ENSA
KHOURIBGA

Projet "

La détection de doublons

Réalisé par :

- El ADNAOUI YOUSSEF
- HABIBI MOHAMED

Encadré par : Mr. IMAD HAFIDI

2020/2021

Remerciements

Nous tenons tout d'abord à remercier notre encadrant
MR. IMAD HAFIDI

pour ses efforts et son aide ainsi que pour nous avoir donner l'opportunité de pratiquer les outils acquis et de développer nos connaissances en matière informatique.

Nous tenons à remercier également tous ceux qui nous ont aidés dans ce travail, et qui ont contribué de près ou de loin à l'aboutissement de ce projet.

Introduction :

Le présent rapport présente le travail réalisé dans le cadre du projet de programmation avancée, visant à réaliser un simulateur de suppression de doublons en utilisant différentes structures de données vues en cours.

En effet, le problème de détection de doublons est considéré comme le plus fondamental en matière d'algorithme par la plupart des théoriciens de l'informatique, surtout que la détection de doublons est l'une des principales opérations exécutées sur les ordinateurs. C'est dans ce
contexte que notre projet a pris pour but de réaliser un simulateur qui considère un ensemble de mots générés dans un fichier, qui seront éliminés en utilisant la distance :Levenshtein.

Dans ce rapport, nous allons présenter les différentes étapes qu'on a suivies pour aboutir à l'application finale. Il est donc axé sur deux grandes parties : la première partie est la partie d'analyse et de conception. Dans cette partie, nous allons parler des structures de données et leurs caractéristiques, les quatre distances utilisées, et nous allons représenter des simulations des résultats obtenus ainsi que la conception de l'application.

La deuxième partie s'articule sur la réalisation de l'application. Nous y présentons les différentes étapes
de la programmation ainsi que des captures d'écran de la solution finale

Chapitre 1 : Structures de données

Les tableaux dynamiques :

1. Définition :

Un tableau en informatique théorique est une structure de données séquentielle, pour des données de même type. Ces données sont contenues en mémoire de façon contigüe. Il existe deux types de tableaux : les tableaux statiques, dont la taille est connue à la compilation, et les tableaux dynamiques, dont la taille est connue à l'exécution. Nous nous intéresserons pour l'instant qu'aux tableaux dynamiques, ils sont très pratiques lors de l'utilisation de l'allocation dynamique pour créer des tableaux d'une taille indéfinie lors de l'exécution du programme. L'accès aux éléments du tableau en lecture ou en écriture se fait à l'aide de son indice en temps. T[i] est le lème élément du tableau, il est ce que l'on nomme un indice.

2. Complexité :

Les opérations de base pour un tableau sont :
-L'accès à un élément s'effectue en temps constant, puisqu'il s'agit d'un accès à un élément d'un tableau :O (1)
-L'insertion à la fin s'effectue en temps constant : O (1)
-Ajout d'un élément au début : O(n)
-Suppression d'un élément au début :O(n)
-Suppression d'un élément à la fin :O(n)

3. Avantage :

L'idée des tableaux dynamiques est de conserver l'accès direct en temps constant comme dans un tableau statique. En effet, Un des principaux intérêts de l'allocation dynamique est de permettre à un programme de réserver la place nécessaire au stockage d'un tableau en mémoire dont il ne connaissait pas la taille avant la compilation ainsi que la gestion des pointeurs pour avoir la structure la plus efficace possible, et de permettre un ajout ou une suppression d'éléments (à la fin par exemple) en temps constant.

4. Inconvénient :

Dans une région de mémoire très fragmentée, il peut être coûteux, voire impossible, de trouver un espace contigu pour un grand tableau dynamique, ainsi que l'insertion et la suppression sont assez difficiles dans un tableau car les éléments sont stockés dans des emplacements de mémoire consécutifs et l'opération de décalage est coûteuse.

Le Tas :

1. Définition :

Un tas est un arbre parfait dont les sommets sont étiquetés par des éléments à trier, et tel que l'élément associé à chaque sommet est inférieur ou égal aux éléments associés aux fils de ce sommet. En effet, le tas est une structure de données utile pour le tri par tas. L'idée principale de tri par tas, on sélectionne les minimums successifs. Un tas est aussi le plus souvent un arbre binaire parfait ou complet avec une hauteur minimale, dans lequel toutes les feuilles sont à la même distance de la racine, plus ou moins 1 . On peut faire l'implémentation par tableau T représentant un arbre parfait partiellement ordonnée selon des règles telles que T [1] est la racine, T[2i] et T[2i+1] sont les fils gauche et droite de T[i] s'ils existent.

2. Complexité :

La complexité d'ajouter un élément dans un tas de taille n est au pire log(n).
La complexité d'extraire le min d'un tas est O (log n).
La complexité de tri par tas est de O(n*log(n)).

3. Avantage :

Cet algorithme est de complexité asymptotique optimale, c'est-à-dire que l'on démontre qu'aucun algorithme de tri par comparaison ne peut avoir de complexité asymptotique meilleure.
Le tri par tas se fait en place, c'est-à-dire qu'il ne nécessite pas l'allocation d'une zone mémoire supplémentaire (plus précisément il ne nécessite qu'une allocation d'une zone mémoire de taille O (1).

4. Inconvénient :

Son inconvénient majeur est sa lenteur comparée au tri rapide (qui est en moyenne deux fois plus rapide. Sur un tableau de taille importante, il sera amené à traiter un nombre élevé d'emplacements mémoire dont l'éloignement peut dépasser la capacité du cache, ce qui ralentit l'accès à la mémoire et l'exécution de l'algorithme.

Table de hachage :

1. Définition :

Les tables de hachage sont des tableaux. On y stocke des données à un emplacement déterminé par une fonction de hachage. La fonction de hachage prend en entrée une clé (ex. : une chaîne de caractères) et retourne en sortie un nombre. Ce nombre est utilisé pour déterminer à quel indice du tableau sont stockées les données.

2. Complexité :

La complexité de recherche est O(1).

3. Avantage :

On dit que c'est une complexité en O (1) car on trouve directement l'élément que l'on recherche. En effet, la fonction de hachage nous retourne un indice : il suffit de « sauter » directement à la case correspondante du tableau. Plus besoin de parcourir toutes les cases !

4. Inconvénient :

Le fait de créer un hash à partir d'une clé peut engendrer un problème de collision : à partir de deux clés différentes, la fonction de hachage peut renvoyer la même valeur de hash, et donc donner accès à la même position dans le tableau ". Pour minimiser les risques de collisions, il faut donc choisir soigneusement sa fonction de hachage. Lorsque deux clés ont la même valeur de hachage, les deux éléments associés ne peuvent être stockés à la même position, on doit alors employer une stratégie de résolution des collisions. De nombreuses stratégies de résolution des collisions existent mais les plus connues et utilisées sont le chaînage l'adressage ouvert.

Chapitre 2 : Distance de Levenshtein

Pour remédier au problème des doublons, nous avons utilisé la distance Levenshtein pour mesurer le degré de ressemblance entre deux chaînes de caractères.

Cette distance de Levenshtein mesure le degré de similarité entre deux chaînes de caractères. Elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne à l'autre. C'est une distance au sens mathématique du terme, donc en particulier c'est un nombre positif ou nul, et deux chaînes sont identiques si et seulement si leur distance est nulle. On a aussi des propriétés de symétrie, et l'inégalité triangulaire de la géométrie est ici aussi vérifiée. Par exemple, pour passer de la chaîne NICHE à la chaîne CHIENS, il faut supprimer les lettres N et I, insérer les lettres I, N et S. Donc la distance de Levenshtein entre ces deux chaînes est au plus 5. Une des applications de cette distance est la correction orthographique : lorsqu'une personne tape un mot, on le compare à un dictionnaire. Si le mot est présent, on ne fait rien, sinon, on cherche parmi les mots du dictionnaire ceux dont la distance de Levenshtein au mot tapé est inférieure à une limite donnée. Les mots les plus proches sont suggérés comme remplacement en premier. La mesure de la distance de Levenshtein entre deux chaînes Chaine1 et Chaine2, de longueurs respectives n1 et n2, consiste en la mise en œuvre l'algorithme suivant.

Chapitre 3 : Algorithmes de recherche

Dans cette partie nous allons présenter les algorithmes de recherche qu'on a utilisés
Et leurs complexités afin d'analyser les résultats numériques obtenus.

1) Algorithme naïf :

Dans cet algorithme, on a utilisé un tableau dynamique dans on a stocké les mots qui se trouvent dans le fichier à l'aide de la fonction « MappingToSet », en ce qui concerne l'allocation dynamique de la mémoire on a fait appel à la fonction « Count_Lines » qui compte le nombre de lignes de fichier (car on sait très bien que chaque mot est dans une ligne). La complexité de cet algorithme : O(n) où n représente le nombre de mots contenus dans le fichier .

```
char **MappingToSet(char *filename)
{
    int i=0, lines = Count_Lines(filename)+1;
    char **mot;
    mot=(char **)malloc(lines*sizeof(*mot));
    char *aux;
    aux=(char *)malloc(40*sizeof(char));

    FILE *fp;
    fp=fopen(filename,"r");

    if(fp != NULL)
    {
        while(!feof(fp))
        {
            mot[i]=strdup(fgets(aux,40,fp));
            i++;
        }
    }
    else
    {
        printf("Impossible d\'ouvrir ce fichier \n");
        EXIT_FAILURE;
    }
    fclose(fp);
    return(mot);
}
```

Pour la suppression des doublons, d'abord on a implémenté la distance, on a pris pour cette distance une seuil de similarité :
✓ Levenshtein : 0,8

A l'aide de la fonction « Purge » on compare chaque mot avec tous les mots qui le suivent, si la distance des deux mots comparés est supérieure au seuil correspondant on supprime le deuxième mot ainsi de suite.

```
void Purge(char *filename)
{
    int number_of_words = Count_Lines(filename)+1;
    char **table = (char **)malloc(number_of_words*sizeof(char *));
    char *File_To_Write_In = (char *)malloc(30*sizeof(char));
    table = MappingToSet(filename);

    puts("le nom du nouveau fichier");
    gets(File_To_Write_In);
    FILE *newfile = fopen(File_To_Write_In,"w+");
    int i,j;

    for(i=0;i<number_of_words;i++)
    {
        for(j=i+1;j<number_of_words;j++)
        {
            if(distanceFinal(table[i],table[j]) >= 0.8)
            {
                strcpy(table[j],"\0");
            }
        }
    }

    //upload les donnees dans le nouveau fichier
    for(i=0;i<number_of_words;i++)
    {
        fprintf(newfile,"%s",table[i]);
    }
}
```

La complexité de cet algorithme : O(n^2)

2) Algorithme de tas :

Dans cet algorithme, on trie les mots avec les trois fonctions :
« ajout » : Permet d'ajouter une chaîne de caractères à un tas, sa complexité est de log(n) .

```
void ajout(char** table,char* s)
{
    int i = 0;
    while(table[i])// Compter les cases de tableau
        i++;
    table[i] = (char*)malloc(sizeof(char)*strlen(s));
    table[i] = strdup(s);
    table[i+1] = (char*)malloc(sizeof(char));
    table[i+1] = NULL;
    while(i > 0 && strcmp(table[i],table[(i-1)/2]) < 0)
    {
        permuter(table , i , (i-1)/2);
        i = (i-1)/2;
    }
}
```

« extraire_min » : qui renvoie le min du tas passé en argument et réorganise le tas, sa complexité est de log(n).

```
char* extraire_min(char** table)
{
    if(table[0] == NULL)
        return NULL;
    int i = 0;
    while (table[i])  // Compter les cases de tableau.
        i++;
    char* extrated_string = (char*)malloc(strlen(table[0])*sizeof(char)) ;
    extrated_string = strdup(table[0]);
    free(table[0]);
    table[0] = (char*)malloc(strlen(table[i-1])*sizeof(char));
    table[0] = strdup(table[i-1]);
    int k = 0;
    while( k < (i-1)/2 && (strcmp(table[k],table[2*k+1]) > 0 || strcmp(table[k],table[2*k+2]) > 0))
    {
        if (strcmp(table[2*k+1],table[2*k+2]) < 0)
        {
            permuter(table , k , 2*k+1);
            k = 2*k+1;
        }
        else
        {
            permuter(table , k , 2*k + 2);
            k = 2*k + 2;
        }
    }
    table[i-1] = NULL;
    return extrated_string;
}
```

« LireToTas » : qui transforme un fichier en un tas, à l'aide de la fonction « ajout ». Sa complexité est de O(n*log(n)).

```
char** LireToTas(char* file_name)
{
    char** Tas = (char**)malloc(sizeof(char*));
    char *temp_string = (char*)malloc(25*sizeof(char));

    Tas[0] = (char*)malloc(sizeof(char));
    Tas[0] = NULL;
    FILE* fp;
    fp = fopen(file_name,"r");
    int i = 2;
    while(!feof(fp))
    {
        Tas = (char**)realloc(Tas , i*sizeof(char*));
        i++;
        fscanf(fp,"%s\n",temp_string);
        ajout(Tas,temp_string);
    }
    fclose(fp);
    return Tas;
}
```

« MappingToTable » : qui permet de renvoyer un tas trié avec un ordre alphabétique en appelant les fonctions « LireToTas » et « extraire_min » avec une complexité de O(n*log(n)).

```
char** MappingToTable(char* file_name)
{
    char** Tas = LireToTas(file_name);
    char** Table_Trier = (char**)malloc(sizeof(char*));
    int j = 0;
    while(Tas[0])
    {
        Table_Trier=(char**)realloc(Table_Trier,(j+2)*sizeof(char*));
        Table_Trier[j] = strdup(extraire_min(Tas));
        j++;
    }
    Table_Trier[j] = NULL;
    return Table_Trier;
}
```

Pour la suppression, on supprime à l'aide de notre distance levenshtein , on compare chaque mot avec le suivant jusqu'on trouve pas de doublons (la distance < seuil). C'est le but réel du tri par tas, alors dans ce cas l'algorithme de recherche sera plus optimal, en effet, la complexité de cette fonction au pire des cas sera O(n)=n^2 .

```
char** purge(char* file_name)
{

    int number_of_words = Count_Lines(file_name)+1;
    char **table = (char **)malloc(number_of_words*sizeof(char *));
    table = MappingToTable(file_name);

    int i,j;

    for(i=0;i<number_of_words;i++)
    {
        for(j=i+1;j<number_of_words;j++)
        {
            if(dis(table[i],table[j]) >= 0.8)
            {
                strcpy(table[j],"\0");
            }
        }
    }
    return table;
}
```

3) Algorithme de Hachage :

Dans cet algorithme, on procède avec cette manière : pour chaque mot, on calcule HashCode avec la fonction « mot_cle ».

```
int mot_cle(char *string,char* filename)
{
    int i=0,sum=0,n = count_lines(filename);
    while(string[i]!='\0')
    {
        sum+=string[i];
        i++;
    }
    return sum%n;
}
```

Nous utilisons la fonction « AjoutMot » pour ajouter un data <mot> dans une liste chaînée .

```
void AjoutMot(node **head,char *string)
{
    node *nv_node = malloc(sizeof(node));
    nv_node->data=strdup(string);
    nv_node->next=NULL;

    if(*head==NULL)
    {
        *head=nv_node;
    }
    else
    {
        node *temp=*head;
        while(temp->next!=NULL)
        {
            temp=temp->next;
        }
        temp->next=nv_node;
    }
}
```

La complexité de la fonction « AjoutMot » au pire des cas :O(n).

On utilise aussi la fonction «insert_into_table» qui stock les données dans le hashtable en se basant sur le chaînage .

```
void insert_into_table(node **hashtable ,char *string,char *filename)
{
    int index=mot_cle(string,filename);

    if(hashtable[index]==NULL)
    {
        hashtable[index]=malloc(sizeof(node));
        node *head=NULL;
        AjoutMot(&head,string);
        hashtable[index]=head;
    }
    else
    {
        node *head=hashtable[index];
        AjoutMot(&head,string);
        hashtable[index]=head;
    }
}
```

cette fonction « insert_into_linked » est réservée pour stocker les données d'un fichier dans un tableau des listes chaînées .

```
node** insert_into_linked(char *file_name)
{
    int i,lines=count_lines(file_name);
    node **linked=calloc(lines,sizeof(node*));
    char **table=mappingtoset(file_name);

    for(i=0;i<lines;i++)
    {
        insert_into_table(linked,table[i],file_name);
    }

    return linked;
}
```

On utilise la fonction « mappingtohash » : elle prend en paramètre un fichier et le transforme en un tableau avec un chaînage en faisant appel à la fonction « insert_into_table ».
La complexité de cet algorithme est de O(n^2).

```
hash* mappingtohash(char *file_name)
{
    int i,j=0,lines=count_lines(file_name);
    node **linked_list=calloc(lines,sizeof(node *));
    hash *hashtable=malloc(sizeof(hash)*lines);

    linked_list=insert_into_linked(file_name);

    for(i=0;i<lines;i++)
    {
        if(linked_list[i]!=NULL)
        {
            hashtable[j].string=strdup(linked_list[i]->data);
            hashtable[j].key=i;
            j++;
            node *temp=linked_list[i]->next;
            while(temp!=NULL)
            {
                hashtable[j].string=strdup(temp->data);
                hashtable[j].key=i;
                j++;

                temp=temp->next;
            }
        }
    }

    return hashtable;
}
```

Pour la suppression, on compare en parcourant la table de hachage avec toutes les cases suivantes puis on donne la clé des doublons -1 . Ensuite on stock les cases non redoublantes dans un autre tableau « filtered table » et le retourner à la fin .

```
char** Purge(char *file_name)
{
    int i=0,j,t=1,key,lines=count_lines(file_name);
    char **Filtredtable=malloc(sizeof(char*));
    hash *hashtable = mappingtohash(file_name);
    for(i=0;i<lines;i++)
    {
        key=hashtable[i].key;
        for(j=i+1;j<lines;j++)
        {
            if(distanceFinal(hashtable[i].string,hashtable[j].string)>=0.8)
            {
                hashtable[j].key=-1;
            }
        }
    }

    for(i=0;i<lines;i++)
    {
        if(hashtable[i].key!=-1)
        {
            Filtredtable[t-1]=(char *)malloc(sizeof(char)*strlen(hashtable[i].string));
            strcpy(Filtredtable[t-1],hashtable[i].string);
            //Filtredtable[i]=strdup(hashtable[i].string);
            Filtredtable=realloc(Filtredtable,(t+1)*sizeof(char *));
            t++;
        }
    }
    Filtredtable[t-1]=NULL;
    free(hashtable);

    return Filtredtable;
}
```

Chapitre 4 : Simulation et résultats numériques

Même si l'ordinateur renvoie les résultats rapidement, on ne remarque pas la différence entre les différents types d'algorithmes de recherche et pour cela on va utiliser la fonction Clock ( ) qui fait partie de la bibliothèque <time.h> pour calculer le temps d'exécution.
Comme vous voyez, on vous représente le code avec lequel on a travaillé pour faire les calculs.
voilà l'exemple de fonction time clock() dans le main :

```
double la_dure ;
clock_t start , finish ;

start = clock() ;
filtredtable = purge(File_To_Read);
finish = clock() ;

puts("nom du fichier a ecrire");
gets(File_To_write);

FILE *f_out = fopen(File_To_write,"w+");

printf("\n\t nous avons : %d mots \n\n",lines);
printf(" \t************************ \n");
 la_dure = (double)(finish-start) / CLOCKS_PER_SEC ;
printf("\n\n****************************************** ");
 printf(" \n\n ***  la duree est %f secondes *** \n\n" , la_dure);
printf("****************************************** \n\n");
```

Et voilà un exemple d exécution :

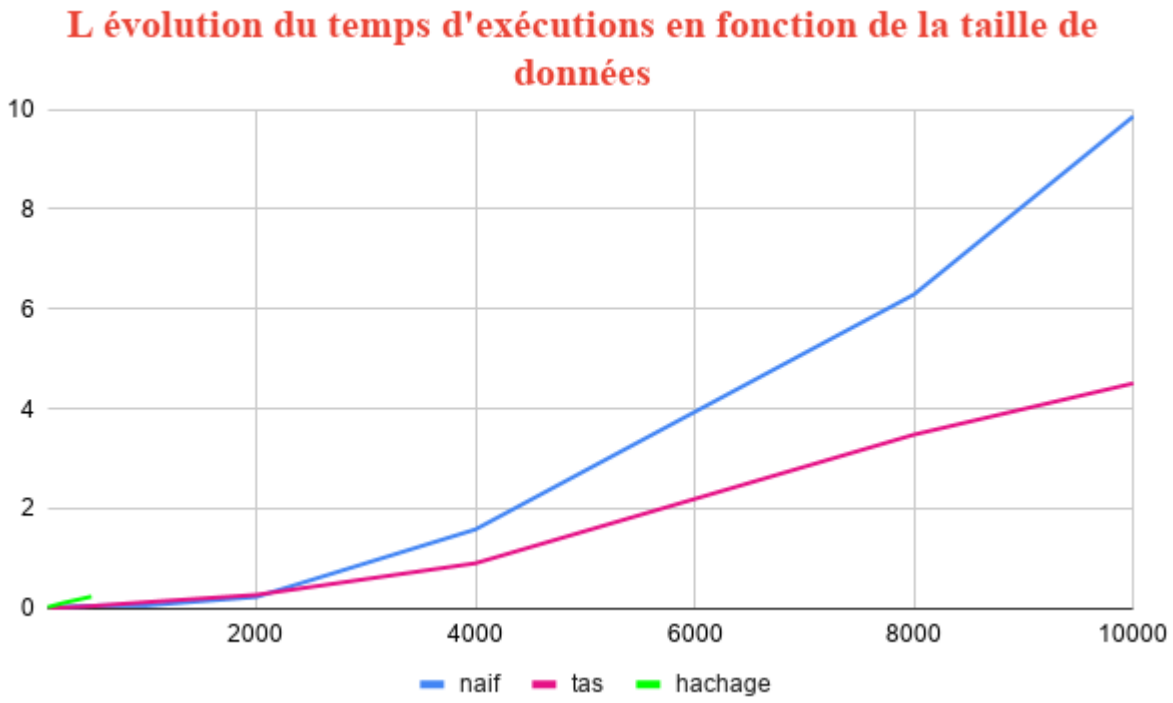
```
names.txt
nom du fichier a ecrire
new.txt

          nous avons : 10000 mots

         ************************


**************************************************

 ***  la duree est 6.544000 secondes ***

**************************************************
```

❖ Graphes représentant l'évolution du temps d'exécution de chaque algorithmes en fonction de la taille des données

A chaque fois, on change la taille des données , puis on observe le temps d'exécution. Voilà le temps d'exécution par seconde concernant les trois algorithmes.

	naif	tas	hachage
100	0,004	0,002	0,018
250	0,025	0,012	0,107
500	0,048	0,041	0,231
1000	0,052	0,119	
2000	0,222	0,265	
4000	1,578	0,899	
8000	6,292	3,48	
10000	9,864	4,509	

D'après les résultats obtenus, on est assuré bien que le temps d'exécution dans chaque algorithme augmente avec la taille des données ce qui est tout à fait logique et normal. On a essayé de représenter ces résultats sur Excel ; On vous représente dans la suite la courbe obtenue.



Interprétation : D'après un premier constat de la courbe ci-dessus, on voit bien que le temps d'exécution de l'algorithme naïf est beaucoup plus grand que les deux autres algorithmes ceci revient au fait que l'algorithme naïf procède une technique de suppression plus lente que les autres(Il compare chaque case avec toutes les cases suivantes donc sa complexité est la plus élevée(O(n^2)) , ceci montre bien l'utilité de l'algorithme tas qui a l'aide de la structure tas supprime les doublons dans un temps plus court.

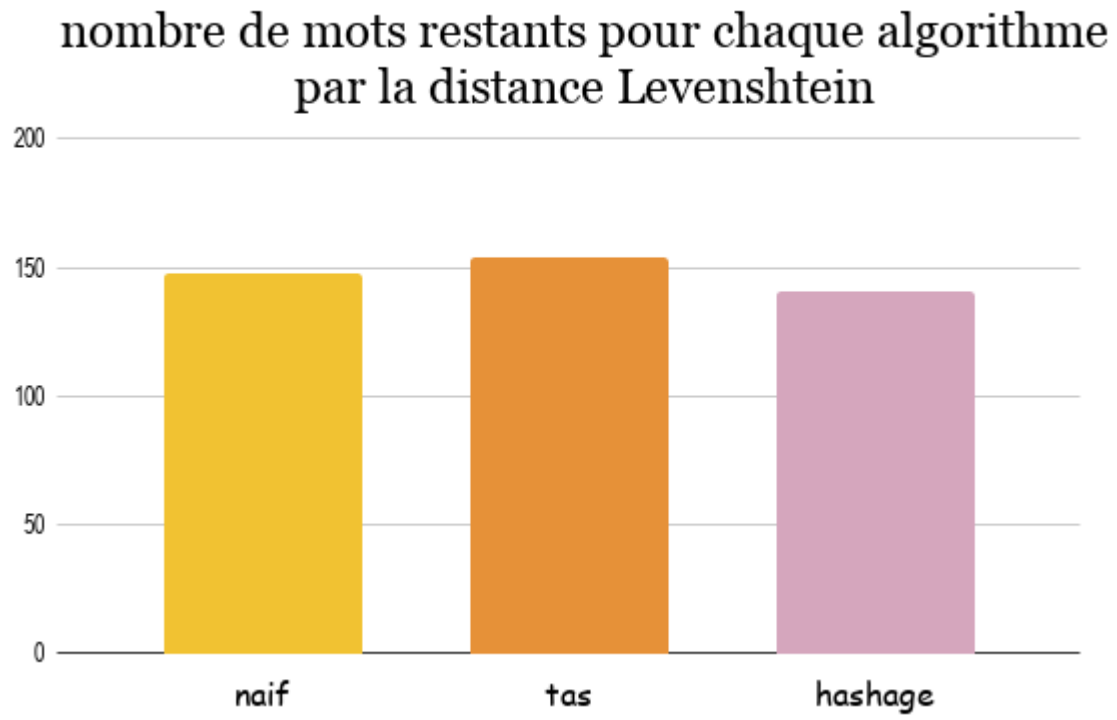
❖ Graphe représentant le nombre de mots supprimés pour chaque algorithme :

Ensuite, pour vérifier l'efficacité des trois algorithmes, on choisit un fichier de 10000 mots, sauf pour l'algorithme de hachage on prend 500 mots et on utilise les trois algorithmes .

algorithme utilisée	naif	tas	hashage
nombre de mots supprimées	9852	9846	359

algorithme utilisée	naif	tas	hashage
nombre de mots restantes	148	154	141

On obtient la figure suivante :



Interprétation :

On constate que pour le même fichier, le nombre de mots restants varie d'un algorithme à un autre .
On voit que le nombre de mots supprimés est presque le même .
On observe aussi, que l'algorithme Tas laisse des doublons supplémentaires, en le comparant avec les deux autres, ceci revient au fait que l'algo Tas ne prend pas en considération les doublons qui se diffèrent par une lettre ou plusieurs lettres au début du doublon, donc on trouve des doublons qui se ressemblent mais ils sont très loin dans le Tableau Tas .

L'algorithme Naïf et Hash utilisent la même technique de suppression, ils comparent chaque mot avec tous les mots qui le suivent .

Conclusion

On conclut que chaque algorithme a ses avantages et ses inconvénients, soit au niveau de son efficacité, soit au niveau de son temps d'exécution.
Ce projet a présenté pour nous une occasion précieuse pour améliorer et développer nos compétences et aussi pour expérimenter de nouvelles circonstances de travail.

Bibliographie

Pour réaliser ce projet, il nous a fallu que nous accédions à des sites internet et des bouquins algorithmiques.

-les polycopiés du cours
-les vidéos sur Youtube
-introduction à l'algorithme
-www.geeksforgeeks.org
-www.openclassroom.com
-www.KooR.fr
-www.developpez.com