## Fiche descriptive de la ressource préparée

| | |
|---|---|
| **Nom et prénom :** | **David GRANJON** |
| **Numéro du thème :** | 1 |
| **Intitulé du thème :** | Parcours séquentiel d'un tableau<br>*(recherche d'occurrence, d'extremum, calcul d'une moyenne etc…)* |
| **Résumé (5 lignes max) :** | A travers cette activité, les élèves vont implémenter un algorithme de recherche d'un extremum dans un tableau sous forme d'une fonction. Ils vont ensuite réutiliser leur fonction dans un cas concret, la rendre robuste, et en évaluer la complexité. Enfin, les élèves seront sensibilisés aux notions de terminaison et correction d'un algorithme. |
| **Cadre pédagogique (TD/TP/Projet, localisation, temporalité, etc.) :** | 1ère générale – Spécialité NSI (Numérique et Sciences Informatique)<br>2ème trimestre de la classe de Première<br>Activité (TP), élève seul aidé selon ses besoins par l'enseignant |
| **Durée (temps élève) :** | 2h |
| **Prérequis :** | *Langage Python, savoir :*<br>• *Affecter une variable, lire sa valeur et son type*<br>• *Utiliser des boucles bornées (for) ou non bornées (while)*<br>• *Importer une bibliothèque*<br>• *Manipuler les listes, lire et modifier les éléments d'une liste grâce à leurs index*<br>• *Utiliser et écrire une fonction/procédure* |
| **Compétences mobilisées :** | **Algorithmique** : Parcours séquentiel d'un tableau :<br>• Écrire un algorithme de recherche d'une occurrence sur des valeurs de type quelconque.<br>• Écrire un algorithme de recherche d'un extremum, de calcul d'une moyenne.<br>**Langages et programmation :** Spécification<br>• Prototyper une fonction.<br>• Décrire les préconditions sur les arguments (Utilisation d'assertions) |
| **Matériel nécessaire :** | 1 ordinateur avec un IDE Python installé comme par exemple Spyder, Pyzo, Edupython, … |
| **Mode d'évaluation (si applicable) :** | Notation après envoi des documents de l'activité par les élèves sur cloud ou par mail. Notation à l'aide d'un tableau excel/calc avec barême (non fourni) |
| **Sources (bibliographie, sitographie, ressources numériques etc.) :** | https://www.datacamp.com/community/tutorials/docstrings-python<br>https://www.ukonline.be/cours/python/apprendre-python/chapitre10-2 |

| NSI | **Algorithme de parcours séquentiel d'un tableau, notions de complexité, correction et terminaison.** | 2 h | **TP** |
|---|---|---|---|

## Introduction

Nous allons à travers cette activité, élaborer et implémenter un algorithme simple de recherche d'un minimum dans une liste. Puis nous le testerons et le réutiliserons dans différentes configurations.

Ensuite nous verrons ce qu'est la ***robustesse*** d'un algorithme/programme, et ferons en sorte qu'une utilisation incorrecte de notre programme n'entraîne pas de dysfonctionnement.

Enfin vous serez sensibilisés aux notions de ***complexité***, ***terminaison*** et ***correction*** d'un algorithme.

## 1ère Partie : élaboration/implémentation de l'algorithme de recherche d'un minimum

**1.1 -** Ouvrez avec un IDE Python le programme *ListeAlea.py*, testez-le, puis :

1) Complétez les commentaires présents en bout de chacune des lignes.
2) Rédigez le *docstring* de la fonction *genListe*.

**1.2 -** Dans le programme précédent, écrire une fonction appelée *trouveMin* à laquelle on passe en argument une liste de valeurs réelles en entrée, et qui nous renvoie **l'indice auquel se trouve la valeur minimale** de cette liste. Pour vous aider, posez-vous les questions suivantes :

- Par quelle valeur initialiser la variable qui stocke le minimum ?
- Comment parcourir toutes les valeurs de la liste ?
- Comment obtenir l'indice d'une valeur minimale trouvée ?

Pour tester votre fonction, vous pouvez décommenter les dernières lignes du programme.

**1.3 -** Une fois votre fonction *trouveMin* fonctionnelle, rédigez son docstring si ce n'est pas déjà fait, ainsi que les commentaires nécessaires en bout de ligne.

*Enregistrez votre programme sous* ***minListe_Partie1.py***

## 2ème Partie : Robustesse de la fonction *trouveMin*

*Enregistrez votre programme précédent sous* ***minListe_Partie2.py***

Lorsque l'on programme et que l'on définit par exemple une nouvelle fonction, il est nécessaire de la rendre résistante à de mauvaises utilisations, cela s'appelle la rendre **robuste**. Pour cela il faut déjà minimiser le nombre de conditions de son utilisation (ex : variable > 0 obligatoirement, ou de type string, …). Mais aussi avoir recours par exemple à **la programmation défensive** en utilisant des **assertions** afin de vérifier que les conditions restantes sont bien vérifiées.

**2.1 -** Supprimez la fonction *genListe* du programme, ainsi que les trois premières lignes du programme principal qui appelaient cette fonction.

Copiez les 5 lignes suivantes dans votre programme (juste avant l'appel de votre fonction *trouveMin*)

```
#maListe=42
#maListe=[(12,"mardi"),"Mars",2019]
#maListe=[(13,"mercredi"),(14,"jeudi"),(15,"vendredi"),(12,"mardi")]
#maListe=[("mercredi",13),("jeudi",14),("vendredi",15),("mardi",12)]
#maListe=["b","c","d","a","Z"]
```

Testez votre fonction *trouveMin* avec ces 5 listes successivement, conclure quant à la robustesse de votre fonction :

……

**2.2 -** Consultez *cette page* sur l'utilisation des assertions, puis faites en sorte (grâce à la commande *assert*) que lorsque la variable d'entrée de la fonction n'est pas une liste, le programme renvoi le message suivant : « *La variable envoyée à la fonction trouveMin doit être obligatoirement de type liste !*»

**2.3 -** De même, faites-en sorte que le programme vérifie que chaque élément de la liste est bien de type float ou int et affiche : « *Les éléments de la liste doivent être des nombres réels !* ». **Attention à ne pas refaire une boucle for ou while pour cette vérification !**

*Enregistrez votre programme*

## 3ème Partie : Cas concret d'utilisation de la fonction trouveMin

*Enregistrez votre programme précédent sous **minListe_Partie3.py***

Lors du lancement d'un ballon sonde, les données d'altitudes/températures en Celsius ont été relevées et stockées dans deux listes distinctes ci-dessous (l'altitude en m de rang 0 correspond à la t°C de même rang).

```
altitudes=[0, 200, 500, 1100, 1800, 2250, 2680, 5070, 6800, 8900, 9800, 11000, 13500, 15120,
17210, 19500, 20100, 21250, 23620, 24865]
temperatures=[15, 13.7, 11.7, 7.9, 3.4, 0.5, -2.3, -17.7, -28.9, -42.5, -48.3, -56, -56.2, -56.5,
-56.3, -56, -56.1, -53, -50.1, -48]
```

**3.1 -** Copier ces deux listes dans votre programme principal. En utilisant votre fonction *trouveMin* faites afficher par votre programme l'altitude où la température est la plus basse.

*Exemple de phrase affichée :* « *C'est à ??? m d'altitude que la température est la plus basse : ??? °C* »

*Enregistrez votre programme*

## 4ème Partie : Complexité temporelle de votre algorithme (fonction *trouveMin*)

Lorsque l'on écrit un algorithme, il est nécessaire de mettre en valeur la façon dont le temps d'exécution croît avec le nombre d'éléments **noté n** à traiter par l'algorithme. Cela s'appelle la **complexité** d'un algorithme.

Cet ordre de grandeur du temps d'exécution d'un algorithme, donne une caractérisation simple de l'efficacité de l'algorithme et permet également de comparer les performances relatives d'algorithmes servant à faire le même travail.

Appelons **T(n)** une fonction qui nous donne le temps d'exécution en fonction du nombre n de données à traiter. Chaque opération élémentaire (ex : Evaluation d'une expression, Affectation d'une valeur à une variable, Indexation d'une liste, …) a un coup temporel constant **égal à 1**.

**4.1 -** Compléter le tableau suivant (vous pouvez ajouter des lignes).
Pour cela : pour chacune des lignes de votre fonction *trouveMin*, comptez le nombre de fois où chaque opération élémentaire est effectuée pour une liste de n éléments à traiter. **On se place dans le pire des cas**, c'est-à-dire que la fonction traite par exemple une liste triée dans l'ordre décroissant.

| ***Ligne de la fonction*** | **Accès à un élément d'une liste** | **Affectation valeur à une variable** | **Test d'une expression** | **Accès au type d'une variable** | **Obtention longueur liste** | **Calculer nb d'itérations boucle for** | **Retour de la fonction** |
|---|---|---|---|---|---|---|---|
| *assert type(liste)==list,"……"* | | | 1 | 1 | | | |
| *assert (type(liste[0])==float or type(liste[0])==int),"……"* | 2 | | 2 | | | | |
| *min=liste[0]* | 1 | 1 | | | | | |
| *ind=0* | | 1 | | | | | |
| *for i in range (1,len(liste)):* | | | | | 1 | 1 | |
| *assert (type(liste[i])==float or type(liste[i])==int) ,"……"* | 2(n-1) | | 2(n-1) | 2(n-1) | | | |
| *if liste[i]<min:* | n-1 | | n-1 | | | | |
| *min=liste[i]* | n-1 | n-1 | | | | | |
| *ind=i* | | n-1 | | | | | |
| *return ind* | | | | | | | 1 |
| *TOTAL par colonne :* | 4n-1 | 2n | 3n | 2n-1 | 1 | 1 | 1 |
| *TOTAL :* | 11n+1 | | | | | | |

Expression du temps d'exécution en fonction de n : T(n) = 11n+1 (c'est l'équation d'une droite)

Le coût temporel d'exécution de l'algorithme est donc **linéaire**, on dit qu'il est en O(n) (prononcer « Grand O de n »), mais nous verrons cela ultérieurement.

D'autres algorithmes plus ou moins complexes peuvent avoir des coûts temporels différents. Voici les fonctions que l'on retrouve le plus couramment **dans l'ordre croissant du coût temporel** (lorsque n grandi) : **log n  n  n.log n  $n^2$  $n^3$  $2^n$  n!**

## 5ème Partie : Notion de terminaison d'un algorithme

En plus de l'évaluation de la complexité d'un algorithme (coût temporel, ou en mémoire), il est nécessaire de s'assurer qu'un algorithme se termine tôt ou tard, et ne rentre pas dans une boucle infinie par exemple. Dans une première approche, nous allons essayer de nous poser les bonnes questions.

**5.1 -** Combien y'a-t-il de passages dans la boucle utilisée par votre fonction *trouveMin* ? Est-ce que cette valeur est finie à chaque fois ? Conclure quant à la terminaison de votre algorithme/fonction.

……..

## 6ème Partie : Notion de correction d'un algorithme

Enfin, il est parfois nécessaire de démontrer que l'algorithme renvoie bien ce qui lui est demandé, c'est-à-dire qu'il répond bien aux spécifications. Comme précédemment, dans une première approche, nous allons essayer de nous poser les bonnes questions.

Dans votre fonction *trouveMin* (justifier vos réponses):

**6.1 -** Est-ce que le minimum est initialisé ? Par quoi ? A quel moment ?

……..

**6.2 -** Le nombre d'itérations de la boucle est-il correct ? La liste est-elle parcourue dans son ensemble ? (Dans le cas d'une boucle *while*, la condition est-elle bien choisie ?)

……..

**6.3 -** Comment et quand est modifié l'indice du minimum dans la boucle ? Est-ce judicieux et juste ?

……..

**6.4 -** Conclure quant à la véracité de l'indice du minimum donné par le programme. Cela s'appelle la preuve de correction.

……..

## POUR LES PLUS RAPIDES :

1. Modifier et implémenter votre algorithme afin de rechercher le minimum d'un tableau à deux dimensions (liste de listes en python). Testez-le avec un ou deux tableaux à deux dimensions que vous créerez pour l'occasion.
2. Evaluez maintenant la complexité de votre algorithme (voir Q4.1). De quel ordre (voir fonctions ci-dessous) est la fonction indiquant l'évolution du coût temporel en fonction de n données en entrée O(?) :

   **log n  n  n.log n  $n^2$  $n^3$  $2^n$  n!**
3. Conclure quant au coût temporel pour la recherche d'un extremum entre un tableau à 1D et 2D.