Simon ARMAND R5A04

Sofia GRIBANOVA Groupe 3A

Mini-projet 1 :

Trouver un élément commun entre 2 listes d’entiers

Sommaire

[Introduction 2](#_Toc185196797)

[Algorithme 1 : 2](#_Toc185196798)

[Modification de l’algorithme en utilisant une boucle ‘‘while’’ 3](#_Toc185196799)

[Algorithme 2 : Boucle ‘‘while’’ 3](#_Toc185196800)

[Performances de l’algorithme 3](#_Toc185196801)

[Nouvel algorithme avec structure de tas 4](#_Toc185196802)

[Comparaison des performances 5](#_Toc185196803)

## Introduction

L’objectif de ce mini-projet est de créer et implémenter un algorithme permettant de trouver un élément commun entre deux listes d’entier. Nous l’implémenterons d’abord en utilisant une structure de liste classique, et ensuite utilisant une structure de tas, puis nous comparerons les performances des deux structures.

Nous partirons de l’algorithme donné ci-dessous pour le modifier puis l’implémenter.

Algorithme 1 :

**Data :** Deux listes d’entiers *L1*, *L2*

**Résultat :** *x* un élément commun aux deux listes s’il y en a, sinon *NAN*

**Etape 1 :** On détermine *m1* le plus petit élément de *L1* et *m2* le plus petit élément de *L2*;

**Etape 2 :** Si *m1* = *m2* alors on pose *x = m1* et on s’arrête;

**Etape 3 :** Si *m1 < m2* on enlève *m1* de *L1*;

**Etape 4 :** Si *m2 < m1* on enlève *m2* de *L2*;

**Etape 5 :** Si les 2 listes sont non vides on retourne à l’étape 1;

**Etape 6 :** Si (au moins) une liste est vide on s’arrête en renvoyant *NAN*;

## Modification de l’algorithme en utilisant une boucle ‘‘while’’

Voici ci-dessous l’algorithme donné dans l’introduction modifié pour ajouter l’utilisation d’une boucle ‘‘while’’, permettant par la suite une implémentation plus rapide.

### Algorithme 2 : Boucle ‘‘while’’

**Data :** Deux listes d’entiers *L1, L2*

**Résultat :** *x* un élément commun aux deux listes s’il y en a, sinon *NAN*

**Tant que** aucunes des 2 listes n'est vides :

**Etape 1 :** On détermine *m1* le plus petit élément de *L1* et *m2* le plus petit élément de *L2*;

**Etape 2 :** Si *m1 = m2* alors on pose *x = m1* et on s’arrête;

**Etape 3 :** Si *m1 < m2* on enlève *m1* de *L1*;

**Etape 4 :** Si *m2 < m1* on enlève *m2* de *L2*;

**Fin Tant que**

**Etape 5 :** On renvoie *NAN*;

## Performances de l’algorithme

Une image contenant texte, Tracé, ligne, diagramme

Description générée automatiquement Une fois l’algorithme implémenté (fonction ‘‘element\_commun1(L1, L2)’’ du fichier element\_commun1.py) et testé (fonction ‘‘test\_element\_commun1()’’ du fichier test\_element\_commun1.py), on fait un graphe permettant de visualiser ses performances (fonction ‘‘afficher\_graph1(n)’’ du fichier test\_element\_commun1.py).

## Nouvel algorithme avec structure de tas

Pour essayer une nouvelle approche, nous changeons ici le fonctionnement en liste de l’algorithme 2 pour essayer avec une structure pas tas. L’idée est d’avoir des valeurs déjà trié, permettant d’optimiser la recherche du minimum utilisé dans l’algorithme 2.

En chronométrant les performances de cette approche, nous nous sommes rendu compte que les résultats n’étaient pas très parlants, car les temps étaient soit 0.5, soit 0. Il a donc fallu lisser les valeurs en utilisant la fonction ‘‘np.convolve’’ pour l’affichage du graphe.

Une image contenant texte, ligne, Police, Tracé

Description générée automatiquement

## Comparaison des performances

Une image contenant texte, ligne, Tracé, diagramme

Description générée automatiquement Pour conclure, voici le graphe comparant les deux approches. Nous avons choisi de lisser les résultats dans les deux cas pour avoir un graphe plus lisible.

On voit bien que l’approche avec la structure de tas est beaucoup plus efficace, ce qui est au final logique, car la partie la plus longue de l’algorithme est la recherche du minimum, partie qui est très rapide en tas car les valeurs sont déjà triées.