RAPPORT PROJET ANALYSE D'ALGORITHMES ET VALIDATION DE PROGRAMMES

Comparaison de plusieurs algorithmes résolvant le même problème :

Calcule de distance Hamming

Réalisé par :

ELFRANI Khadija

Encadré par :

M. Devan Sohier

Année Universitaire 2022-2023

INSTITUT DES SCIENCES ET TECHNIQUES DES YVELINES - 10-12 AV de l'Europe, 78140 Vélizy Villacoublay
Tél : 01 39 25 38 50 site web : www.isty.uvsq.fr

Sommaire

I - Introduction

La distance de Hamming est un problème fondamental en informatique théorique qui est utilisé dans de nombreux domaines, tels que la correction d'erreur, la compression de données et la cryptographie. La distance de Hamming est une mesure de la différence entre deux chaînes de bits ou deux chaînes de caractères de même longueur, et elle est définie comme le nombre de positions où les deux chaînes diffèrent. L'objectif de ce projet est de concevoir et d'implémenter et de comparer plusieurs algorithmes efficaces pour calculer la distance de Hamming entre deux chaînes de caractères ou deux chaînes de bits.

J'ai choisi de travailler sur le problème de la distance de Hamming pour mon projet d'informatique parce que je suis passionné par la cryptographie et la sécurité des données. La distance de Hamming est un concept important en cryptographie car elle permet de mesurer la distance entre deux codes secrets, ce qui est essentiel pour détecter les erreurs de transmission ou les modifications malveillantes des données. J'ai également choisi ce sujet parce que je suis intéressé par les applications pratiques de la théorie de l'information et de l'algorithmique, et la distance de Hamming est utilisée dans de nombreux domaines, tels que la correction d'erreurs dans les transmissions de données, la classification de données en apprentissage automatique, on peut aussi prendre l'exemple de la recherche de mots dans un dictionnaire, si le mot recherché est introuvable on peut trouver les mots les plus proches du celui-ci.

II - Présentation de l'algorithme de base

La distance de Hamming et une notion qui est définie par Richard Hamming, c'est une distance au sens mathématique du terme. À deux suites de symboles de même longueur, elle associe l'entier désignant le cardinal de l'ensemble des symboles de la première suite qui diffèrent de la deuxième.
Ci-dessous, un pseudo code simple qui calcule la distance Hamming de deux chaînes de caractères :

Algorithme de Distance Hamming (str1,str2) ⇒ distance : entier
Variables :
    i : indice
Précondition : longueur(str1) = longueur(str2)
Début :
    distance = 0 // initialiser le compteur de distance hamming
    pour i allant de 0 à longueur(str1) faire
        si str1[i] != str2[i] alors
            distance++ // incrémenter la distance si deux lettre sont les mêmes
        finSi
    finPour
    renvoyer distance
fin

Cet algorithme consiste à parcourir les deux mots par une boucle. Pour chaque itération de boucle : l'algorithme compare le caractère de l'index i du Mot1 avec le caractère de l'index i du Mot2, si les deux caractères sont différents, on incrémente la variable distance.
Voici un exemple de déroulement de l'algorithme avec deux chaînes de taille n=8

i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
str1	**A**	**L**	**G**	**O**	**R**	**I**	**T**	**H**	**M**	**E**
str2	**A**	**L**	**G**	**U**	**R**	**I**	**Y**	**H**	**M**	**E**
distance	**0**	**0**	**0**	**1**	**1**	**1**	**2**	**2**	**2**	**2**

La distance de Hamming à la fin des itérations est d(str1,str2) = 2 , ce qui indique que les deux mots sont proches.

III - Présentation du langage

J'ai choisi d'utiliser le langage C++ pour implémenter le problème de la distance de Hamming pour plusieurs raisons. Tout d'abord, j'ai été attiré par le fait que je n'ai pas eu l'opportunité de travailler avec ce langage pendant mon cursus, et donc c'était une occasion pour moi de l'explorer davantage et d'acquérir des compétences supplémentaires en programmation. De plus, C++ est connu pour sa performance et son efficacité, ce qui est important pour l'étude de la complexité de l'algorithme.
Enfin, il est plus facile de manipuler les chaînes de caractères avec C++ par rapport à C, grâce aux bibliothèques standard de C++ qui fournissent des fonctionnalités pour manipuler les chaînes de caractères de manière efficace et élégante.

IV - Mise en oeuvre

4.1. Spécification du problème

Jusqu'à ici, on a compris que la distance hamming représente le nombre de différences entre deux chaînes de caractères A et B de longueur n, cette distance est caractérisé par :

- Elle est toujours positive (>=0)
- S'elle est égale à 0 alors les deux mots sont identiques
- Elle ne peut pas être supérieure au nombre de caractères des deux chaînes données.
- Elle est symétrique, c'est-à-dire que la distance entre la chaîne A et la chaîne B est la même que la distance entre la chaîne B et la chaîne A.

La distance hamming est alors le cardinal de l'ensemble i tel que i est borné par 0 et n , et A[i] différent de B[i]. On peut exprimer mathématiquement cela avec la spécification suivante :

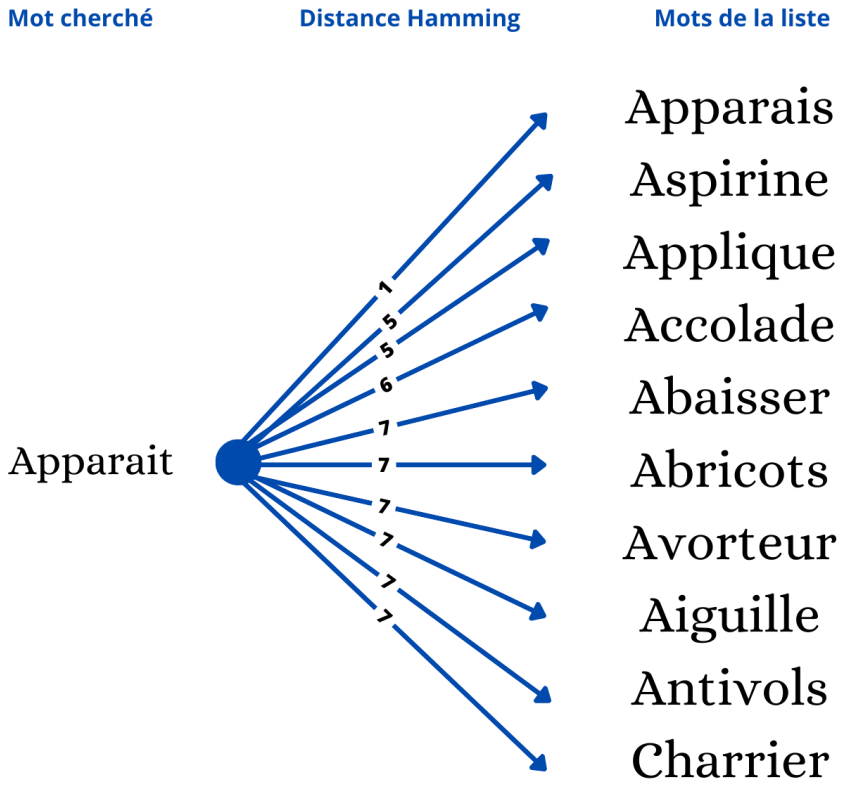
$$\textbf{DistanceHamming}(A,B) = |\{i \mid 0 \leq i \leq n \wedge A[i] \neq B[i]\}|$$

4.2. Jeu de test

Le jeu de test que je propose pour valider les algorithmes sera une petite simulation de recherche dans un dictionnaire comme suit :
Une petite liste de mots de même longueur est enregistrée, l'utilisateur entre un mot de même longueur, l'algorithme alors sera utilisé pour calculer la distance hamming entre le mot fournie et tous les mots de cette liste, et puis affiche le mot le plus proche du mot donnée.

4.3. Résultat attendu

Pour pouvoir valider nos algorithmes, tous ces derniers doivent nous retourner le même résultat (les mêmes mots).
On va spécifier la liste de 10 mots de longueur n=8 suivante :
{Abaisser,Abricots,Accolade,Aspirine,Avorteur,Aiguille,Antivols ,Apparais,Applique,Charrier}
Le mot entrée (pour le test) sera le mot : **{Apparait}**
L'algorithme doit alors calculer la distance hamming entre "Apparait"et tous les mots de la liste, puis afficher le mot le plus proche de celui-ci.
Les distances hamming entre Apparait et tous les mots de la liste est (en ordre croissant) est comme suit :



Le mot le plus proche est **"Apparais"** , l'algorithme doit alors afficher ce mot.

V- Algorithme de Hamming

5.1. Algorithme naïf

Un algorithme naïf est souvent la première solution qui vient à l'esprit, mais peut ne pas être la meilleure en termes de performance.

Cette version de l'algorithme hamming est implémentée à partir du pseudocode présenté.

5.1.1. Test et affichage

Après avoir présenté notre jeu de test, on va maintenant tester notre programme pour s'assurer que le résultat est bien le résultat attendu :

```
PS C:\Users\khadi\Projet-AAVP> ./HAMMING Apparait
Le mot le plus proche de "Apparait" est : Apparais (distance de Hamming = 1)
PS C:\Users\khadi\Projet-AAVP>
```

Cet algorithme a alors validé notre jeu de test.

5.1.2. Complexité

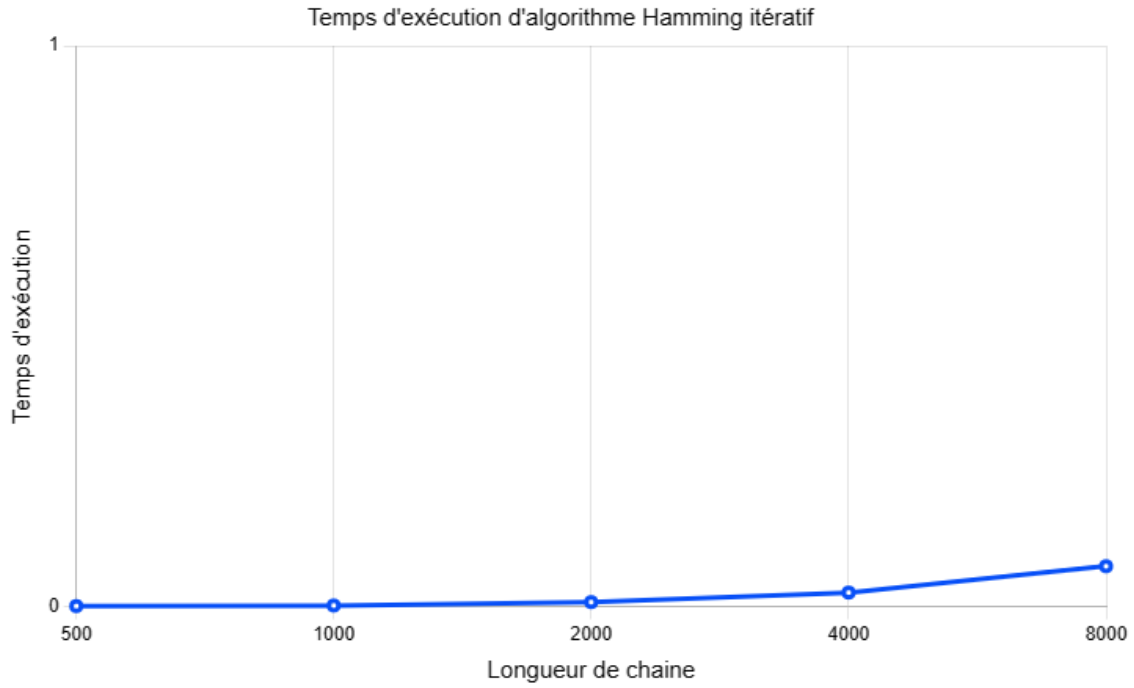
La complexité de ce programme dépend de la longueur des deux chaînes d'entrée. Dans le pire des cas, où les deux chaînes sont de longueur n, la complexité de la boucle for est O(n), car elle itère sur chaque caractère des deux chaînes et effectue une comparaison. Par conséquent, la complexité totale du programme est de O(n).
En moyenne, la complexité sera également O(n) puisque chaque caractère de chaque chaîne de caractères doit être comparé une fois.

5.1.3. Temps d'exécution

Le temps d'exécution dépend principalement de la longueur des chaînes de caractères en entrée, afin de tester le temps d'exécution avec des chaînes de longueurs différentes, j'ai utilisé la bibliothèque standard C++ <chrono>, cette bibliothèque fournit un moyen de mesurer le temps en utilisant des horloges haute résolution.
J'ai testé le programme avec des chaînes de caractères de grande longueur et j'ai noté le temps d'exécution à chaque test.
On obtient les résultat suivantes :

Longueur de la chaîne	Temps d'exécution
500	0.001000 s
1000	0.001997 s
2000	0.008022 s
4000	0.024985 s
8000	0.072002 s

Graphiquement :



5.1.4. Limitations

L'algorithme naïf de distance de Hamming présenté dans le code ci-dessus a plusieurs limitations :

- L'algorithme ne gère pas le cas oú une des chaînes est vide.
- L'algorithme suppose que les deux chaînes de caractères contiennent uniquement des caractères ASCII, mais cela ne fonctionnera pas si les chaînes contiennent des caractères Unicode ou d'autres encodages.

5.2. Algorithme Récursif

Un algorithme récursif est un algorithme qui s'appelle lui-même pour résoudre un problème en le décomposant en sous-problèmes plus simples.
Cette version de programme est récursive , elle est implémentée à partir du pseudo code suivant :

Algorithme Récursif de Distance Hamming(str1,str2) ⇒ distance : entier
Variables :
i : indice
Précondition : longueur(str1) = longueur(str2)
Début :
si la longueur(str1) == 0 alors

```
            retourner la longueur de str2
        si la longueur(str2) == 0 alors
            retourner la longueur de str1
         si le premier caractère de str1 == premier caractère de str2  alors
            retourner hamming_distance_recur(str1[1:], str2[1:])
        sinon :
            distance1 = hamming_distance_recur(str1[1:], str2[1:])
            distance2 = hamming_distance_recur(str1, str2[1:])
            distance3 = hamming_distance_recur(str1[1:], str2)
            min_distance = distance1
            si distance2 < min_distance alors
               min_distance = distance2
            si distance3 < min_distance alors
               min_distance = distance3
               retourner 1 + min_distance
fin
```

L'algorithme commence par vérifier si l'une des deux chaînes est vide, auquel cas la distance de Hamming est simplement la longueur de l'autre chaîne. Si les deux chaînes ne sont pas vides, l'algorithme compare le premier caractère de chaque chaîne. Si les caractères sont identiques, la fonction récursive est appelée avec les chaînes restantes. Si les caractères sont différents, trois appels récursifs sont effectués pour les sous-chaînes de str1 et str2. La distance de Hamming est ensuite déterminée comme la plus petite distance trouvée parmi ces trois appels, à laquelle 1 est ajouté pour tenir compte de la différence entre les deux premiers caractères.

5.2.1. Test et affichage

On va maintenant tester notre programme pour s'assurer que le résultat est bien le résultat attendu :

```
g++ -Wall -Wextra -g -o HAMMING main.o algo_hamming.o hamming_recursif.o
PS C:\Users\khadi\Projet-AAVP> ./HAMMING Apparait
Le mot le plus proche de "Apparait" est : Apparais (distance de Hamming = 1)
PS C:\Users\khadi\Projet-AAVP>
```

Et donc cet algorithme aussi nous retourne le résultat prévu, il est alors valide.

5.2.2. Complexité

Pour calculer la complexité de cet algorithme, on peut utiliser la méthode de substitution ;

Prenons l'exemple où les deux chaînes ont la même longueur n, Dans le pire des cas, chaque appel récursif doit être exécuté avec deux sous-chaînes différentes. Cela signifie qu'il y a un total de 2^n appels de fonction.

Les opérations principales de la fonction sont des appels récursifs et des comparaisons de caractères, ces appels sont de complexité de T(n-1).

Les comparaisons de caractères ont une complexité constante, donc on peut les ignorer pour une approximation de la complexité.

On utilisant le méthode substitution on obtient :

$T(n) = 2T(n-1) + c$

$= 2(2T(n-2) + c) + c$

$= 2^2T(n-2) + 2c + c$

$= 2^3T(n-3) + 2^2c + 2c + c$

$= 2^kT(n-k) + (2^{(k-1)} + 2^{(k-2)} + ... + 2 + 1)c \Rightarrow$ *Somme géométrique*

On simplifie en utilisant la formule 2^k - 1 et en remplaçant k par n, on obtient :

T(n) = 2^nT(0) + (2^n - 1)c , la complexité de l'algorithme est alors exponentielle en fonction de la longueur n des chaînes ⇒ O(2^n).

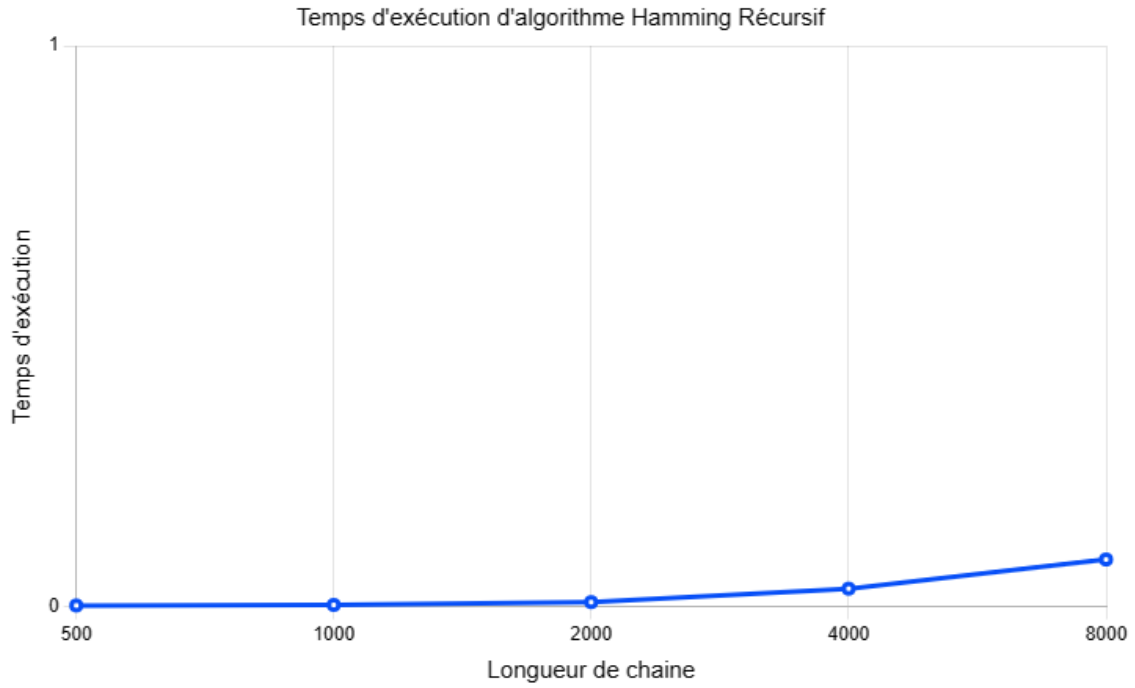
La complexité moyenne de l'algorithme dépend de la distribution des caractères dans les chaînes de caractères en entrée et peut être de O(n^2) ou de O(n log n). Cependant, la complexité dans les cas les plus courants est probablement plus proche de O(n log n).

5.2.3. Temps d'exécution

Afin de comparer le temps d'exécution du ce programme, On utilise encore la bibliothèque <chrono> pour mesurer le temps qui s'écoule avant et après l'appel à la fonction avec des chaîne de longueur différentes, on note les résultats à chaque fois et on obtient:

Longueur de la chaîne	**Temps d'exécution**
500	0.002002 s
1000	0.003003 s
2000	0.007994 s
4000	0.031997 s
8000	0.083999 s

Graphiquement :



⇒ On constate clairement que la version récursive de cet algorithme a une complexité plus élevée et donc un temps d'exécution plus long que son équivalent itératif.
Cela est dû au fait que l'utilisation de la récursion peut entraîner une surcharge supplémentaire sur la pile d'appels de fonctions, ce qui peut entraîner une utilisation excessive de la mémoire et une diminution des performances.
Cependant, il est important de noter que dans certains cas, l'implémentation récursive peut être plus élégante, plus facile à comprendre et à maintenir que l'implémentation itérative. En outre, il existe des cas où l'utilisation de la récursion peut simplifier considérablement la résolution d'un problème.

5.2.4. Limitations

L'algorithme récursif de distance de Hamming présenté dans le code ci-dessus a plusieurs limitations :

- ➔ L' algorithme utilise une grande quantité de mémoire car il crée des sous-chaînes répétées à chaque appel de la fonction récursive. Cela peut causer des problèmes de mémoire si les chaînes sont très longues.
- ➔ L' algorithme a une complexité exponentielle en fonction de la longueur des chaînes, car chaque appel de la fonction récursive divise la longueur des chaînes d'une unité. Cela peut rendre l'algorithme très lent pour des chaînes de taille importante.
- ➔ L' algorithme suppose que les deux chaînes ont la même longueur. Si ce n'est pas le cas, il produira une erreur de segmentation, car il tentera d'accéder à des caractères qui n'existent pas dans la chaîne la plus courte.

VI- Algorithme de Levenshtein

6.1. Présentation

L'algorithme de Levenshtein, également connu sous le nom de distance d'édition, est utilisé pour mesurer la différence entre deux chaînes de caractères. Il calcule le nombre minimal d'opérations d'insertion, de suppression et de substitution nécessaires pour transformer une chaîne en une autre. Cette distance de Levenshtein peut être utilisée pour comparer la similitude entre deux chaînes de caractères, par exemple pour la correction orthographique ou la recherche de chaînes similaires, cet algorithme n'impose pas que les deux chaînes soient de la même longueur, il résout alors les limitations que posent les algorithmes de hamming.
Prenons l'exemple suivant :
On veut convertir le mot BIRD de longueur n = 4 vers le mot HEARD de longueur m = 5, on doit alors effectuer les opérations suivantes :

- ➔ Remplacer « I » de BIRD par « A » ⇒ BIRD devient BARD
- ➔ Remplacer « B » de BARD par « E » ⇒ BARD devient EARD
- ➔ Ajouter H au début. Enfin, nous obtenons HEARD.

	*	B	I	R	D
sub	*	B	**A**	R	D
sub	*	**E**	A	R	D
ins	**H**	E	A	R	D

On constate qu'on a effectué 3 opérations (2*sub + ins) , La distance Levenshtein alors est : Levenshtein(BIRD,HEARD) = 3

6.2. Version Récursive de Levenshtein

6.2.1. Pseudocode :

Le pseudocode de l'algorithme en version récursive est le suivant :

Algorithme Récursif de Levenshtein (s1, s2, n, m) : entier
Variables :
cost : entier

Début :
si n = 0 alors retourner m
si m = 0 alors retourner n
si s1[n-1] = s2[m-1] alors cost = 0 sinon cost = 1

ins = distance(s1, s2, n, m-1) + 1
suppr = distance(s1, s2, n-1, m) + 1
subst = distance(s1, s2, n-1, m-1) + cost

retourner min(ins, suppr, subst)
fin

Cet algorithme se sert de la récursivité pour calculer la distance de Levenshtein entre deux chaînes de caractères, il commence par vérifier si n (la longueur de la première chaîne) ou m (la longueur de la seconde chaîne) est égal à 0 ; si c'est le cas, il retourne m ou n, respectivement, si ce n'est pas le cas, il calcule la distance en comptant le nombre d'opérations requises pour transformer une chaîne en l'autre. Dans ce cas, les opérations autorisées sont l'insertion, la suppression et la substitution, il calcule alors la distance en utilisant une fonction récursive pour calculer les distances pour chaque opération.
Une fois toutes les distances calculées, il retourne la distance minimale.

6.2.2. Etude de validité

L'algorithme prend deux mots A et B de longueur n et m.
Dans le cas où les deux mots sont complétement différent, la distance de Levenshtein retourne le maximum entre les deux longueur de mot A et mot B, la distance est alors borné par le maximum(n,m) .
On sait aussi que la distance est toujours positive, elle est égale à 0 si A et B sont identiques ou un nombre qui est inférieur à n et à m si on les deux mots ont des caractères en commun. On peut alors écrire comme spécification :

$$\textbf{Livenshtein(A,B) = min(d[i-1,j] + 1,d[i,j-1] + 1, d[i-1,j-1] + cost)}$$

Avec: $$\textbf{|n-m| <= Livenshtein(A,B) <= max(n,m)}$$

6.2.3. Test et affichage

Après avoir implémenté le pseudocode présenté de Levenshtein, on doit maintenant tester son fonctionnement. On exécute notre programme avec les deux mots HEARD et BIRD et vérifions si le résultat est le même qu'attendu :

```
g++ -Wall -Wextra -g -o HAMMING main.o algo_hamming.o hamming_recursif.o Levenshtein_recursif.o
PS C:\Users\khadi\Projet-AAVP> ./HAMMING BIRD HEARD
La distance Levenshtein entre "BIRD" et "HEARD" est : 3
Le temps d'execution est de : 0.000000 s
```

La distance d'édition est alors 3 comme prévu.

6.2.4. Complexité

On considère mot A de longueur n et mot B de longueur m.

L'algorithme commence par vérifier si l'un des mots est vide, si c'est le cas il retourne une distance qui est égale au mot qui n'est pas vide, alors la complexité est T = n ou T= m Sinon si les deux mots sont pas vide on effectue une suppression et une insertion et une substitution. On peut définir la fonction T(n,m) comme étant le nombre d'opérations nécessaires pour calculer la distance de Levenshtein entre les chaînes de longueur n et m, T(n,m) est alors donné par :

$$T(n,m) = T(n-1,m-1) + T(n-1,m) + T(n,m-1)$$

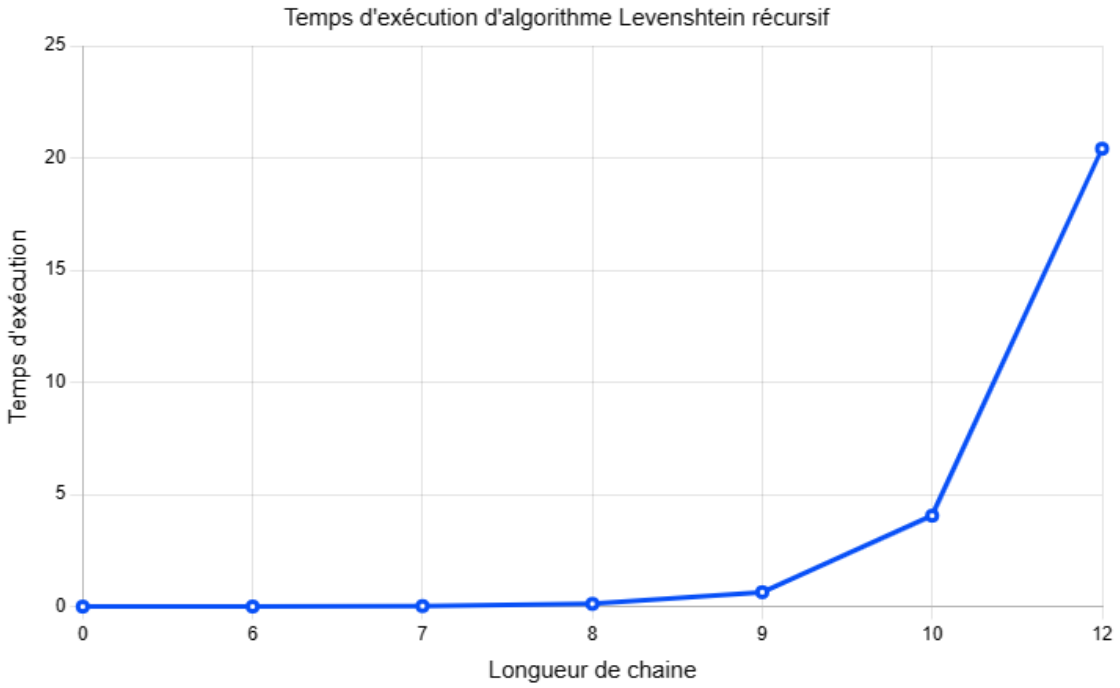
Cette expression montre que le nombre d'opérations requises est déterminé par la somme des opérations requises pour les sous-problèmes moins un, ce qui nous donne une complexité d'ordre **O(3^min(m,n))**.

6.2.5. Temps d'exécution

Le temps d'exécution est proportionnel aux longueurs n et m des deux chaînes donnée, afin de comparer le temps d'exécution du cet algorithme, On utilise encore la bibliothèque <chrono> pour mesurer le temps qui s'écoule avant et après l'appel à la fonction avec des chaîne de longueur différentes, on note les résultats à chaque fois et on obtient:

Longueur de la chaîne	Temps d'exécution
6	0.003989 s
7	0.024993 s
8	0.122009 s
9	0.642990 s
10	4.053992 s
12	20.409985 s

Graphiquement :



⇒ On constate que à partir de chaine de caractère de longueur 10, le temps d'exécution augmente dramatiquement, ce qui entraîne des problème d'overflow avec des chaînes d'une grande longueur, cet implémentation et une traduction direct de l'algorithme de Levenshtein, mais elle est pas de tout efficace, ci-suite, on va essayer de dérécursiver l'algorithme pour voir si ça donne une meilleur performance.

6.3. Dérécursivation de Levenshtein

6.3.1. Pseudocode

La dérécursivation est le processus de transformation d'une fonction récursive en une fonction itérative. Les appels récursifs peuvent consommer beaucoup de mémoire et de temps d'exécution, surtout pour des entrées importantes.

Dans cette partie, on va essayer de dérécursiver l'algorithme de Levenshtein pour voir si on obtient une meilleure performance par rapport à la version récursive.
Le pseudocode de l'algorithme en version itérative est le suivant :

Algorithme Itératif de Levenshtein (s1, s2, n, m) : entier
Variables :
cost : entier
d : tableau[n+1, m+1] d'entiers
i, j : entiers
Début :
Pour i de 0 à n faire :
d[i, 0] = i

```
        Fin Pour
        Pour j de 0 à m faire :
            d[0, j] = j
        Fin Pour

        Pour j de 1 à m faire :
                Pour i de 1 à n faire :
                        si s1[i-1] = s2[j-1] alors cost = 0 sinon cost = 1
                        ins = d[i, j-1] + 1
                        suppr = d[i-1, j] + 1
                        subst = d[i-1, j-1] + cost
                        d[i, j] = min(ins, suppr, subst)
                Fin Pour
        Fin Pour
    retourner d[n, m]
fin
```

Dans cette version itérative, nous créons une matrice d de taille (n+1) x (m+1) pour stocker les distances calculées.

Nous initialisons les premières lignes et colonnes de la matrice en fonction de la longueur des chaînes s1 et s2.

Ensuite, nous parcourons la matrice et calculons la distance entre s1 et s2 pour chaque sous-chaîne de s1 et s2 jusqu'à atteindre s1[n] et s2[m], nous stockons la distance calculée dans la matrice d, et finalement, nous retournons la distance entre s1 et s2 en accédant à la case d[n, m].

6.3.2. Etude de validité

L'algorithme prend deux mots A et B de longueur n et m.

A la fin de l'algorithme itératif de Levenshtein, la valeur de la distance se trouve dans la dernière case de la matrice. On peut alors écrire comme spécification :

Levenshtein(A,B) = min(d1,d2,d3)

avec : ⇒ **d1 = Levenshtein(i-1,j) + 1**

⇒ **d2 = Levenshtein(i,j-1) + 1**

⇒ **d3 = Levenshtein(i-1,j-1) + cost**

(et : s1[i-1] = s2[j-1] alors cost = 0 sinon cost = 1)

6.3.3. Test et affichage

Après avoir implémenté le pseudocode ci-dessus de la version itérative de Levenshtein, on doit maintenant tester son fonctionnement. On exécute notre programme avec les deux mots HEARD et BIRD et vérifions si le résultat est le même qu'attendu :

```
g++ -Wall -Wextra -g -o HAMMING main.o algo_hamming.o hamming_recursif.o Levenshtein_recursif.o Levenshtein_iteratif.o
PS C:\Users\khadi\Projet-AAVP> ./HAMMING BIRD HEARD
La distance Levenshtein entre "BIRD" et "HEARD" est : 3
Le temps d'execution est de : 0.000000 s
PS C:\Users\khadi\Projet-AAVP>
```

Le programme affiche la distance entre les deux mots qui est égale à 3 comme attendu.

6.3.4. Complexité

Dans cette version, le nombre d'opérations effectuées dans la boucle interne est n*m, car nous parcourons chaque sous-chaîne de s1 et s2 une fois. Chaque sous-chaîne nécessite trois opérations (une comparaison de caractères, une addition et un accès à un élément de la matrice), donc le coût de chaque sous-chaîne est de 3c.

On peut exprimer la formule de la complexité temporelle de l'algorithme de Levenshtein comme suit :

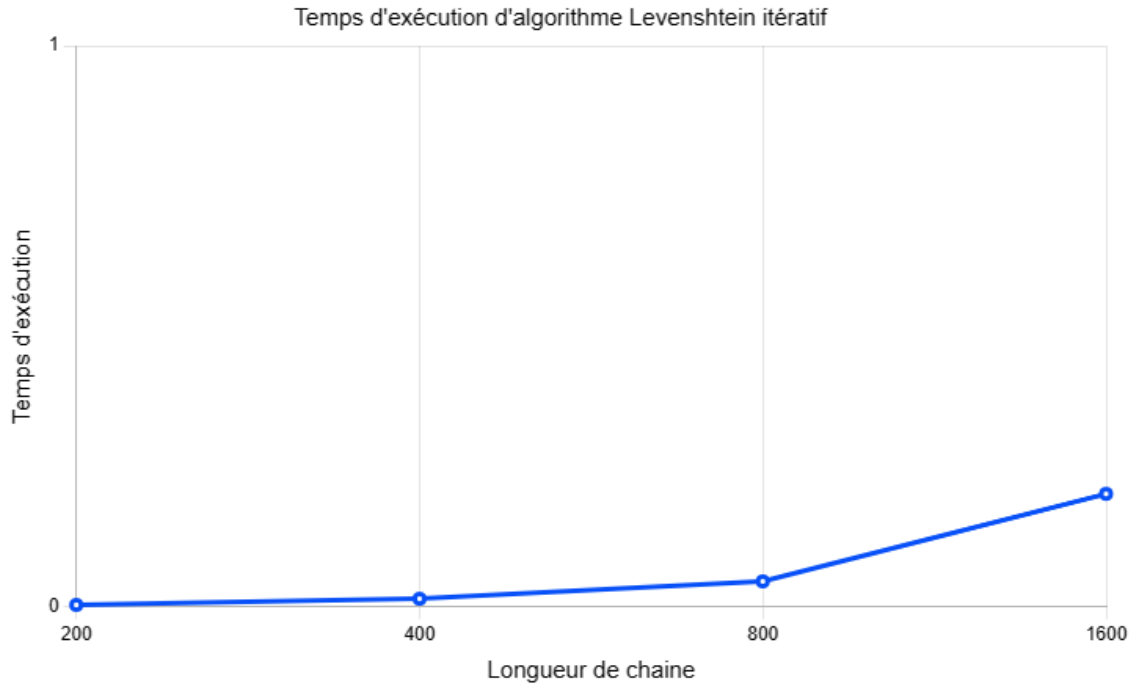
$$T(n,m) = (m+1) * (n+1)$$

6.3.5. Temps d'exécution

Le temps d'exécution est proportionnel aux longueurs n et m des deux chaînes donnée, afin de comparer le temps d'exécution de l'algorithme itératif de Levenshtein, on utilise encore la bibliothèque <chrono> pour mesurer le temps qui s'écoule avant et après l'appel à la fonction avec des chaîne de longueur différentes, on note les résultats à chaque fois et on obtient:

Longueur de la chaîne	Temps d'exécution
200	0.002993 s
400	0.013970 s
800	0.044997 s
1600	0.201017 s

Graphiquement :



⇒ On constate que cette version itérative est plus efficace et plus rapide en termes de temps de calcul, elle peut alors traiter des grandes chaînes de caractères sans problème d'overflow.

VII - Conclusion

Le but de ce projet était de comparer plusieurs algorithmes résolvant le même problème, en utilisant les différents outils vus en cours. J'ai choisi de me concentrer sur les problèmes de distance, en étudiant les algorithmes de distance de Hamming et de Levenshtein, ainsi que leurs versions itératives et récursives.

Au cours de ce projet, j'ai utilisé différentes méthodes pour comparer ces algorithmes, notamment une analyse théorique de leur complexité, ainsi que des mesures pratiques de leur performance, en utilisant des implémentations en langage C++. J'ai également examiné les limites et les avantages de chaque algorithme, en considérant des scénarios d'utilisation réels.

D'après cet étude, on s'est assuré que pour n'importe quel problème informatique il existe plusieurs solutions pour le résoudre, l'analyse et l'étude de ces solutions est très importante, ça nous permet de choisir la solution la plus performante car la première solution qui vient à l'esprit n'est pas toujours pertinente. Comme on a vu tout au long de ses étapes, chaque algorithme a des points forts, ainsi que des points faibles, mais à la fin on peut dire que la version itératif de l'algorithme de Levenshtein est la plus

conveniente grâce à sa rapidité et son avantage de pouvoir traiter deux chaînes de taille différente.

Grâce à ce projet, j'ai amélioré ma compréhension des méthodes d'analyse et d'optimisation d'algorithmes, ainsi que de la manière de les implémenter en utilisant un langage de programmation. J'ai également appris à évaluer l'efficacité des algorithmes en termes de temps d'exécution et d'utilisation de la mémoire, ainsi qu'à prendre en compte des facteurs tels que la longueur des chaînes et la structure des données. En fin de compte, ce projet m'a permis de mieux comprendre les choix qui sous-tendent la sélection et l'implémentation d'un algorithme pour résoudre un problème donné.

VIII - Webographie

Distance de Hamming : définition et explications (techno-science.net)
Complexité d'un algorithme – l'Informatique, c'est fantastique ! (blaisepascal.fr)
Line Graph Maker | Create a line chart for free
Algorithm Implementation/Strings/Levenshtein distance - Wikibooks, open books for an open world
La distance de Levenshtein : comment mesurer la similarité entre des mots ? • 10h11