par code ASCII. Ainsi, on dira qu'un caractère `c` est plus petit qu'un caractère `d` ssi **`ord`**`(c) < `**`ord`**`(d)`, avec **`ord`** la fonction qui renvoie le code ASCII (entier entre 0 et 255) d'un caractère.

L'ordre est défini sur deux chaînes, `s1` et `s2`, tel que `s1 < s2` si et seulement si :

- `s1` est un préfixe strict de `s2`, ou
- le premier caractère de `s1` différent de celui de `s2` situé à la même position, est plus petit.

Par exemple :

- `aab < aabaa` (prefixe stric)
- `aghab < aghbb`
- `nadine < net`
- `zab < zba`

Sans utiliser les opérateurs de comparaison Python sur les chaînes de caractères, implémenter une fonction `lexico(s1,s2)` qui renvoie `-1` si `s1 <s2` , `0` si `s1 = s2` et `1` si `s2 < s1`.

Question 8. *Tri Sélection - Simulation*

L'algorithme de tri sélection prend en entrée une liste de chaînes de caractères `l` qu'on supposera *disjointes et non vides* et modifie `l` en sa version triée dans l'ordre croissant de l'ordre lexicographique.

Le principe du tri sélection est d'échanger le minimum de la liste avec l'élément en position 0, puis le second minimum avec l'élément en position 1, puis le troisième minimum avec l'élément en position 2, etc.

L'échange se fait dans la même liste. Par exemple voici la liste `l = ['a','b','c','d','e','f']`, puis la liste après échange des éléments en position 1 et 3 : `['a','d','c','b','e','f']`.

Soit `l = ['ead', 'nee', 'neead', 'eaz', 'nadine', 'e', 'net', 'nad']`.

Donnez la liste des échanges qui seront effectués par l'algorithme sur la liste `l` lors d'un tri sélection, dans l'ordre dans lequel ils seront effectués. Les échanges seront donnés sous la forme de couples `(i,j)`.

Question 9. *Tri Sélection - Code*

Compléter le code de la fonction suivante qui implémente le tri sélection.

```
def tri_selection(l) :
    n = len(l)
    dernier_trie = ""
    for nb_tours in # Trou 1 :
        min_tour = l[nb_tours]
        imin_tour = # Trou 2
        for i in range(n) :
            s = l[i]
            if lexico(dernier_trie,s) == -1 and # Trou 3:
                min_tour = s
                imin_tour = i
        # Trou 4
        dernier_trie = min_tour
    return None
```

Question 10. *Tri Insertion*

L'algorithme de tri insertion prend en entrée une liste de chaînes de caractères `l` qu'on supposera *disjointes et non vides* et modifie `l` en sa version triée dans l'ordre croissant de l'ordre lexicographique.

Le principe du tri insertion est de parcourir la liste de telle manière à ce que la liste soit triée jusqu'à l'indice `i` courant du parcours. Pour cela, lors du traitement de l'élément `i`, on réarrange la liste par échange pour placer l'élément `i` à la bonne place. Ainsi la liste sera triée jusqu'à l'indice `i+1` et on peut passer à l'élément suivant.

Soit `l = ['ead', 'aee', 'eaa', 'a', 'nad', 'e', 'nadine', 'na']`. Voici les différentes versions de la liste générée à la fin de chaque étape du parcours, la valeur de `i` est représentée par la `||` dans l'écriture :

1. `['ead', || 'aee', 'eaa', 'a', 'nad', 'e', 'nadine', 'na']`
2. `['aee', 'ead', || 'eaa', 'a', 'nad', 'e', 'nadine', 'na']`
3. `['aee', 'eaa', 'ead', || 'a', 'nad', 'e', 'nadine', 'na']`
4. `['a', 'aee', 'eaa', 'ead', || 'nad', 'e', 'nadine', 'na']`
5. `['a', 'aee', 'eaa', 'ead', 'nad', || 'e', 'nadine', 'na']`
6. `['a', 'aee', 'e', 'eaa', 'ead', 'na', 'nad', 'nadine'|| ]`

Implémenter la fonction `tri_insertion(l)` décrite ci-dessus.