**UNIVERSITE PARIS 13** **STRUCTURES DE DONNEES ET ALGORITHMES**

**INSTITUT GALILEE**

**Licence 2 – informatique**

**Semestre 1 – 2019/2020**

# LE RAPPORT :

## Implémentation de structures de données et algorithmes pour l'intersection de deux fichiers

**Réalisé par**

Meriem BOUDRISS 11808654

Reda LAMHATTAT 11809054

Betty OKO 11801530

Groupe Double Licence math-info

05/01/2020

# 2. Ensemble :

## 2.1.1. Analyse :

### *1) a) Choix d'implémentation :*

Pour implémenter l'ensemble en utilisant **les arbres binaires de recherche**, on a choisi d'utiliser une structure **Set** contenant un champ " int nb_elts" pour la taille de l'ensemble et un champ " tree arbre " pour l'ABR. La structure et les fonctions pour manipuler les ABR sont dans le même fichier TreeSet.c et les prototypes des fonctions dans tree.h.

L'utilisation des arbres binaires de recherche équilibrés nous a donné une très grande complexité en termes de temps, on a donc choisi l'implémentation avec les ABR normaux qui a une complexité en temps raisonnable.

- Implémentation des fonctions :

**Set* createEmptySet(void)** : alloue la mémoire pour un ensemble(Set) et aussi pour le champ arbre binaire et initialise les champs de la structure. Des messages d'erreur sont affichés en cas d'échec de l'allocation mémoire.

**Void freeSet(Set *set)** : libère la mémoire occupée par l'ensemble set et son champ (set->arbre).

**Size_t sizeOfSet(Set* set)** : renvoie le nombre d'éléments présents dans l'ensemble, le nombre de mots insérés.

**Insert_t insetInSet(set*set, char * element)** : insère le mot passé en argument dans l'ensemble set, vérifie d'abord si l'élément n'est pas dans set pour éviter les doublons (s'il est répété, elle renvoie la constante OLD). Si set est NULL, on alloue la mémoire pour un ensemble(en cas d'échec de l'allocation, elle renvoie ALLOC_ERROR) et on utilise la fonction itérative (insert_iter) pour insérer le mot dans le champ arbre de set. La fonction inset_iter qui prend en arguments un mot et l'arbre dans lequel on veut insérer, et parcourt l'arbre avec une boucle pour insérer au niveau des feuilles.

**Contains(const Set*set, const char* element)** : teste l'appartenance du mot element a l'ensemble set, c'est une fonction récursive qui utilise la fonction de recherche dans les ABR (search), elle utilise la propriété des ABR en cherchant l'élément soit dans la partie droite de l'arbre s'il est plus grand que la racine, et dans la partie gauche sinon.

**StringArray*setIntersection(const Set* set1, const Set* set2) :** calcule l'intersection de deux ensembles set1 et set2 en passant directement par la fonction de calcule d'intersection entre 2 ARB.

Pour implémenter l'ensemble avec **les tables de hachage** en utilisant la méthode du **chainage**, on a choisit une structure Set contenant un champ taille pour représenter le nombre des listes chaînées dans le tableau, un champ nb_mots pour le nombre total des mots dans l'ensemble, et un champ tab (liste** tab) qui est un tableau de listes chaînées. La structure et les fonctions pour manipuler les listes chaînées (création, ajout,…) sont dans le même fichier HashSet.c.

- Implémentation des fonctions :

**Set* createEmptySet(void)** : alloue la mémoire pour un ensemble(Set) et aussi pour le champ tab et initialise les champs de la structure. Des messages d'erreur sont affichés en cas d'échec de l'allocation mémoire.

**Void freeSet(Set *set)** : libère la mémoire occupée par l'ensemble set et son champ (set->tab) en passant par une boucle qui détruit toutes les listes du tableau.

**Size_t sizeOfSet(Set* set)** : renvoie le nombre d'éléments présents dans l'ensemble, le nombre de mots insérés.

**Insert_t insetInSet(set*set, char * element)** : insère le mot passé en argument dans l'ensemble set, vérifie d'abord si l'élément n'est pas dans set pour éviter les doublons (s'il est répété, elle renvoie la constante OLD). Si set est nul, alloue la mémoire pour un ensemble(en cas d'échec de l'allocation, elle renvoie ALLOC_ERROR), puis calcule le code de hachage et insère en tête de la liste correspondante a la case tab [hash].

**Contains(const Set*set, const char* element)** : teste l'appartenance du mot element a l'ensemble set, calcule son code de hachage et le cherche dans la liste correspondante dans le tableau.

**StringArray* setIntersection(const Set* set1, const Set* set2):** crée un StringArray « res » et parcourt les listes non nulles du tableau et utilise la fonction contains, si le mot est dans les 2 ensembles, elle l'insère dans res et renvoie res a la fin.

**Int h (const Set* s, const char* val):** calcule le code de hachage du mot val en utilisant le code ASCII des caractères *31*le conteur de la boucle, puis renvoie le code modulo taille du tableau.

### *b) principe de la fonction d'intersection pour les ABR :*

Dans le cas des ABR, la fonction **StringArray setIntersection(const Set* set1, const Set*set2) ;** calcule l'intersection de deux ensembles passés en arguments en appliquant la fonction **StringArray * set_intersection_tree(StringArray *arr, tree t1, tree t2)** ;aux champ **set1->arbre** et **set2->arbre.** Le pseudo-code:

**StringArray * set_intersection_tree(StringArray *arr, tree t1, tree t2){**

```
Si t1==NULL || t2==NULL
        Return arr ;
Si search(t1->key,t2)
        insertInArray(arr,t1->key) ;
set_intersection_tree(arr,t1->left,t2);
set_intersection_tree(arr,t1->right,t2);
return arr;
}
```

### *c) La complexité dans le pire des cas de la fonction **setIntersection** :*

**Le cas des ABR =>** si l'arbre dans l'ensemble B est dégénéré, on va devoir parcourir tous les éléments de B donc la complexité est **O (Na*Nb)**.

Si l'arbre n'est pas dégénéré le coût du parcourt de l'ensemble B est Hb (la hauteur de l'arbre), donc la complexité est **O (Na*Hb)**.

**Le cas des tables table de hachage=>** la complexité est **O (Na*la complexité de la fonction contains)**, la complexité de contains est **O (strlen(element)) + O(longueur de la liste dans la case du tableau correspondante au code de hachage de l'élément).**

Donc la complexité de l'intersection est égale a **O (Na* max (strlen(élément) ,L ))** où **L** est la longueur de la liste.

## *2) *) La complexité de l'insertion :*

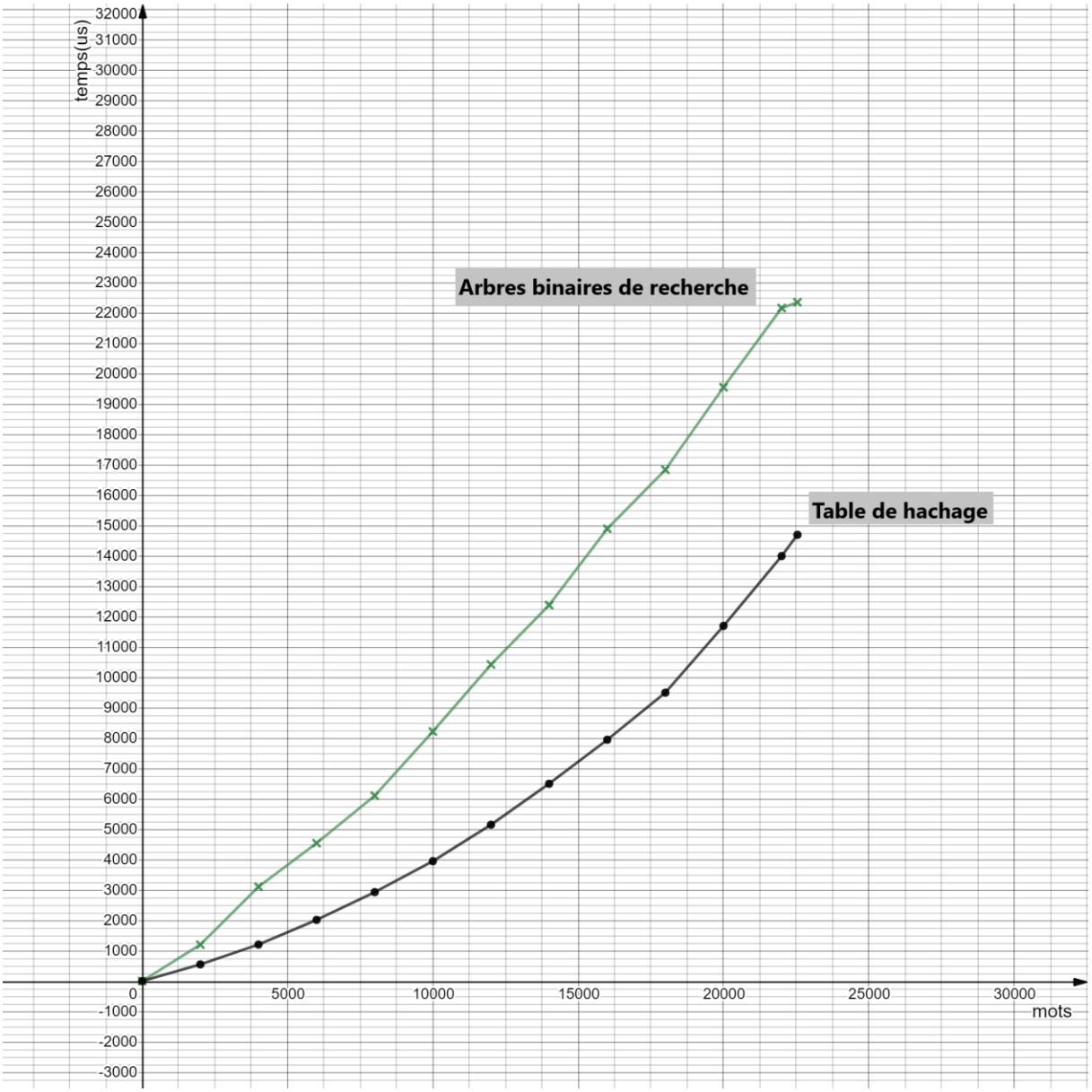
**Dans le cas des ABR : O (H(A))** où H(A) est la hauteur de l'arbre A car l'insertion se fait au niveau des feuilles. Elle est donc linéaire.

**Dans le cas des tables de hachage : O (strlen (element))** qui correspond à la complexité de la fonction qui calcule le code de hachage, l'insertion de l'élément est constante car elle se fait en tête de liste et l'accès à la case du tableau est constant. Elle est donc linéaire.

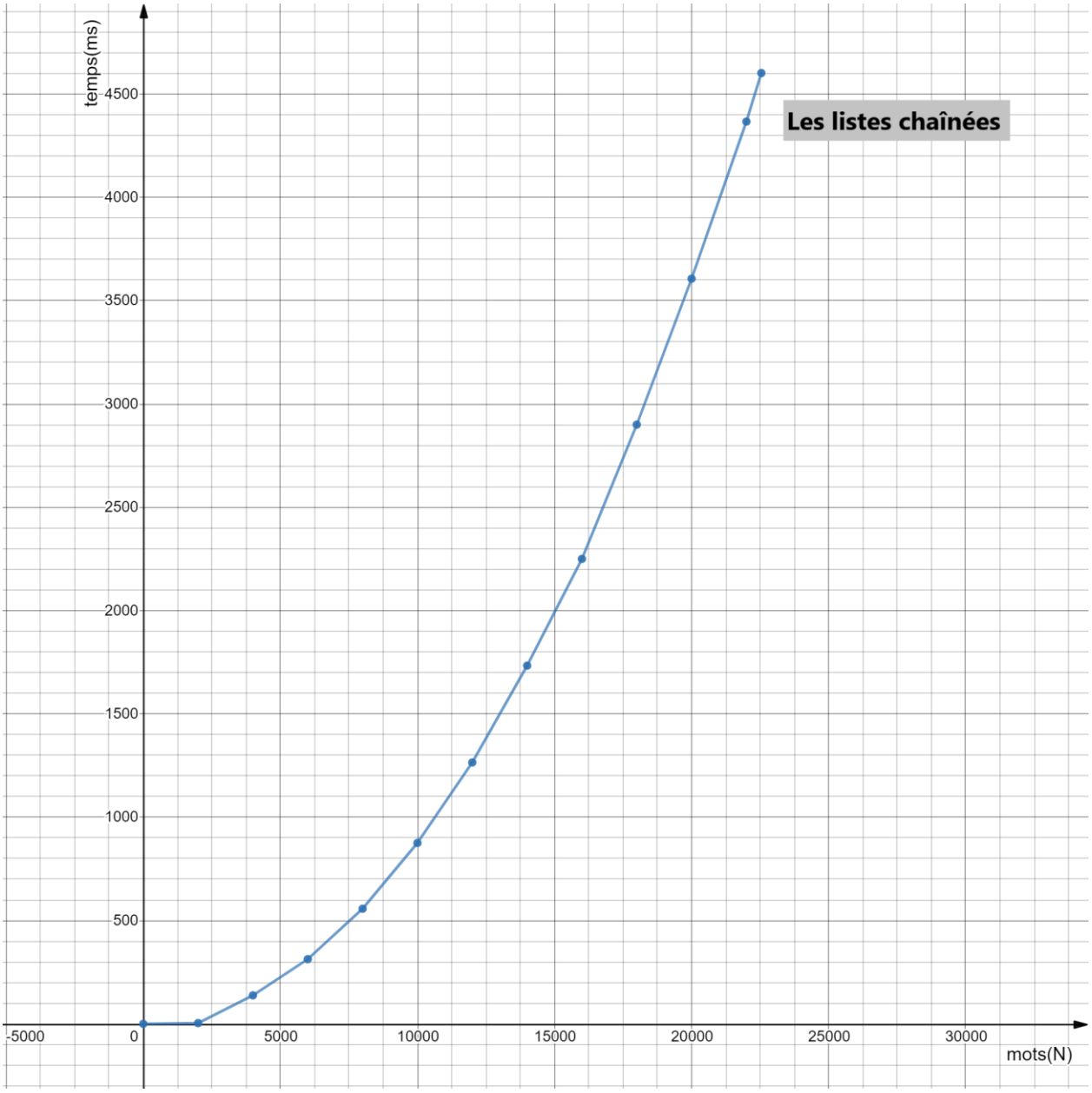
Donc la complexité pour insérer N éléments pour N croissant est **O (N*H(A))** pour les ABR et **O (N*strlen(element))** pour les tables de hachage, et en général, la hauteur de l'arbre est supérieure à la taille des chaines de caractères, cela explique le fait que

l'implémentation avec les ABR prend plus de temps, mais les deux complexités sont linéaires ce qui correspond aux résultats obtenus.

## ➔Le temps nécessaire pour insérer N valeurs dans un ARB et table de hachage. Le temps est calcule en microseconde



➔ **Le temps nécessaire pour l'insertion dans les listes chainées. Le temps est calcule en milliseconde.**




**) La complexité de la recherche :*

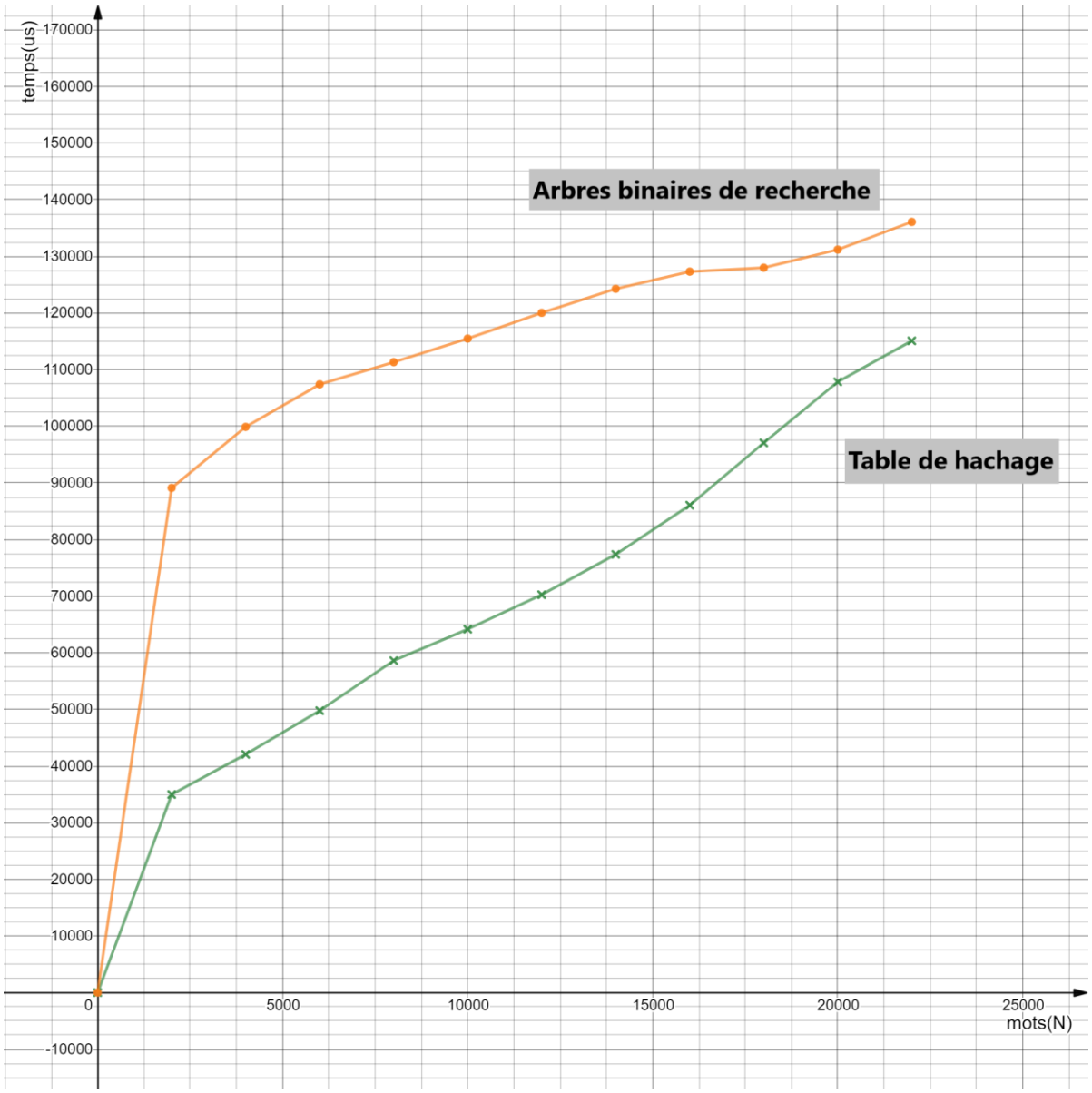
**Dans le cas des ABR : O (H(A))** où H(A) est la hauteur de l'arbre A car dans le pire des cas, l'élément que l'on cherche peut se trouver au niveau d'une feuille.

**Dans le cas des tables de hachage :** la complexité correspond au maximum entre la complexité de la fonction qui calcule le code de hachage et celle de la fonction qui

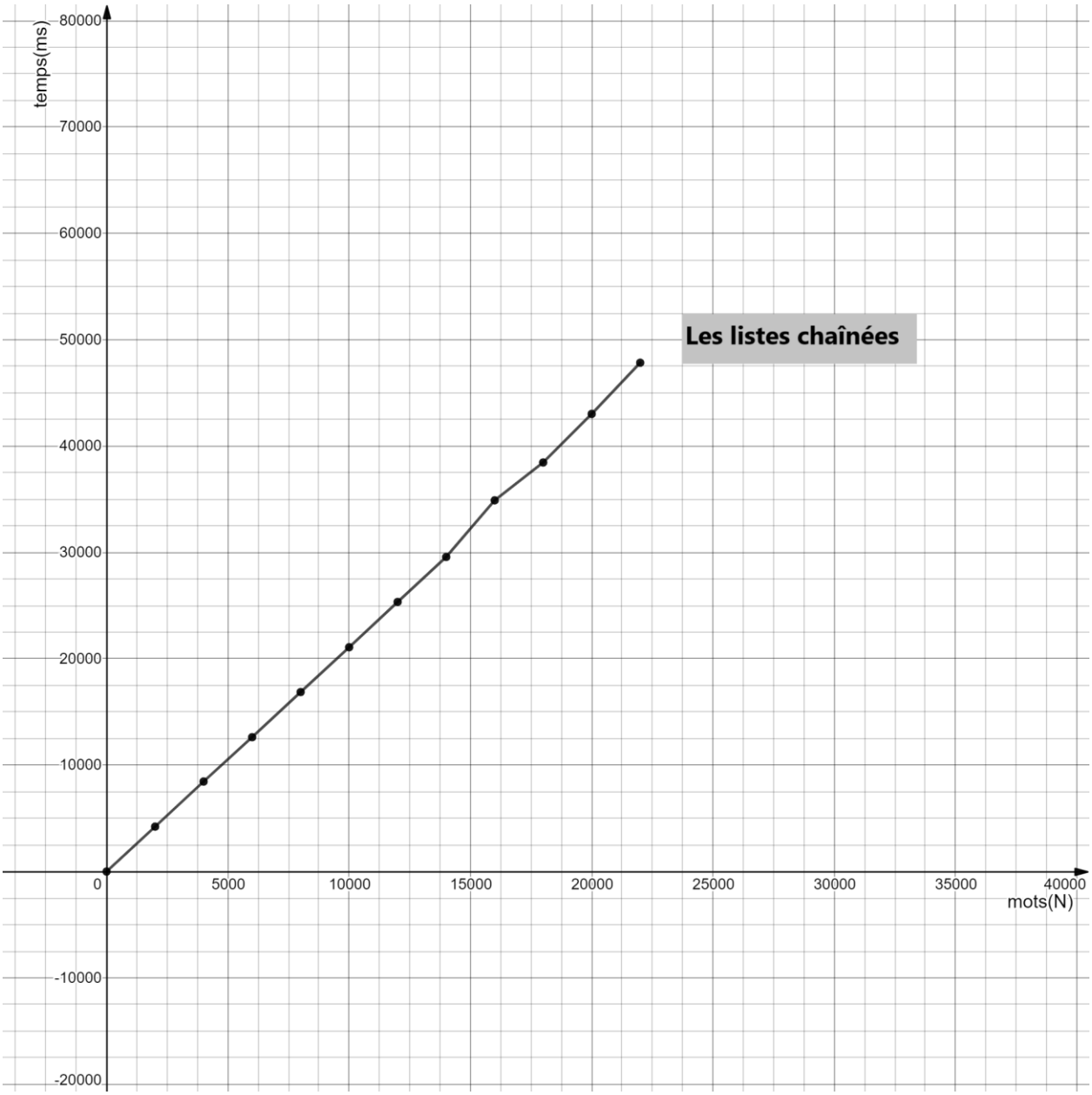
cherche l'élément dans la liste correspondante, donc la complexité est égale à **max( O(strlen(element)) ,O(L) )** , où L est la longueur de la liste qui contient l'élément, car l'élément peut se trouver a la fin de la liste .

Donc la complexité pour chercher N élément pour N croissant est égale à **O(N*H(A))** pour les ABR, et **O(N* max(strlen(element) ,L ) )** pour les tables de hachage, et la hauteur de l'arbre est supérieure à la taille des listes et à la longueur des chaines de caractères dans le cas où la table de hachage ne donne pas lieu à beaucoup de collision(la taille du tableau est grand), ce qui est le cas dans notre programme, donc l'implémentation avec les ABR est plus couteuse en temps que les tables de hachage.

⇨ **Le temps nécessaire pour la recherche dans les ABR et les tables de hachage (microseconde).**



➔ **Le temps nécessaire pour la recherche dans les listes chaînées (le temps est en milliseconde).**




## *3) comparaison des résultats :*

On remarque bien que les **listes chaînées** représentent l'implémentation la plus couteuse en temps, et plus précisément la recherche qui a une complexité quadratique, ensuite vient l'implémentation avec les **ARB** qui est moins couteuse mais cela dépend de l'arbre s'il est dégénéré alors on se retrouve dans le cas des listes chaînées, et s'il est non dégénéré, la complexité peut être optimisée **O(h)** ou même **O ($\log_2(n)$)** pour les arbres équilibrés.

L'implémentation avec **les tables de hachage** est la moins couteuse en temps, car l'insertion dépend de la fonction qui calcule le code, mais ensuite on insère l'élément en tête de la liste ce qui est constant, la recherche dépend de la taille de la liste qui correspond au code de hachage, et plus le tableau est grand et la fonction gère bien les collisions, plus la recherche est plus efficace.

# 3. Intersection de deux fichiers :

## 3.2 Analyse :

*1)* la complexité théorique minimale pour l'intersection de deux tableaux A et B de taille Na et Nb est **O (Na*Nb).**

### *2) a) La 1ère approche pour les arbres binaire de recherche :*

On stocke d'abord les éléments du tableau A dans un ARB avec une complexité de **O (Na*H(Sa)),** ou **Na** est la taille du tableau et **H(Sa)** est la hauteur de l'arbre, puis on parcourt le tableau B et on cherche ses éléments un par un dans l'arbre Sa donc la complexité est **O (Nb*H(Sa))**, donc la complexité totale est **max ( O (Na*H(Sa)), O (Nb*H(Sa)),** et vu que **Na <= Nb** alors c'est **O(Nb*H(Sa)).**

### *La 1ere approche pour les tables de hachage :*

Stockage des éléments du tableau A dans une table de hachage Sa : **O (Na*strlen(mot)).**

Parcourt du tableau B et recherche de ses éléments dans la table Sa : **O (Nb*max (strlen(mot),L)** où L est le longueur de la liste dans la case qui correspond au code de hachage.

Donc la complexité totale est **O (Nb*max (strlen (mot), L)).**

### *La 2e approche pour les ABR :*

Stockage des éléments du tableau A : **O (Na*H(Sa)).** Sa est l'arbre dans lequel on stocke A.

Stockage des éléments du tableau B : **O (Nb*H(Sb)).** Sb est l'arbre dans lequel on stocke B.

Appel a set Intersection : **O (Na*H(Sb)).**

Donc la complexité totale est : le maximum entre les trois complexités mentionnées ci-dessus.

### *La 2e approche pour les tables de hachage :*

Stockage des éléments du tableau A : **O (Na*strlen(mot)).**

Stockage des éléments du tableau B : **O (Nb*strlen(mot)).**

Appel a  set Intersection : **O (Na* max (strlen(élément) ,L  )).**

Donc la complexité totale est : le maximum entre les trois complexités mentionnées ci-dessus.

### *b) La pertinence des deux approches :*

la deuxième approche est plus pertinente car pour la recherche, on fait moins de comparaisons selon la taille de l'ensemble le plus petit (A), par contre la première nous impose le parcourt du tableau B et la comparaison de tous ses éléments avec ceux dans l'ensemble SA, de plus le fait de stocker les deux tableaux dans un ARB ou une table de Hachage minimise le nombre de comparaisons à effectuer, car la recherche dans les ABR et les table de hachage est plus efficace que les tableaux normaux**.** Concernant l'implémentation d'ensemble la plus appropriée, c'est les tables de hachage car elle donne lieu à une complexité raisonnable en temps.

## *3) Tableaux triés :*

Si les deux tableaux A et B sont triés par ordre alphabétique, on stocke les éléments du tableau A dans un ensemble Sa (table de hachage) et on cherche ces éléments par dichotomie dans le tableau B.

#### ⇨ La complexité au meilleur des cas :

Stockage des éléments de A : **O (Na*strlen (mot)).**

Recherche des éléments stockés dans la table de hachage  par dichotomie dans B : dans le meilleur des cas l'élément se trouve au milieu du tableau B donc O (1) pour chaque élément, ce qui fait **O(Na).**

Donc la complexité totale est **O(Na*strlen (mot)).**

#### ⇨ La complexité au pire des cas :

Stockage des éléments de A dans la table de hachage: **O (Na*strlen(mot)).**

Recherche de ces éléments par dichotomie dans B : dans le pire des cas l'élément se trouve a l'extrémité  du tableau B donc **O ($log_2$(Nb))** pour chaque élément, ce qui fait **O (Na*$log_2$(Nb)).**

Donc la complexité totale est **max ( O (Na*$log_2$(Nb)) , O (Na*strlen(mot)) ).**