Projet d'analyse des algorithmes et validation des programmes

sujet : *plus grande sous-séquence croissante dans une liste*

IATIC 4

Réalisé par : **Driss Nait Belkacem**

Introduction

Ce rapport explique les différentes étapes du projet, il permet aussi de décrire la gestion du projet ainsi que les différentes étapes de de développement. Nous allons présenter les différences entre les différents algorithmes implémentés et les différentes solutions apportées.
Ce rapport permettra donc de suivre l'évolution du projet.

Description du projet et objectif

L'objectif de notre projet est l'implémentation de différents algorithmes du même problème qui est dans notre cas le problème de "la plus longue sous-séquence strictement croissante dans une liste", on peut décrire le problème de la manière suivante :

Étant donné une séquence de n entiers, imaginons que l'on y retrouve k séquences avec des nombres strictement croissants, l'objectif est de garder celle composée du maximum de nombres, autrement dit la plus grande.

Si plusieurs sous-séquences possèdent la taille la plus longue, l'algorithme retourne n'importe laquelle.

L'algorithme contient des appels récursifs qui ont pour but de réduire un problème en sous-problèmes.

Ce problème a été choisi parce que c'est un sujet riche dans on peut implémenter plusieurs algorithmes différents qui peuvent résoudre le problème, et cela est intéressant au niveau d'études de complexité et de temps d'exécution des programmes ainsi que l'espace mémoire utilisé par chacun des programmes, ce qui nous permettra de faire des comparaisons, voir les différences, et essayer des améliorations sur les programmes.

Environnement de travail

En ce qui concerne l'environnement de développement, j'ai choisi de travailler sur un système d'exploitation sous Linux.

Le projet a été développé sous Python 3.8.5.

Implémentation

Algorithme récursif

```
def plus_longue_sequence_recursif(liste):
        max_sequence = []
        for i in range(len(liste)):
                sequence = plus_longue_sequence_a_partir_du_premier_element(liste[i:len(liste)])
                if(len(sequence) > len(max_sequence)):
                        max_sequence = sequence
        return max_sequence
```

Figure 1 : algorithme de la plus longue sous-séquence dans une liste version récursif

L'algorithme récursif fait appel à chaque itération à la fonction "plus_longue_sequence_a_partir_du_premier_element" qui prend la liste à partir de l'élément i

```
def plus_longue_sequence_a_partir_du_premier_element(liste):
        result = [liste[0]]
        for i in range(1, len(liste)):
                if(liste[i] > liste[0]):
                        sequence = [liste[0]] + plus_longue_sequence_a_partir_du_premier_element(liste[i: len(liste)])
                        if(len(sequence) > len(result)):
                                result = sequence
        return result
```

Figure 2 : Fonction "plus_longue_sequence_a_partir_du_premier_element" utilisée par l'algorithme récursif

La fonction "plus_longue_sequence_a_partir_du_premier_element" permet d'avoir la plus longue séquence démarrant du premier élément de la liste passée en paramètre

Algorithme itératif

```
def plus_longue_sequence_iteratif(liste):
        elements_to_process = []
        for i in range(len(liste)):
            elements_to_process.append([ListItem(liste[i], i)])
        longest_sequence = []
        while elements_to_process:
            element = elements_to_process.pop()
            for i in range(element[len(element) - 1].position + 1, len(liste)):
                for j in range(i, len(liste)):
                    if liste[j] > element[len(element) - 1].item:
                        elements_to_process.append(element + [ListItem(liste[j], j)])
                        if len(longest_sequence) < len(element) + 1:
                            longest_sequence = element + [ListItem(liste[j], j)]

        return list(map(lambda list_item: list_item.item, longest_sequence))
```

Figure 2 : Deuxième algorithme résolvant le problème de la plus longue sous séquence dans une liste

L'algorithme utilise une structure "ListItem" que nous manipulons pour le traitement des éléments de la liste, elle nous servira pour mémoriser l'élément et sa position dans la liste principale.

```
class ListItem:
    def __init__(self, item, position):
        self.item = item
        self.position = position

    def __str__(self):
        return "[" + str(self.item) + "," + str(self.position) + "]"
```

La figure 2 représente le deuxième algorithme, ce dernier n'utilise pas les appels récursifs, ce dernier fonctionne de la manière suivante :

- Parcourt tous les éléments de la liste pour les charger dans la liste avec une boucle for
- Il fait une boucle while qui parcourt la liste elements_to_process qui permet de parcourir les séquences
- au sein de la boucle while, il utilise une boucle for avec l'itérateur i allant jusqu'à fin de la liste liste.
- il fait une troisième boucle for imbriqué allant de i jusqu'au dernier élément de la liste liste.
- il fait des conditionnelles au sein de la boucle qui vont permettre de garder à chaque itération la plus longue sous-séquence et la mettre à jour au fur et à mesure.

- Retourne la liste des éléments en sélectionnant juste les items (valeurs de l'items) parce qu'on n'aura pas besoin de leurs indices à la fin puisque l'on cherchera la plus longue séquence au niveau des valeurs des entiers.

Algorithme avec la recherche dichotomique

Le troisième algorithme implémenté est celui proposé par Wikipédia.

```
Entrée : X, un tableau indicé de 1 à n.
Sortie : L, longueur de la plus longue sous-suite strictement croissante de X.
         P, tableau de prédécesseurs permettant de reconstruire explicitement la suite.

P = tableau indicé de 1 à n
M = tableau indicé de 0 à n

L = 0
M[0] = 0
pour i = 1, 2, ..., n :
   par recherche dichotomique, trouver le plus grand entier j tel que 1 ≤ j ≤ L
     et X[M[j]] < X[i] ou définir j = 0 s'il n'en existe aucun.
   P[i] = M[j]
   si j == L ou X[i] < X[M[j + 1]] :
      M[j + 1] = i
      L = max(L, j + 1)
```

Figure 3 : Troisième algorithme résolvant le problème de la plus longue sous séquence dans une liste (proposé par Wikipédia)

L'algorithme ci-dessus se trouve implémenté dans le fichier "src/algo3.py", cet algorithme applique la recherche dichotomique sur les suites, L'idée consiste à parcourir le tableau de gauche à droite en conservant à chaque itération des séquences optimales de longueur 1 à L.

Premiers Tests

Avant de commencer la comparaison entre les algorithmes, il était nécessaires de s'assurer que les trois résolu le même problème, donc j'ai effectué plusieurs tests sur plusieurs listes, par exemple sur la liste suivante :

[10, 15, 7, 19, 2, 5, 7, 16, 3, 9, 15, 0, 1, 15, 6, 11, 0, 14, 7, 9]

Les trois algorithmes retournent la même sous séquence de la même longueur qui est la suivante :

[2,5,7,9,11,14]

La sous séquence valide les critères de l'algorithme, elle est bien strictement croissante et c'est elle la plus longue dans la liste.

Validité

Spécification

De la partie précédente, on peut constater la spécification du problème que l'on peut définir comme cela :
supposons que *i* représente un élément de la liste de sortie, *L2* est la liste en question.

$$\forall\ i \in L2,\ L2[i-1] < L2[i] < L2[i+1],\ |L2| <= |L|$$

Les algorithmes parcourent la liste afin de trouver la plus longue séquence, ils trouvent différentes séquences à chaque itération de manière différentes, l'objectif final est d'obtenir la séquence la plus grande qui valide la spécification. Pour cela, chacun des algorithmes précède différemment.

-PREMIER ALGORITHME

Le deuxième algorithme conserve à chaque itération *i* que la meilleure sous séquence trouvée au moment de l'itération, ce dernier donc modifie et met à jour à chaque fois la sous séquence.

Le troisième algorithme procède en utilisant la recherche dichotomique.

Le premier point commun entre les trois algorithmes c'est qu'ils ont tous besoin d'une liste en entrée, et qu'ils donnent tous un résultat qui valide la spécification à la fin.

la taille de la liste de sortie peut être inférieur ou égale à la taille de la liste d'entrée, la taille de la sous-séquence de sortie peut être égale à la taille de la liste d'entrée si les éléments de cette dernière sont déjà strictement croissante, la séquence de sortie des programmes ne peut pas avoir une taille supérieur à

celle de la liste de base, on représente cela dans la spécification par “|L2|<=|L|”, L2 représente la liste de sortie des programmes et L la liste d’entrée.

Un jeu de test a été implémenté afin de vérifier que les sorties des algorithmes valident vraiment la spécification, cela est expliqué dans la partie “des tests approfondis pour le recueil des données”.

Invariant de boucle

Exécution des codes

Après les premiers tests effectués, nous avons remarqué que l’exécution de l’algorithme itératif prend bien plus de temps que les autres algorithmes, mais malgré cela, nous avons essayé d’augmenter la liste sur laquelle nous effectuons nos tests, cela a pris plus de temps, du coup nous testons sur une liste plus longue et qui nous permet aussi l’exécution des trois fichiers dans un délai correcte.

Tests approfondis pour le recueil des données

Nous avons implémenté un fichier qui nous servira pour faire des test en prenant en compte le temps d'exécution ainsi que l’espace mémoire utilisé par chaque programme lors de son exécution, le fichier se trouve dans le répertoire src avec les programmes python, il s’intitule “launch_program.sh”, il suffit d’éxecuter la commande suivante sur le terminal pour le lancer :

“bash launch_program.sh”

Le script réalise plusieurs exécution sur chacun des codes python, chacun d’eux est lancé 10 fois afin d’avoir les données de temps d'exécution et de mémoire lors de plusieurs exécutions.

Les tests sont réalisés sur la même liste pour chacun des programmes, pour chaque programme on crée la liste avant de commencer la recherche de la plus longue sous-séquence .

Dans le cas où l’on a plusieurs sous séquences de la même longueur (longueur maximale) avec des éléments différents, il se peut que l’algorithme itératif retourne une liste avec quelques éléments différents des autres algorithmes,

mais qui fait la même longueur qui est la longueur maximale d'une sous séquence strictement croissante.

Les données d'exécutions recueillies sont transférées dans des fichiers .dat qui se trouve dans le répertoire /data, ensuite c'est le code "src/transferData.py" qui s'occupe de l'organisation des données dont il organise les données de façon à avoir un fichier pour le temps d'exécution pour chacun des codes, les fichiers contiennent des données de la forme suivante :

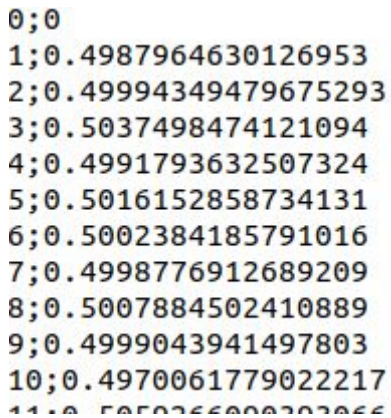

Figure 4 : format des fichiers de données pour le temps d'exécution

La figure 4 représente un fichier de données concernant le temps d'exécution pour un des codes Python, à gauche le numéro d'exécution, le numéro d'exécution dépend de la valeur d'arrêt que l'on met dans le fichier bash, donc le fichier est composé d'autant de lignes que de nombre d'exécutions mis dans la boucle du fichier bash (n exécutions = n lignes dans le fichier de données).

On s'est basé sur la même structure pour l'espace mémoire utilisé par chacun des codes, c'est-à-dire on transmet les données dans des fichiers dont chacun représente les données d'espace mémoire pour un des codes python, l'espace représenté en Mb, tout comme avec le fichier des données du temps d'exécution, le fichier des données de l'espace mémoire utilisée dépend du nombre d'exécution à réaliser (n exécutions = n lignes dans le fichier de données)

```
0;0
1;15.7734375
2;16.03125
3;15.69921875
4;15.7265625
5;15.6875
6;15.7265625
7;15.69921875
8;15.65234375
9;15.60546875
10;15.60546875
```

Figure 5 : format des fichiers de données pour l'espace mémoire en Mb

Nous avons essayé d'effectuer les tests avec des listes contenant des valeurs dans différents ordres pour compliquer la tâche aux algorithmes, cela nous permettent de bien voir les différences.

Comparaisons temps et mémoire des algorithmes

Les codes donnent le même résultat, mais ne fonctionnent pas de la même manière pour la simple raison qu'elle n'ont pas le même algorithme, ceci dit qu'il y a des différences au niveau du temps ainsi que la mémoire utilisée lorsque l'on lance le programme. Dans cette partie, nous avons développé un code "gnuplot" qui nous permet de voir les différentes données représentées en graphe afin de bien voir la différence.

Les données sont représentés sous format de graphes, ces graphes sont stockées dans l'image "plot.png", il suffit de lancer la commande suivante :

gnuplot "plot.gp" > plot.png

Ci-dessous un exemple des graphes générés :

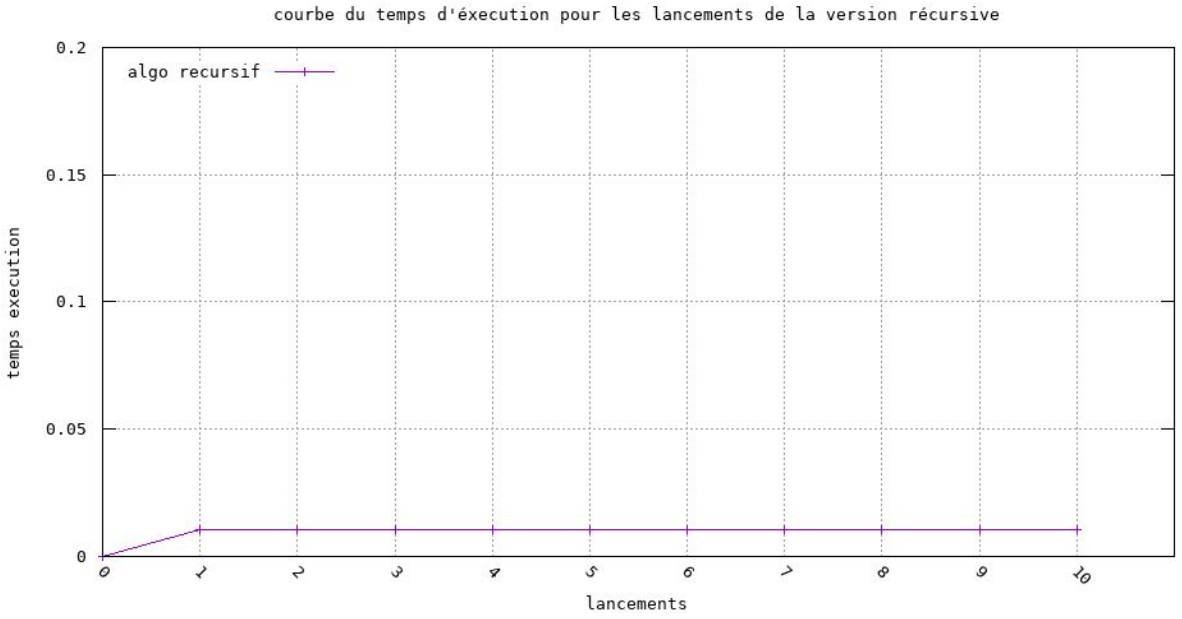

Figure 6 : courbe du temps d'exécution à chaque lancement du programme pour la version récursive

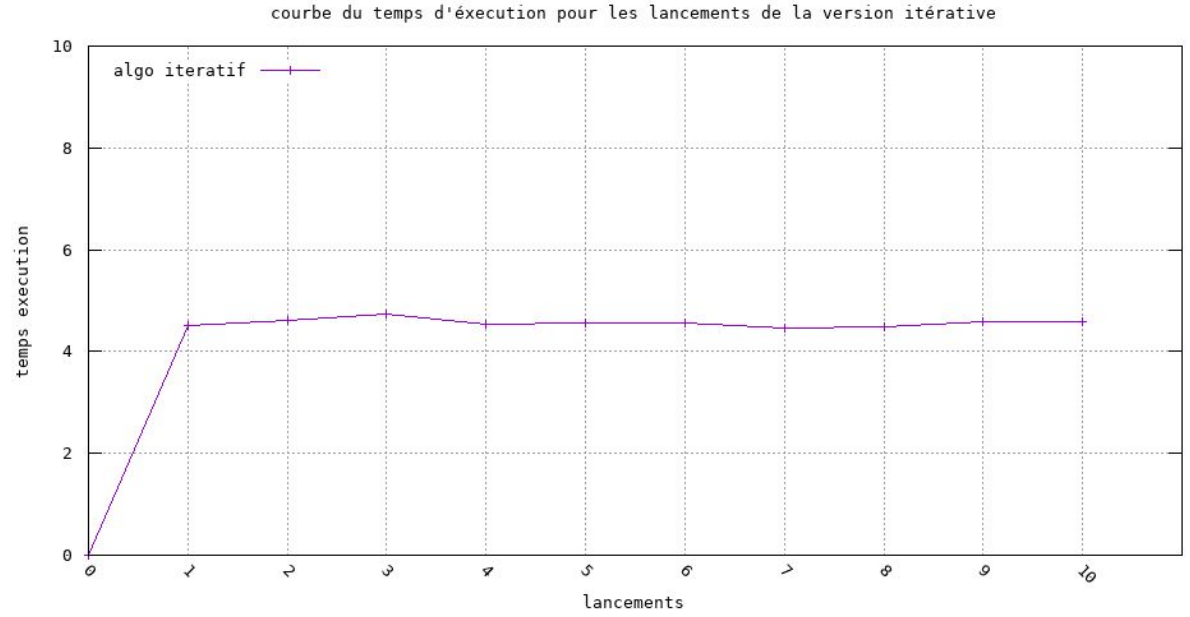

Figure 6 : courbe du temps d'exécution à chaque lancement du programme pour la version récursive

Nous avons essayé de mettre les deux courbes dans le même graphe, mais vu la différence de temps présente, j'ai décidé de les mettre chacune dans un graphe avec des valeurs différentes au niveau de l'axe des ordonnées afin de mieux voir le temps pris pour chaque exécution.

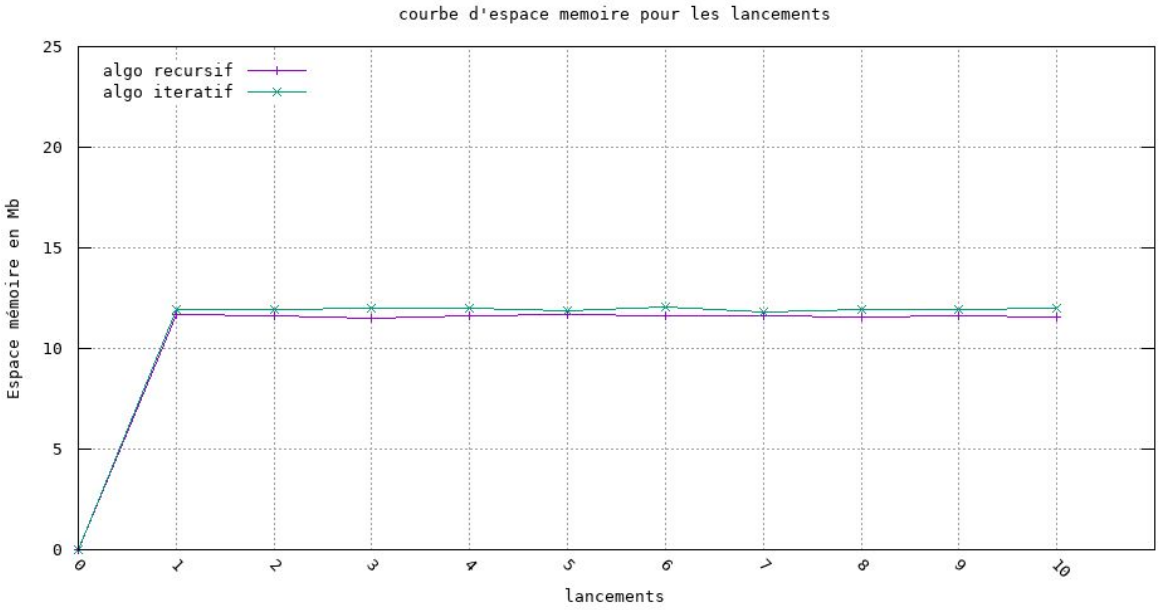

Figure 7 : courbe de l'espace mémoire utilisé à chaque exécution du programme pour la version itérative et récursive

Validation de la spécification

Afin de s'assurer que les résultats des codes respectent la spécification, nous avons développé le code "src/test.py" qui nous permet de voir si cela est vraiment respecté dans les résultats des algorithmes, il contient la fonction verifier_croissance(L) prenant en paramètre la sous-séquence finale, et vérifie si cette dernière est vraiment croissante.

Cette vérification se fait en s'assurant qu'à la fin de chaque exécution, le résultat de la plus longue séquence est bien croissant, si c'est le cas on écrit "1" dans les fichiers "resultat_test/resultat_test_algo1.dat", "resultat_test/resultat_test_algo2.dat", "resultat_test/resultat_test_algo3.dat", si ce n'est pas le cas nous écrivons "0".

A chaque lancement, nous vérifions le contenu des fichiers de résultat, le contenu prend la forme suivante :

```
1
1
1
1
1
1
1
1
1
1
1
1
```

Figure 10 : Exemple de contenu du fichier de test

Après l'analyse des fichiers à chaque lancement, nous remarquons que les fichiers contiennent que la valeur "1" à chaque ligne (une ligne représente le résultat d'une exécution 1 si le résultat respecte la spécification, 0 sinon), donc les résultats valident la spécification de l'algorithme.

Complexité

Algorithme récursif

```
def plus_longue_sequence_recursif(liste):
        max_sequence = []
        for i in range(len(liste)):
                sequence = plus_longue_sequence_a_partir_du_premier_element(liste[i:len(liste)])
                if(len(sequence) > len(max_sequence)):
                        max_sequence = sequence
        return max_sequence

def plus_longue_sequence_a_partir_du_premier_element(liste):
        result = [liste[0]]
        for i in range(1, len(liste)):
                if(liste[i] > liste[0]):
                        sequence = [liste[0]] + plus_longue_sequence_a_partir_du_premier_element(liste[i: len(liste)])
                        if(len(sequence) > len(result)):
                                result = sequence
        return result
```

Figure 10 : code récursif

Supposons que n est la taille de notre liste, le programme réalise un appel de la fonction "plus_longue_sequence_a_partir_du_premier_element(liste)" n fois parce que ce dernier parcourt la liste.

la fonction “plus_longue_sequence_a_partir_du_premier_element(liste)” calcule la plus longue séquence à partir du premier élément de la liste passée en paramètre, il le fait en parcourant la liste, donc il fait |liste|(taille de liste) itérations, l’algorithme contient aussi des conditionnelles.

La complexité de cet algorithme est donc de $O(n^2)$

Algorithme itératif

L’algorithme fonctionne de la manière suivante :

- Calcule les sous-séquence en commençant par le premier élément
- il remplit au fur et à mesure les sous-séquences qui valident ce que l’on cherche
- Il continue à remplir la sous-séquence tant qu’il trouve d’autres éléments à ajouter
- à chaque ajout, il compare la longueur de la sous-séquence avec la plus longue sous-séquence connu, si c’est le cas, cette dernière devient la plus longue sous-séquence

En résumé, cet algorithme calcule toutes les sous-séquences tout en comparant avec la plus longue sous-séquence qu’il mémorise

```
def plus_longue_sequence_iteratif(liste):
        elements_to_process = []
        for i in range(len(liste)):
            elements_to_process.append([ListItem(liste[i], i)])
        longest_sequence = []
        while elements_to_process:
            element = elements_to_process.pop()
            for i in range(element[len(element) - 1].position + 1, len(liste)):
                for j in range(i, len(liste)):
                    if liste[j] > element[len(element) - 1].item:
                        elements_to_process.append(element + [ListItem(liste[j], j)])
                        if len(longest_sequence) < len(element) + 1:
                            longest_sequence = element + [ListItem(liste[j], j)]

        return list(map(lambda list_item: list_item.item, longest_sequence))
```

Figure 11: code itératif

Finalement, la complexité de cet algorithme est de : $O(n^3)$

Algorithme 3

```
def plus_grande_sequence_rd(E):
    P = [-1 for m in E]
    M = [-1 for n in E]
    L = 0
    for i in range(0, len(E)):
        lo = 1
        hi = L
        while lo <= hi:
            mid = (lo + hi) // 2
            if E[M[mid]] < E[i]:
                lo = mid + 1
            else:
                hi = mid - 1
        newL = lo
        P[i] = M[newL - 1]

        if newL > L:
            M[newL] = i
            L = newL
        elif E[i] < E[M[newL]]:
            M[newL] = i
    S = [-1 for i in range(L)]
    k = M[L]
    for i in range(L-1, -1, -1) :
        S[i] = k
        k = P[k]
    return S
```

Figure 12 : code avec la recherche dichotomique (algorithme de wikipedia)

Le troisième algorithme a été implémenté afin de faire plus de comparaisons au niveau de temps d'exécution et de l'espace mémoire.

Le troisième algorithme réalise ses itérations récursivement en fonction de la recherche dichotomique. Notons que n est la taille de la liste, la complexité au pire des cas de la recherche dichotomique est de $O(log_2(n))$.
L'algorithme de la recherche dichotomique est utilisé dans une boucle for qui va de 1 à n (taille de la liste), cela dit que l'on utilise la recherche dichotomique à chaque itération.
Finalement, la complexité de cet algorithme est de $O(n) * O(log_2(n))$, et donc : $O(n*log_2(n))$.

Conclusion

Sur le plan technique, je considère que ce travail a été une expérience enrichissante dans la mesure où elle m'a permis premièrement de mettre en pratique nos connaissances de programmation informatique en Python et de les améliorer grâce aux tâches qui nous ont été confiées.

Ce projet m'a aussi permis de mettre à l'épreuve mon esprit critique et mes capacités d'analyse et de décision.

Le projet m'a permis de travailler sur des tâches qui ont été nouvelles pour moi comme la comparaison d'efficacité au niveau du temps et de la mémoire de différents algorithmes Python et mener une étude de complexité dessus.

Les tests ont été effectués avec des valeurs qui permettent de lancer le programme sur une grande quantité de données afin de bien voir la différence, nous avons essayé de pousser au maximum les trois programmes tout en gardant un temps d'exécution qui n'est pas trop long.

Les données recueillies sur le temps d'exécution ainsi que l'utilisation mémoire des programmes sont mises à jour à chaque fois que l'on relance le script bash, il faut donc remettre à jour les graphes avec la commande citée précédemment.

Ressources

https://www.javaer101.com/article/3312912.html

https://www.ipgirl.com/8152/comment-determiner-la-sous-sequence-croissante-la-plus-longue-en-utilisant-la-programmation-dynamic.html

https://fr.wikipedia.org/wiki/Plus_longue_sous-suite_strictement_croissante

https://leetcode.com/problems/longest-increasing-subsequence/

https://en.wikipedia.org/wiki/Longest_increasing_subsequence

https://www.baeldung.com/cs/longest-increasing-subsequence-dynamic-programming

https://stackoverflow.com/questions/20588021/how-does-algorithm-for-longest-increasing-subsequence-onlogn-work