TP2

Table des matières

Index des figures

Index des tableaux

1. Produit de matrices

Mon approche pour cet algorithme a été la suivante :

Pour chaque coefficients de la matrice résultante, multiplier 1 a 1 les coefficients sur la ligne de A correspondante avec ceux de la colonne de B correspondante.

Pour connaître le nombre d'opérations élémentaires, j'ai initialisé a 0 une variable au début de la fonction. Je l'incrémentais de 1 a chaque multiplication

(algorithme productMatrix(A,B) dans le fichier matrix.py)

2. Algorithmes de recherche

2.1. Recherche simple

Pour savoir si un élément existe dans une liste, il est possible de comparer les éléments de la liste avec celui que l'on cherche de manière séquentielle.

(algorithme searchSeq(word, table) dans le fichier search.py)

2.2. Recherche dans un dictionnaire

Pour rechercher un élément dans une liste triée, l'utilisation de la dichotomie est plus efficace qu'une recherche séquentielle :

Comparer l'élément du milieu de la liste avec celui recherché.

Si il correspond, renvoyer vrai.

Sinon,

si l'élément est plus grand que l'élément recherché, retirer la partie inférieure de la liste et répéter.

sinon, si l'élément est plus grand que l'élément recherché, retirer la partie supérieure de la liste et répéter.

Cet algorithme se prête bien a la récursivité.

(algorithme searchDicoRecursive(word, l=d) dans le fichier search.py)

2.3. Recherche de chaîne de caractère dans une autre

J'ai utilise un recherche séquentielle pour cet algorithme. Celui ci compare la 1ere lettre de la chaîne contenante avec la chaîne contenue. Si elle correspond, l'algorithme compare la lettre suivante de la chaîne contenue avec celle du contenant. Sinon, il compare la lettre suivante du contenant avec la 1ere lettre du contenu. Et ainsi de suite jusqu'à ce que la chaîne soit trouvée ou que toute la chaîne contenante ai été check.

Afin de l'utiliser plus tard, la fonction renvoie aussi le 1er index de la position de la chaîne cherchée.

(algorithme searchStrInStr(container, contained) dans le fichier search.py)

2.4. Remplacement de chaîne de caractères par une autre

Cet algorithme utilise celui précédemment défini.

Il cherche la présence de la chaîne a remplacer dans celle contenante. Si la chaîne a remplacer n'existe pas ou qu'elle est la même que la chaîne remplaçante, la chaîne est renvoyée sans modification.

Sinon la chaîne contenante est coupée autour de la chaîne a remplacer, et la chaîne remplaçante y est insérée.

Dans mon implémentation, la variable howMany correspond au nombre de fois que l'on veut remplacer la chaîne, dans le cas ou celle ci apparaît plusieurs fois. Elle est initialisée au maximum possible par le système.

(algorithme replaceStr(container, replaced,replaceing,howMany=sys.maxsize) dans le fichier search.py)

3. Jeu de la pyramide

Pour ce jeu, j'ai utilisé des fonctions définies dans la partie précédente.

La fonction prend en paramètre le mot précédent, demande a l'utilisateur un nouveau mot.

Pour chaque lettres de l'ancien mot, l’algorithme efface la lettre correspondante avec la fonction replaceStr() (qui utilise une recherche séquentielle). Si la chaîne résultante a une longueur de 1, et que le mot appartient au Lexique Gutenberg avec ou sans accent (cette partie est vérifiée avec searchDicoRecursive(), qui utilise la dichotomie), la fonction s'apelle avec comme argument le nouveau mot. Sinon elle renvoie "game over".

(algorithme pyraimid(oldWord) dans le fichier pyramid.py)

4. Algorithmes de tri

Pour créer un jeu de carte aléatoire, j'ai utilisé la fonction shuffle() de la bibliothèque random.

Pour la complexité empirique, je trace le graphique du temps d'exécution pour les 3 algorithmes, pour des tailles jeu entre 0 et 499 (au delà, la limite de récursivité de python est atteinte).

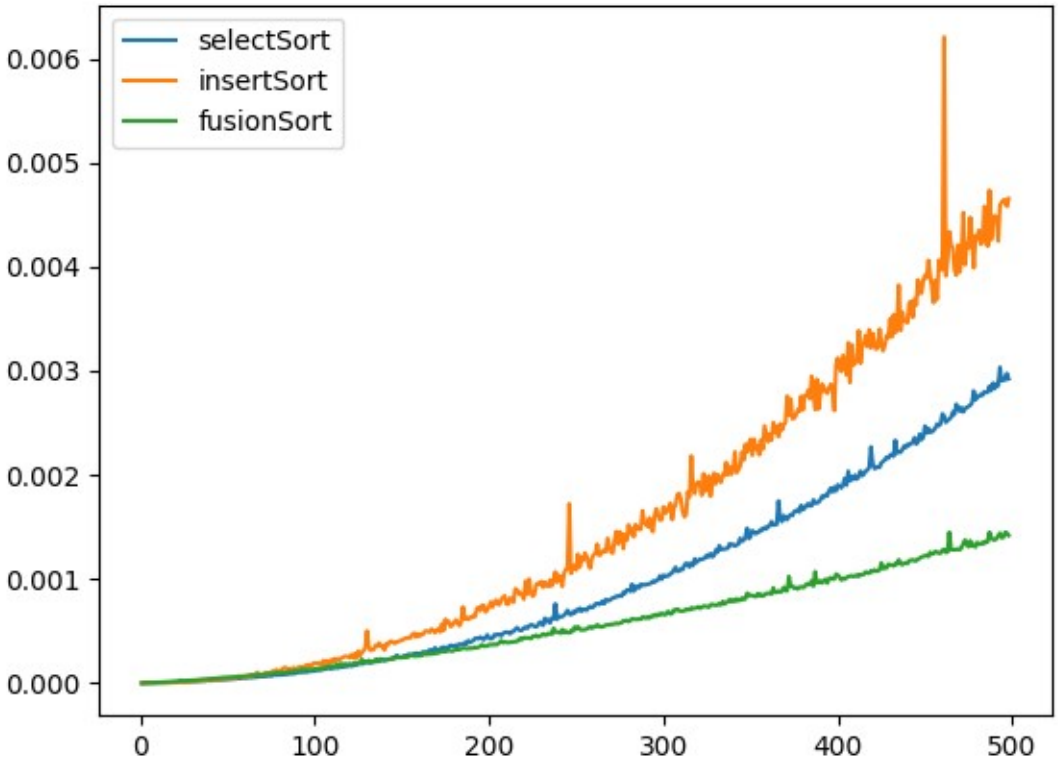

Figure 1: Graphique du temps d'execution en secondes en fonction de la taille du jeu, pour les 3 algorithmes de tri

(algorithmes dans le fichier sort.py)