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**](#_zd63nnte8jps) **3**

[**Description du projet et objectif**](#_d27m3fqk10kq) **3**

[**Environnement de travail**](#_uih949qsqwq7) **4**

[**Implémentation**](#_tih76c4amadf) **4**

[présentation des algorithmes](#_cshk2ts6pf87) 4

[Premiers Tests](#_jfli4gv6hkzr) 6

[Spécification](#_l7fu5m3q84kd) 7

[**Exécution des codes**](#_fk9akepb2v5g) **8**

[Tests approfondis](#_q8ejbjt3jxd3) 8

[**Comparaisons temps et mémoire des algorithmes**](#_vv8m2t8j9c8s) **9**

[**Validation de la spécification**](#_encg1gy2z28s) **13**

[**Complexité**](#_scck2v6nsymh) **14**

# 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.

# Environnement de travail

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

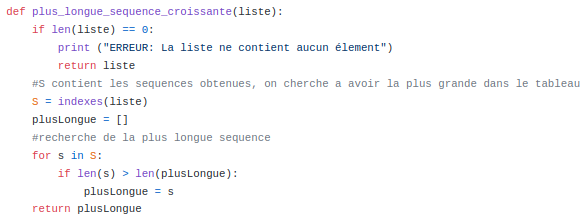
Le projet a été développé sous Python 3.8.5.

## 

# Implémentation

## présentation des algorithmes

Pour notre problème, nous avons implémenté trois algorithmes sous Python, l’un différent de l’autre, le premier algorithme est récursif, le deuxième est itératif, et le troisième est différent des deux premiers mais résout le même problème aussi, ce dernier utilise la recherche dichotomique.

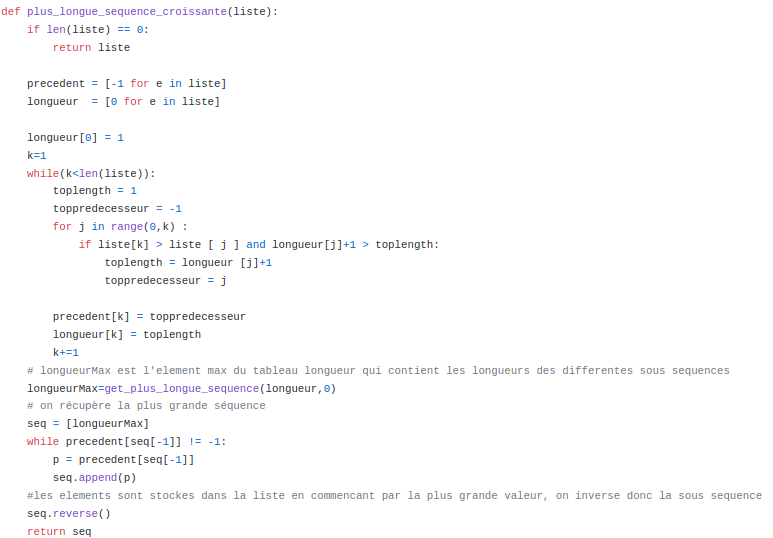


**Figure 1 :** premier algorithme résolvant le problème de la plus longue sous séquence dans une liste

La figure 1 nous montre le premier algorithme, cet algorithme fait appel à une fonction indexes qui retourne les différentes séquences obtenues dans un tableau, chacune dans une case, par exemple :

[[1,2,7,9,10],[3,5],[7,9,10]]

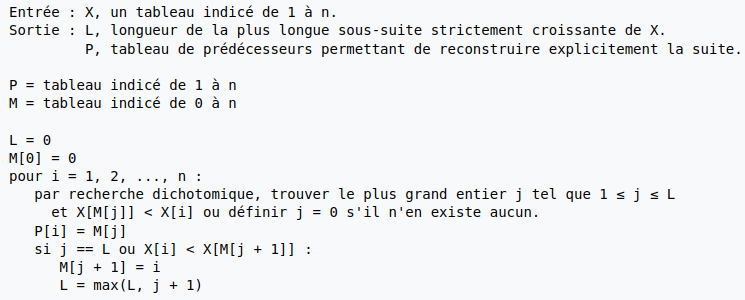
Dans notre cas nous nous intéressons à la séquence la plus longue, donc c’est la première dans notre exemple, et c’est ce que fait la boucle dans la fonction, elle parcourt le tableau afin d’avoir la plus longue séquence, comme la recherche du maximum dans une liste d’entiers.



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

La figure 2 représente le deuxième algorithme, ce dernier n’utilise pas les appels récursifs, afin d’améliorer l’algorithme, ce dernier conserve à chaque itération la meilleure sous-séquence se terminant à la position k au lieu de retenir toutes les séquences comme le premier algorithme.

Le troisième algorithme implémenté est celui proposé par Wikipédia.



**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.

## 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.

∀ i∈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.

Le premier algorithme comme nous l’avons cité stocke dans un tableau toutes sous séquences croissantes qu’il peut trouver pendant chaque itération *i,* ensuite ce dernier garde celle avec la plus grande longueur.

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, cela est le cas d’une liste d’entrée qui est 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.

# 

# Exécution des codes

Après les premiers petits tests effectués sur des listes de taille entre 10 et 100, nous avons essayé manuellement avec des listes un peu plus grandes (taille entre 100 et 500), cela a pris un peu plus de temps et a consommé plus de mémoire, pour nous faciliter la tâche et réaliser des tests plus performant et sûrs, j’ai implémenté un fichier bash qui nous servira pour les tests.

## Tests approfondis

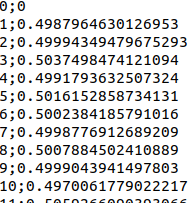
Nous avons implémenté un fichier qui nous servira pour faire des test en prenant en compte le temps d'exécution ainsi que l’espace mémoire utilisé par chaque programme lors de son exécution, le fichier se trouve dans le répertoire src avec les programmes python, il s’intitule “launch\_program.sh”, il suffit de 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 50 fois afin d’avoir les données de temps d'exécution et de mémoire lors de plusieurs exécutions.

Les tests sont réalisés sur des listes de taille 3000 pour chacun des programmes, pour chaque programme on crée la liste avant de commencer la recherche de la plus longue sous-séquence dedans, les tests sur des listes d’une telle taille prennent plus de temps que les tests que l’on a fait précédemment avec des listes moins grandes , de plus le script lance le traitement de chacun des codes sur une liste de taille 3000 50 fois.

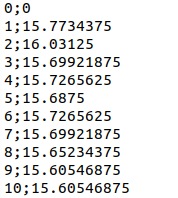
Les données d'exécutions sont transférées dans des fichiers .dat qui se trouve dans le répertoire /data, ensuite c’est le code “src/transferData.py” qui s’occupe de l’organisation des données dont il organise les données de façon à avoir un fichier pour le temps d'exécution pour chacun des codes, les 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, comme nous lançons les programmes 50 fois, nous aurons des fichiers composés de 50 lignes, et à droite le temps d’exécution.

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.



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

Nous avons essayé d’effectuer les tests avec des grandes valeurs qui nous permettent de voir les différences sans que ce soit trop long lors de l’exécution du fichier bash.

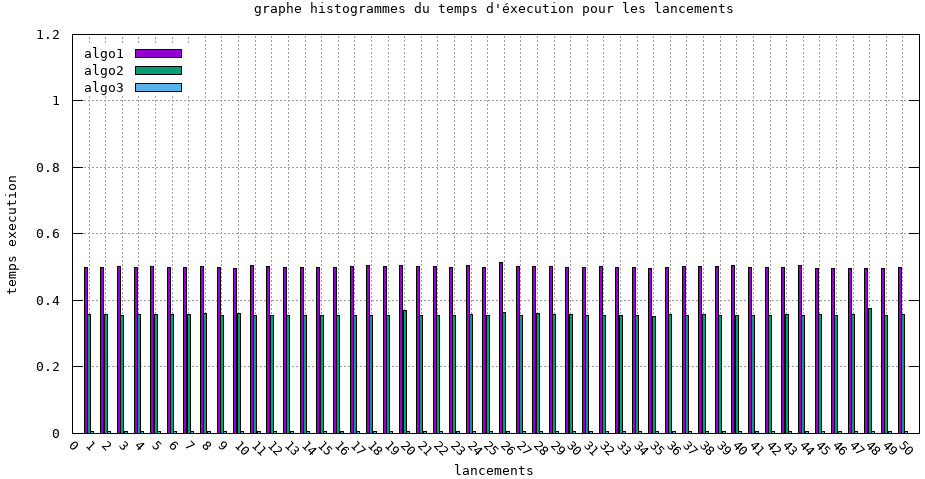
# Comparaisons temps et mémoire des algorithmes

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.

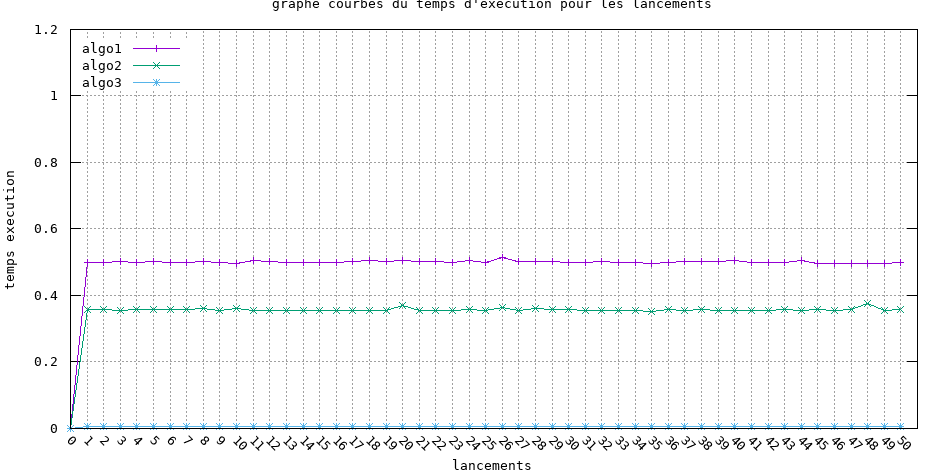
Les données sont représentés sous format de graphes, ces graphes sont stockées dans l’image “plot.png”, il suffit de lancer la commande suivante :

gnuplot “plot.gp” > plot.png

L’image dispose de deux graphes représentants le temps d'exécution, un sous forme d’histogrammes, et un autre sous forme de courbe, et de même pour l’espace mémoire utilisée.



**Figure 6 :** Graphes histogrammes pour le temps d'exécution pour chacun des lancements du programme (50 exécution)

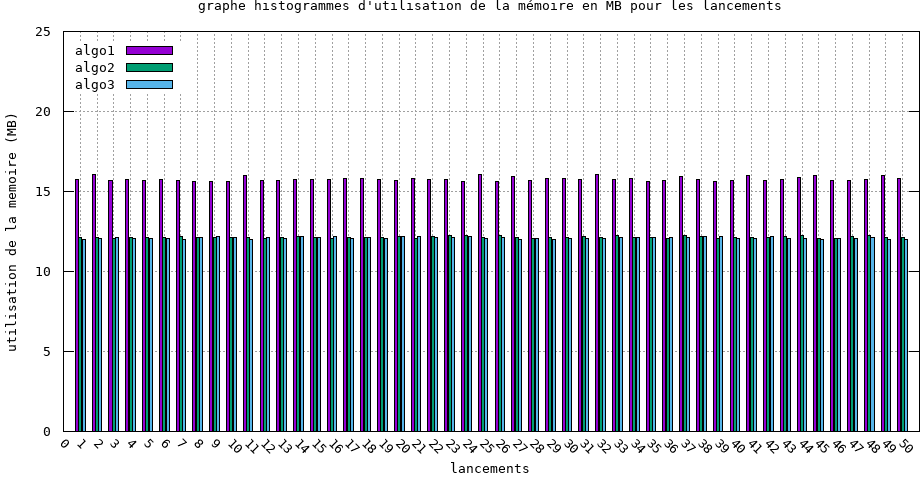


**Figure 7 :** Courbes représentant le temps d'exécution pour chacun des lancements du programme (50 exécution)

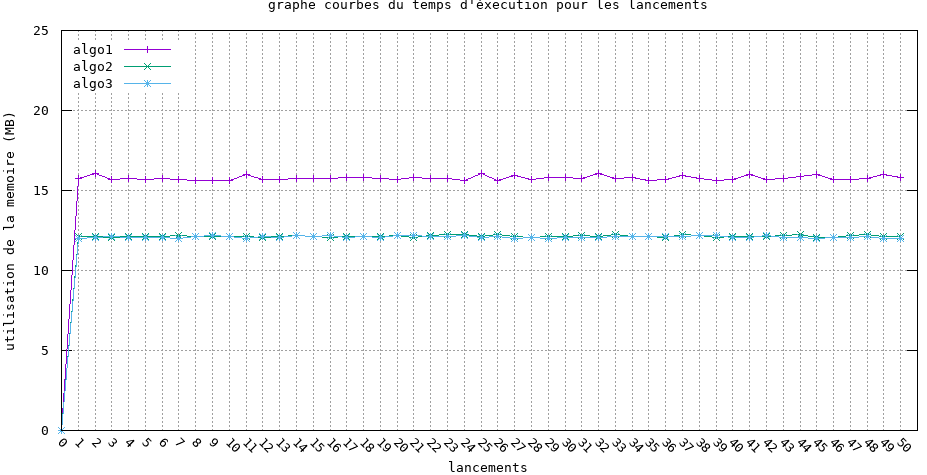
Comme nous pouvons le constater les deux graphes représentent les mêmes données, nous pouvons remarquer que l’algorithme 3 qui utilise des appels récursifs de la dichotomie est bien plus efficace que les autres au niveau du temps d'exécution.

L’algorithme 1 prend plus de temps que l’algorithme 2 parce qu'il garde l’historique des plus longues séquences, ce qui fait un peu plus de traitements à chaque itération par rapport à l’algorithme 2.

Les résultats au niveau du temps d'exécution représentent bien les programmes, parce qu'on le remarque que l’algorithme qui fait le plus de tâches est celui qui prend le plus de temps.



**Figure 8 :** Graphes histogrammes pour l’espace mémoire de chacun des lancements du programme (50 exécution)



**Figure 9 :** Courbes représentant l’espace mémoire utilisé pour chacun des lancements du programme (50 exécution)

Concernant l’espace mémoire, nous remarquons que l’algorithme 3 et l’algorithme 2 prennent presque le même espace à chaque, il y a une différence minime, tandis que le premier utilise plus d’espace parce qu’il a plus de données à traiter avant de fournir le résultat final.

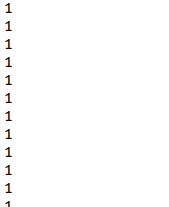
# 

# 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 permettra de voir si cela est vraiment vérifié, 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 :



**Figure 10 :** Exemple de contenu du fichier de test

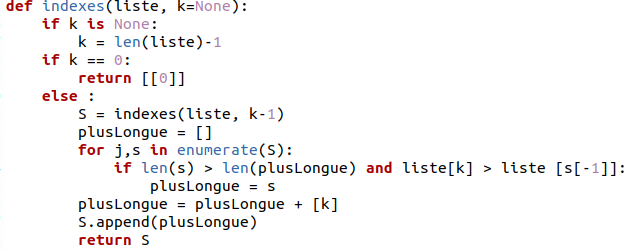
Après l’analyse des fichiers à chaque lancement, nous remarquons que les fichiers contiennent que la valeur “1” à chaque ligne (une ligne représente le résultat d’une exécution 1 si le résultat respecte la spécification, 0 sinon),

donc les résultats valident la spécification de l’algorithme.

# Complexité

## Algorithme 1

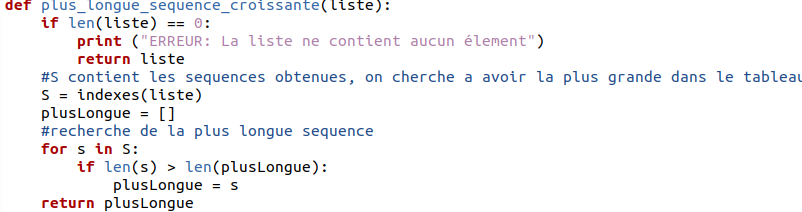
Concernant le premier algorithme, comme nous le constatons dans la figure 1, il fait appel à la fonction indexes



**Figure 10 :** fonction indexes(liste, k)

La fonction indexes dépend de la valeur de k, dans notre fonction principal nous appelons indexes(liste, k) sans paramètre k, donc ce dernier est défini par défaut à l’indice du dernier élément, et l’on fait des appels récursifs en décrémentant la valeur de k par 1 à chaque fois, donc, supposant que la taille la liste est *n,*

On fait appel à la fonction indexes *n* fois, et donc cela représente une complexité de *O(n²)* (le if est constant)



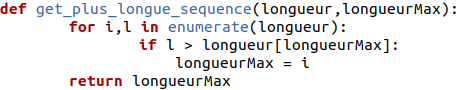
**Figure 11 :** fonction principale

Tandis que dans la fonction principal du calcul de la plus longue sous-séquence nous mettons le résultat de indexes(listes, k) dans la variable S que l’on parcourt par une boucle for, avec un if à l’intérieur donc cela fait : *O(n²)* + *O(|S|)*  \* *O(1)*

Donc la complexité de l’algorithme est de : *O(n²)*

## Algorithme 2

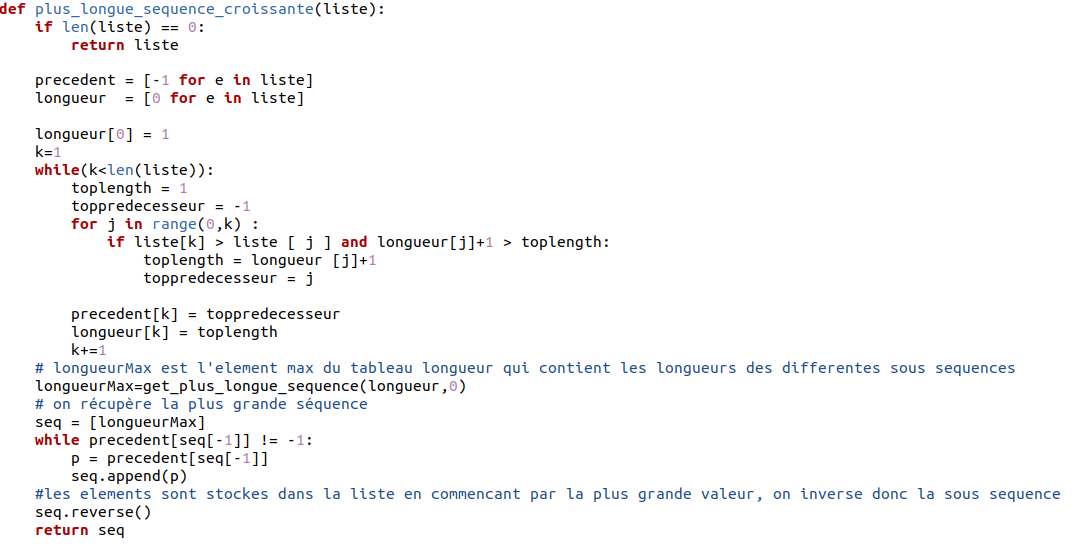
Pour le second algorithme, on fait appel à la fonction get\_plus\_longue\_sequence()



**Figure 12 :** fonction get\_plus\_longue\_sequence

Dans la fonction principale, on utilise :

* Une initialisation de deux tableaux chacun de la même taille que notre liste d’entrée avec des boucles for, donc, n opérations dont n est la taille de la liste.
* on parcourt la liste avec une boucle while qui va de 1 jusqu’à |liste| (longueur de la liste), avec une boucle for imbriquée allant de de 0 jusqu’à k.
* un appel à la fonction get\_plus\_longue\_sequence(longueur,0), qui dépend du paramètre longueur, parce qu’elle utilise une boucle parcourant la liste longueur.
* une boucle while pour stocker les éléments de la plus grande sous-séquence dans une liste, supposons que la plus grande sous-séquence trouvée est de taille m, donc la boucle while fait m itération.

**

**Figure 12 :** fonction principale de l’algorithme 2

Finalement, la complexité de cet algorithme est de : *O(n²)*

## Algorithme 3

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(log2(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(log2(n)),* et donc *: O(n\*log2(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 recueillis 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.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://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://fr.wikipedia.org/wiki/Plus_longue_sous-suite_strictement_croissante)

[**https://leetcode.com/problems/longest-increasing-subsequence/**](https://leetcode.com/problems/longest-increasing-subsequence/)

[**https://www.baeldung.com/cs/longest-increasing-subsequence-dynamic-programming**](https://www.baeldung.com/cs/longest-increasing-subsequence-dynamic-programming)

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