SAÉ S1.02

Comparaison d'approches algorithmiques

Berne Léa, Malosse Audran

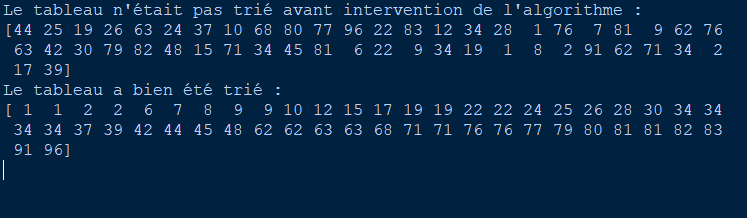
# Vérification et description

## Algo X

### Analyse théorique

L’algorithme X respecte la post-condition (trie un tableau par ordre croissant). La variable i est initialisé à 1. On va donc parcourir le tableau à partir de la seconde valeur de celui-ci. On affecte à la variable j la valeur de i-1, soit l’indice dans le tableau précédent celui de i. Dans la première boucle, on affecte la valeur à l’indice i dans une variable tmp. Puis on affecte la valeur du tableau à l’indice j à tab[j+1] tant que la valeur à indice j est supérieur à la variable tmp (précédemment affecté à la valeur d’indice i) et que j est supérieur ou égale à 0. Et on décrémenter j de 1. Cela a pour effet de faire descendre la valeur à trier dans le tableau pour qu’elle se retrouve à sa place. En sortie de boucle, il faut donc replacer la valeur stockée dans tmp à sa nouvelle place qui est l’indice j+1. On incrémente i et on recommence. Il s’agit d’un algorithme de type tri insertion.

### Analyse pratique

Grâce à l’algoritme automatic\_testing que nous avons créé, nous pouvons vous montrer ce qu’il se passe quand on trie un tableau de 50 entiers compris entre 1 et 100 à l’aide de la fonction test\_tri \_detail :

Nous avons automatisé la création de tableau de valeurs aléatoires et la vérification dans la fonction test\_tri \_boucle : un test effectué avec 100 tableaux de 100 valeurs montre que l’algorithme de vérification retourne bien TRUE à chacunes des fois, et donc, que algo\_x effectue bien un tri par ordre croissant.

### Description de la complexité temporelle et spatiale

* Complexité Temporelle : O(n²) car une première boucle parcoure le tableau n fois, et dans cette boucle, une autre boucle parcoure le tableau n fois. La complexité temporelle de cet algorithme ne changera pas, que le tableau soit tout à fait aléatoire, trié, ou trié à l’envers.
* Complexité Spatiale :
  + En place : OUI puisqu'on ne crée pas de nouveau tableau.
  + Stable : NON car on vérifie que la valeur à classer (stockée dans tmp) est strictement inferieure à la valeur comparée. Au moment où tmp sera comparé à une valeur identique à elle-même, la boucle continuera de tourner.

## Algo Y

### Analyse théorique

L’algorithme Y respecte la post-condition (trie un tableau par ordre croissant).

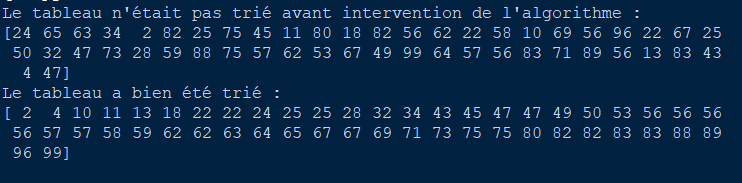
Cet algorithme est composé de trois fonctions qui s’appellent entre elles. C’est la fonction algo\_main qui prends en argument le tableau et appelle la fonction algo\_b avec, en paramètre, le tableau à trier, 0 (qui est l’indice minimum) et len(tab)-1 (qui est l’indice maximum). En vérité, algo\_main sert simplement à appeler algo\_b avec les paramètres nécessaires à un tri fusion.

Algo\_b est un algorithme récursif, et utilise une méthode « diviser pour régner », c’est-à-dire qu’il divise le tableau qu’il prend en paramètre en deux parties. Pour cela, on teste d’abord si i (indice minimum) est égal à j (indice maximum), ce qui entrainerait l’arrêt de la fonction (car un tableau qui n’aurait qu’un élément serait déjà trié). En cela, l’algorithme est similaire au tri fusion, mais contrairement au tri fusion vu en cours, nos trois algorithmes produisent un tri qui est en place : cela vient de l’appel à la fonction algo\_a, qui remplace l’algorithme de fusion que l’on a vu en cours.

L’algo\_a ne fait pas appel à la fonction zeros, et ne prends pas en paramètre le milieu du tableau comme le fait la version de la fusion vue en cours. Par contre, l’algo\_a cherche lui-même le milieu du tableau qui lui a été donné en paramètre pour effectuer un tri par insertion à gauche du milieu.

Par ailleurs, une autre différence entre algo\_b + algo\_a et tri\_fusion + fusion vu en cours est que algo\_b effectue un tri du tableau si le tableau est de deux éléments en permutant les voisins qui ne seraient pas triés. L’algorithme test si le tableau à trier n’a que deux éléments, et ce n’est que si le tableau à trier comporte plus de deux éléments qu’algo\_b fait appel à algo\_a : trier de cette manière permet de ne pas appeler algo\_a dans le cas d’un tableau à deux éléments, ce qui permet sûrement d’optimiser l’algorithme.

### Analyse pratique

 Voilà ce qu’il se passe quand on trie un tableau de 50 entiers compris entre 1 et 100 à l’aide de notre fonction test\_tri \_unique :

Un test effectué avec 100 tableaux de 100 valeurs montre que l’algorithme de vérification retourne bien TRUE à chacune des fois, et donc, que algo\_y effectue bien un tri par ordre croissant.

### Description de la complexité temporelle et spatiale

* Complexité Temporelle : O(n\*log(n)) car l’aglo\_b est récursif et que l’algo\_a va être exécuté n fois et ce, que le tableau soit tout à fait aléatoire, trié, ou trié à l’envers.
* Complexité Spatiale :
  + En place : OUI puisqu'on ne crée pas de nouveau tableau.
  + Stable : NON puisque l'algorithme déplace les éléments vers la bonne position, donc il peut modifier l'ordre des éléments à valeurs égales.

## Algo Z

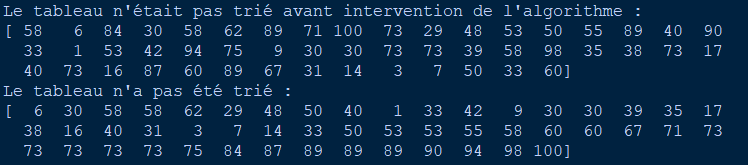
### Analyse théorique

L’algorithme Z ne respecte la post-condition (trie un tableau par ordre croissant). En réalité, l’algorithme compare deux valeurs voisines dans le tableau (celle d’indice j et celle d’indice j+1) et les inverse si la valeur de tab[j] est plus grande que celle de tab[j+1]. Ainsi, à la fin du premier tour de la première boucle while, la plus grande valeur du tableau parcouru aura remonté à droite, et donc la partie de droite sera la partie triée du tableau, ce qui explique que le deuxième while s’arrête à len(tab)-1-i.

Le problème, c’est qu’à la fin du premier tour de la deuxième boucle while, i est incrémenté, or la valeur de j dépend de celle de i. Comme j est initialisé à i, la valeur d’indice 0 ne sera comparée à son voisin de droite qu'une seule fois au court de l’exécution, celle d'indice 1 que deux fois etc. Pour généraliser, la valeur d’indice n ne sera comparée à son voisin de droite que n+1 fois, et s’il se trouve que la valeur d’indice n+1 est plus grande que lui mais que la suivante (n+2) est plus petite, alors la valeur d’indice n restera où elle est malgré tout. La partie de gauche ne sera donc pas triée.

Pour corriger cela, il faut initialiser j à 0 pour reparcourir toute la partie gauche du tableau, puisqu’elle n'est pas triée. Il s’agit en fait d’un tri à bulle.

### Analyse pratique

Voilà ce qu’il se passe quand on trie un tableau de 50 entiers compris entre 1 et 100 :

Nous voyons ainsi que l’algorithme effectue bien un tri des valeurs du tableau, et, notamment, les valeurs les plus grandes du tableau se retrouvent bien à la fin, à la place où elles devraient être. Cependant, plus on se rapproche de l’indice 0, plus des erreurs de tri se retrouvent dans le tableau, du fait du comportement décrit précédemment.

Ce problème surviendra bien chaque fois qu’un tableau non trié sera fourni à algo\_z : notre fonction test\_tri \_boucle a en effet retourné faux dès le premier tableau, et a continué à le faire jusqu’au dernier tableau.

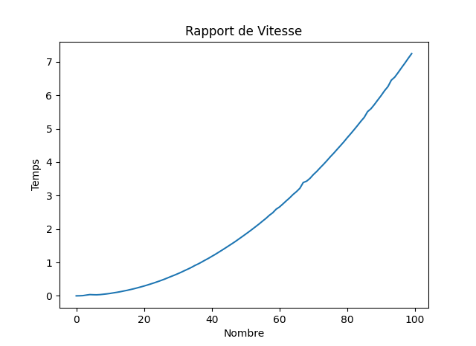
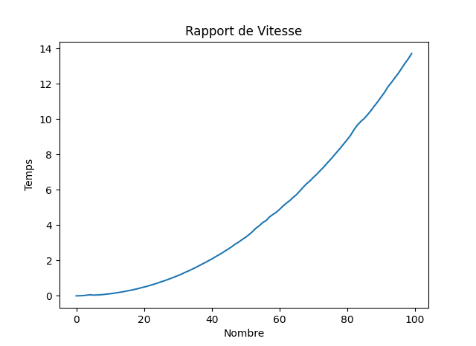
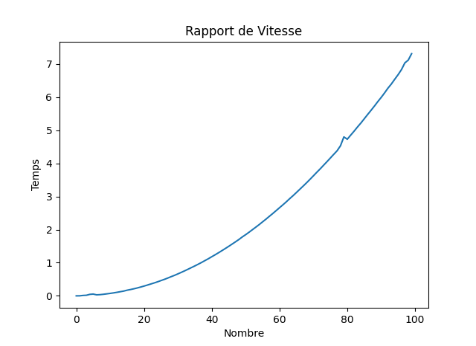
### Description de la complexité temporelle et spatiale

* Temporelle : O(n²) car la fonction va comparer n fois le tableau en entier, et ce, que le tableau soit tout à fait aléatoire, trié, ou trié à l’envers.
* Spatiale :
  + En place : OUI, puisqu'on ne crée pas de nouveau tableau.
  + Stable : NON, car en échangeant deux valeurs, on peut inverser l'ordre dans lequel ils apparaissaient.

# Comparaison

Nous avons réalisé différents tests sur les algorithmes en leur passant des tableaux de façon successive de i \* 100 valeurs avec i allant de 0 à 100.

La fonction crée donc des tableaux de plus en plus conséquents. Ceux-ci peuvent être trié de façon aléatoire, croissant ou décroissant. Dans le cas de l’aléatoire, les valeurs sont sélectionnées dans l’intervalle [-1 000 ; 1 000] pour avoir un jeu de valeur assez large pour l’analyse de trie des algorithmes.

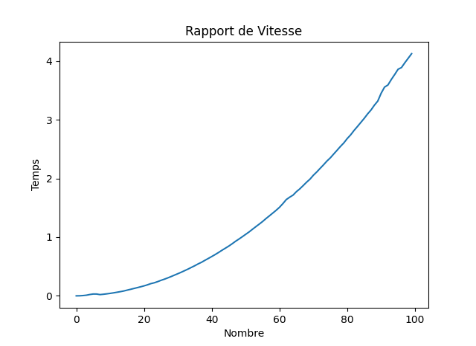


*Z random*

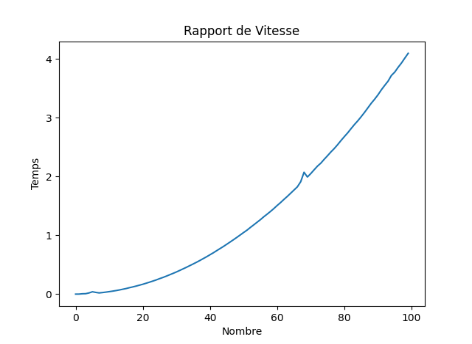
*Z inversement trié*

*Z trié*

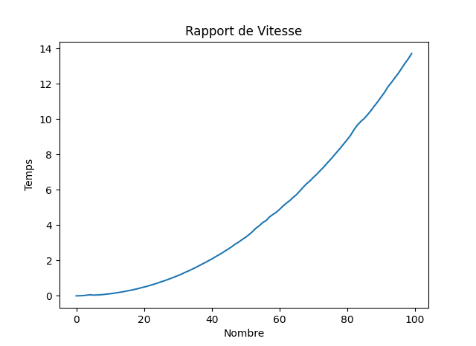
Nous pouvons constater que l’algorithme Z est le moins performant. C’est celui qui mets le plus de temps à trier un tableau, quel que soit sa taille ou le tri de départ de celui-ci. Il lui faudra 14s pour faire le tri d’un tableau aléatoire de 10 000 valeurs. Il est toutefois plus rapide si le tableau est trié dans l’ordre croissant ou décroissant. Dans ce cas, le temps d’exécution est divisé par deux. Son temps d’exécution reste linéaire.



*X inversement trié*

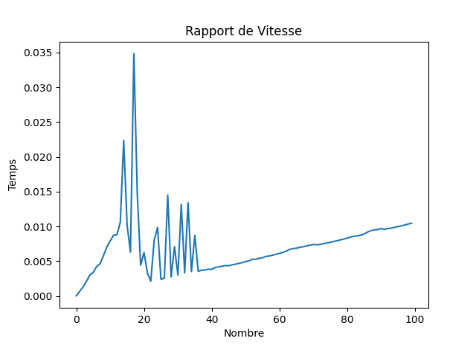


*X trié*

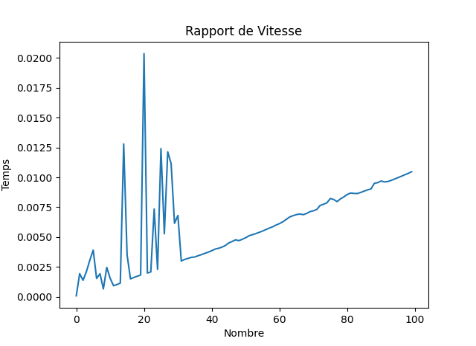


*Z random*

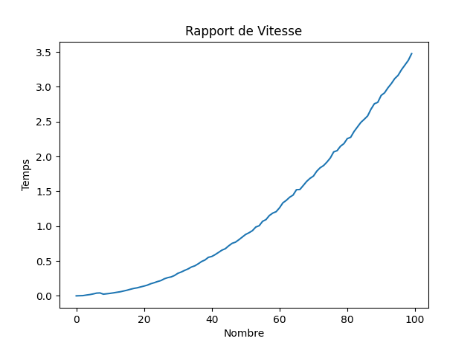
L’algorithme X est plus performant que Z. En effet, il ne lui faudra 4s pour faire le tri du dernier tableau aléatoire de 10 000 valeurs. Son comportement temporel est similaire à celui de Z.



*Y trié*



*Y inversement trié*



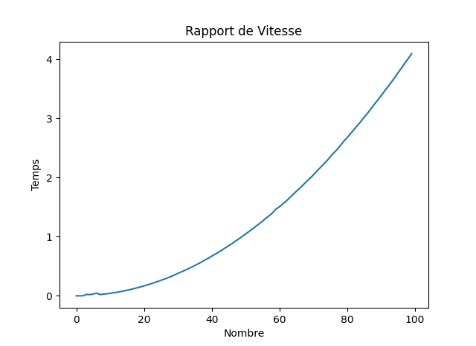
*Y random*

L’algorithme Y est beaucoup plus performant quel que soit le cas de figure. Il ne mettra environ que 3,5s pour le tri le plus extrême. Cela est rendu possible grâce à son fonctionnement récursif. Si le tableau est déjà trié (dans l’ordre croissant ou décroissant), alors l’exécution est extrêmement rapide avec une exécution de l’ordre du millième de secondes sur nos tests.

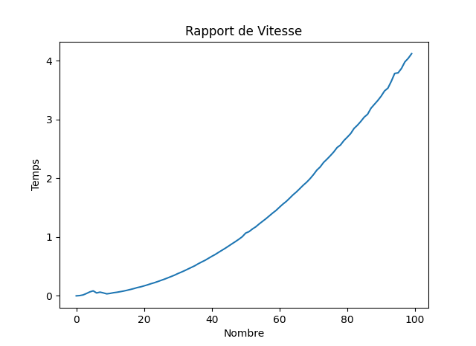
# Ecriture d’un algorithme de tri

*Vitesse d’exécution en fonction de la taille des tableaux*

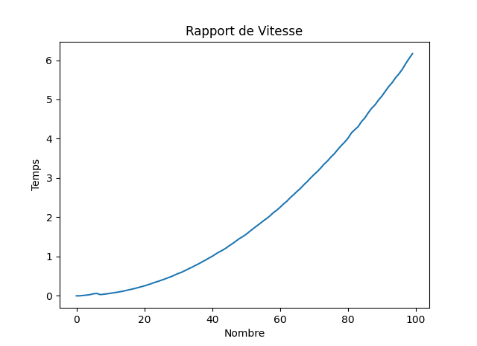
L’algorithme que nous avons imaginé est une version d’un tri selection qui utilise une fonction externe (recherche\_min) pour trouver la valeur minimum du tableau et son indice. Dans une boucle while qui parcourt le tableau, notre algorithme stocke les valeurs retournées par recherche\_min dans min et indice\_min, puis échange la valeur contenue dans le tableau à l’indice i avec celle contenue à l’indice indice\_min. Ainsi, à gauche du tableau, il y a la partie triée du tableau, et lorsque i est incrémenté, l’algorithme rappelle recherche\_min avec le tableau et i pour que recherche\_min trouve la plus petite valeur du tableau non trié (la partie de droite).



*Perso trié*



*Perso inversement trié*



*Perso random*

Par rapport aux autres, notre algorithme a l’avantage d’être à la fois en place et stable.

En termes d’exécution pratique, nous pouvons voir que cet algorithme rivalise avec l’algo\_x (tri insertion) et qu’il est plus performant que l’algo\_Z modifié (tri à bulle). Par contre, il est bien moins performant que l’algo\_Y, qui implémente un tri fusion, couplé avec un tri insertion.

# Contribution individuelle au projet

J’ai eu l’impression que notre travail avait été divisé très équitablement, puisqu’un tiers des tâches ont été effectuées directement à deux (code de checktab, random\_table, modification d’algo\_Z et algorithme de tri personnel), et que nous nous sommes partagés individuellement les deux tiers restants. J’ai fait l’analyse des algorithmes, ainsi que la description d’algo Z, algo Y, et notre algorithme personnel. J’ai aussi codé automatic\_testing. Audran a fait description algo X et le code de main, les courbes avec print\_time et time\_to sort.

J’ai trouvé que notre groupe avait très bien fonctionné parce que même quand nous avions séparé des tâches, nous nous sommes toujours expliqués ce qu’on faisait, et nous travaillions toujours en même temps, l’un à côté de l’autre, pour que nous soyons toujours au courant de ce que faisait l’autre. C’était très appréciable parce que nous avancions très efficacement.