Rayan Contuliano Bravo 24/05/2022

N° ordre = 0000537050 BA1-INFO

# Rapport Langage de programmation

## Introduction :

Dans le cadre du cours de langage de programmation, nous avons dû réaliser un projet en plusieurs parties. La première partie de ce projet consistait à écrire un programme en Python 3 qui encrypte (encrypt.py) un texte grâce à une clé avec la technique du Chiffre de Vigenère et un programme en C++ qui, quant à lui, le décrypte (decrypt.cpp) avec la même clé toujours en se référant à Vigenère. Pour la seconde partie, nous devions tout d'abord diviser notre programme C++ en plusieurs fichiers et ensuite écrire un programme en C++ qui déchiffre un texte non plus grâce à une clé mais grâce à une longueur de clé (attack.cpp), ce mécanisme est appelé l'attaque du Chiffre de Vigenère. Concernant la 3e partie (et la dernière), nous avons pour tâche de réviser et d'adapter nos programmes (decrypt.cpp et attack.cpp) afin de les rendre plus performants et plus « beaux », grâce aux compétences que nous avons acquises, pour évaluer ces performances.

Rayan Contuliano Bravo 24/05/2022

N° ordre = 0000537050 BA1-INFO

# Table des Matières :

Rayan Contuliano Bravo 24/05/2022

N° ordre = 0000537050 BA1-INFO

# I. Choix en fonctions des parties

## A. Partie 1 :

### 1. Encrypt.py :

L'approche est très simple, tout d'abord nous commençons par lire le texte ligne par ligne, on les met toutes dans une liste (chacune des lignes est une valeur) pour ensuite concaténer[1] les valeurs. Parallèlement, nous mettons leurs longueurs dans une autre liste (la première fonction renvoie donc un string et une liste de longueur). La liste de longueur va nous permettre de replacer les lignes à la bonne place dans le fichier décrypté.

Ensuite nous créons le mot de passe sur base du texte. On commence par retirer tous les espaces du texte et on le parcourt en créant un mot de passe (pour chaque caractère de A-Z, on associe une lettre du mot de passe et si on arrive à la fin de celui-ci, on retourne à son début). Subséquemment, nous parcourons le mot de passe temporaire que nous venons de créer et on lui ajoute des espaces afin de correspondre au texte initial[2]. (Pour mieux comprendre voir l'exemple a).

Nous avons donc un mot de passe et un texte qui « matchent » il ne reste plus qu'à crypter. Pour ce faire, nous parcourons le texte original et créons un nouveau texte crypté en appelant une fonction, qui, va trouver la lettre cryptée correspondante en fonction de la lettre du texte et celle du mot de passe (*associate_letter*). Cette fonction va créer la même suite de lettres (dans une liste) de la ligne de la clé dans le tableau (p.7) de Vigenère, et, grâce au décalage dans l'alphabet, de la lettre à crypter, nous trouvons la lettre cryptée correspondante. Au lieu d'utiliser une formule, nous avons préféré utiliser des listes parce que c'était plus « lisible et compréhensible » et que ça faisait bien référence au tableau.

Pour finir nous écrivons dans le bon fichier le texte crypté, que nous venons de trouver, en se servant de la liste de longueurs afin de reconstituer les mêmes lignes que dans le fichier original.

#### *a) Exemple :*

Texte que nous devons crypter → SALUT CA VA ? Mot de passe utilisé : MDP

Mot de passe correspondant →MDPMD PM DP ?

### 2. Decrypt.cpp :

Pour la partie décryptage, nous faisons plus ou moins la même implémentation qu'en python3 mais en C++. En effet, nous avons préféré ne pas se casser la tête en faisant du C++ pure.

Lorsque nous lisons le fichier texte, nous faisons la même chose qu'en Python seulement, au lieu de créer une liste des lignes pour ensuite la concaténer, nous créons tout de suite un string avec les lignes déjà concaténées (ce qui aurait été préférable de faire en Python) et nous utilisons un « vecteur d'int[3] » pour stocker les longueurs des lignes.

Pour créer un mot de passe qui « match » avec le texte, nous faisons complètement la même chose qu'en Python. Nous utilisons des « long int » pour les tailles du texte et du mot de passe que nous créons (size_t aurait peut-être été plus approprié seulement nous utilisons int parce que nous avons

[1] J'aurais juste pu concaténer les strings. Mais j'ai tendance à mettre des listes un peu partout.

[2] J'aurais pu le faire en une seule boucle, j'avais eu des problèmes à comprendre l'astuce et j'ai préféré ne pas me casser la tête ce qui a donné cette double boucle « pas belle » (ainsi que le remove des espaces)

[3] size_t aurait peut-être été plus approprié étant donné que les longueurs peuvent être très longues

utilisé une valeur négative dans la fonction (ce qui n'est pas la bonne méthode, nous aurions dû faire autrement)).

Pour le décryptage, comme on a voulu faire la même chose qu'en python3 (avec la liste), il en a résulté quelque chose de pas très clair avec plusieurs vecteurs. Nous avons d'abord un vecteur avec toutes les lettres de l'alphabet, puis, pour créer le décalage de la ligne de la clé (tableau p.7), nous utilisons un premier vecteur qui, sur base de l'index de la clé dans l'alphabet, va créer un premier vecteur de la clé jusqu'à « Z » et un deuxième de « A » jusqu'à la clé (non compris). On va ensuite concaténer les 2 vecteurs et, sur base de l'index de la lettre crypté dans le vecteur final, que nous utilisons dans le vecteur d'alphabet, nous obtenons la lettre décryptée.

L'écriture dans le fichier se fait de la même façon mais en utilisant substring pour récupérer la ligne à écrire et supprimer ce qui a déjà été écrit (au lieu d'utiliser des indexes moins compréhensibles). Cette fonction utilise la variable globale (vecteur des longueurs des lignes). En effet, comme les fonctions se trouvent dans un autre fichier, il nous est impossible de tout le temps passer en arguments le vecteur car on avait des signatures de fonctions à respecter. Le seul moyen effectif trouvé pour utiliser le vecteur de longueurs est la variable globale (nous l'avons fait même si nous savions que c'est déconseillé).

## B. Partie 2 :

### 1. Decrypt.cpp

Le fichier decrypt.cpp ne contient rien d'autre que la fonction main de decrypt.cpp de la partie1. En effet l'ensemble des autres fonctions dont a besoin ce fichier se trouvent dans le fichier vigenere.cpp, car nous devions diviser nos programmes de la sorte. Decrypt.cpp fait donc la même chose que lors de la partie 1 sauf que les fonctions qu'elle utilise se trouvent dans un autre fichier.

### 2. Attack.cpp

Le fichier attack.cpp est similaire au fichier decrypt.cpp de la partie 2. Seulement, au lieu d'appeler la fonction « *write_file* », il appelle la fonction « *attack* » pour commencer l'attaque du chiffre de Vigenère. Cette dernière va chercher les potentielles clés, pour déchiffrer le texte, via la fonction « *trouver_candidat »*[4] (qui va se charger de diviser le texte en colonnes et de prendre la lettre la plus commune dans chaque colonne pour créer une clé potentielle). Une fois avoir trouvé une clé potentielle, elle compare les fréquences d'apparition de E dans le texte décrypté par chacune des 2 clés et prend la fréquence la plus proche de 17.115%. En effet, il s'avère que nous avions mal interprété les consignes et nous pensions que pour chaque clé, il fallait décrypter le texte et calculer la fréquence de E alors qu'en fait, il ne fallait que comparer les erreurs des clés. Le fait de décrypter à chaque fois rallonge fortement le temps d'exécution.

Une fois le bon mot de passe trouvé, il ne nous reste plus qu'à décrypter le texte et de l'écrire dans le fichier (avec la fonction « *decode »* qui va faire tous les appels nécessaires)[5]

[4] Elle est assez claire à comprendre, je ne l'expliquerais pas. Divise le texte en L colonnes, prend la lettre la plus commune de chaque -> mot de passe candidat trouvé.

[5] Même mécanisme que pour decrypt.cpp : associate_pw -> decrypt -> write_file

### 3. Vigenere.cpp[6]

Le fichier vigenere.cpp, lui, contient l'ensemble des fonctions utilisé par les fichiers attack.cpp et decrypt.cpp.

# II. Changements concernant la 3e partie

## A. Utilisation de classe :

Comme demandé dans les consignes, nous avons implémenté une classe. Nous avons décidé d'en faire une pour gérer le contenu des fichiers. En effet nous avons créé une classe File qui va avoir comme attributs, le nom du fichier crypté, le nom du fichier décrypté, le contenu du fichier crypté et le vecteur des tailles des lignes du fichier crypté. La classe aura une méthode pour lire le fichier, une pour diviser le texte en L longueur et une autre qui va décoder et écrire dans le fichier décrypté.

De ce fait, je n'aurais plus besoin d'une variable globale pour le vecteur des tailles. Car elle est en attribut de classe. Plus besoin également de copies du texte crypté car il est également en attribut de classe.

## B. Modification des signatures obligatoire de la partie 2 :

Pour pouvoir utiliser ma classe, j'ai modifié les signatures des fonctions « *trouver_candidats* » et « *decode* ».

Au lieu de prendre le texte (cypher) en argument, elle prend une instance de ma classe File qui contient tout ce que j'ai dit au point A. Grâce à ça on pourrait facilement avoir accès au contenu du fichier crypté et à la méthode pour diviser le texte en L colonnes.

La fonction « *decode* » ne sera plus une fonction normale mais une méthode de ma classe File et ne prendra en arguments qu'un pointeur vers une structure clé. Elle décryptera les lettres une par une et les écrira dans le bon fichier de la même façon.

## C. Autres modifications :

Lors de nos tentatives d'optimisation, nous avons supprimé beaucoup de fonctions. Même si nous trouvions que les fonctions se partageaient bien les tâches nous avions remarqué qu'elles parcouraient plusieurs fois chacune le contenu du fichier crypté, ce qui n'est pas très optimal pour des grandes valeurs.

J'ai donc réuni les fonctions « *associate_pw* », « *associate_letter* », « *decrypt* », et « *write_file*[7] » en une seule méthode de ma classe File, « *decode* ». Cette méthode ouvre le fichier décrypté et parcourt le contenu du fichier crypté pour associer à chaque lettre du texte original une lettre du mot de passe pour ensuite utiliser une formule[8] (voir annexe E) pour décrypter et écrire le résultat dans le fichier. Grâce à ça on ne parcourt plus qu'une seule fois en entier le contenu du fichier crypté. De plus, tous les arguments (ou presque) sont passés par référence afin de ne pas refaire une copie inutile (qui pourrait remplir la mémoire) à chaque appel d'une fonction. Nous avons également remplacé toutes

[6] Je ne parlerais pas du fichier vigenere.hpp car il ne contient que les déclarations des fonctions du fichier vigenere.cpp

[7] Car elles parcouraient toutes le contenu du fichier crypté

[8] Au lieu d'utiliser la fonction decrypt et associate_letter

les variables de type *size_t* par *uint_fast32_t* car c'est apparemment fait pour être utilisé des programmes où l'optimisation est importante.

Ensuite, dans la fonction « *attack* », nous avons remarqué que plus la longueur du mot de passe que l'on veut trouver est grande, plus le mot de passe trouvé correspondra à une succession du vrai mot de passe. Imaginons qu'on a crypté un texte avec la clé « MDP », si on veut trouver une clé dont la longueur est maximum 10, alors la clé trouvée par « *attack* » sera par exemple MDPMDPMDP. Donc pour optimiser la recherche, il suffit qu'à partir d'une certaine longueur on vérifie si la clé trouvée est une rotation de notre clé actuelle. Si c'est le cas alors on a trouvé la clé de base et on peut s'arrêter d'attaquer le chiffre de Vigenère[9].

Pour finir, nous avons également ajouté le flag -O3 qui va optimiser la compilation du programme.

# III. Résultats des évaluations

## A. Avant les optimisations :

Tous les schémas seront dans l'annexe B.

Vous pouvez voir sur les différents graphiques que le temps d'exécution (en seconde) est clairement exponentiel (et prend beaucoup de temps) par rapport à la longueur L. Pour les fichiers 7 et 8 nous n'avons même pas pu faire toutes les longueurs car ça aurait pris trop de temps. Rien que pour le fichier 6 avec un L = 16384, ça prenait plus de 1300 secondes (23 minutes). De plus, les écarts-types sont aussi médiocres car il peut y avoir jusqu'à +-600 secondes de différence entre les temps d'exécution pour un même fichier (ce qui n'est évidemment pas ce que l'on recherche). C'est en effet dû aux faits que pour chaque clé potentielle, nous décryptons le texte pour calculer la fréquence de E, ce qui allonge grandement l'exécution du programme. De plus, nous parcourons plusieurs fois le contenu du fichier crypté alors que nous pouvons le faire en une seule fois. A ce moment-là je me suis rendu compte que mon code n'était vraiment pas optimal.

## B. Après les optimisations :

Tous les schémas seront dans l'annexe C.

Via les schémas nous pouvons nous rendre compte que les temps d'exécution ont fortement diminués (nous avons pu d'ailleurs faire tous les tests cette fois-ci). Justement parce que nous parcourons moins de fois le contenu du fichier crypté et nous utilisons beaucoup moins de stockage. Le temps d'exécution entre les fichiers est plus ou moins constant car il ne dépend plus de la longueur L que l'on recherche mais du fait qu'on a trouvé, ou pas, la bonne clé. L'écart-type à lui aussi été diminué et est plus constant car le maximum qu'il y a eu lors des exécutions est de 2 secondes.

Si cette optimisation (celle de trouver la bonne clé et ensuite d'arrêter de chercher une clé) va à l'encontre des consignes ou alors est trop « cheaty », il suffirait de retirer la 2e condition dans la fonction « *attack* ». Cependant les temps d'exécution seraient plus longs alors que c'est « inutile ».

[9] Je suis concient que si la clé avec lesquelles on a crypté le texte ont une grande longueur, cela ne sera pas du tt une optimisation (car on devra vérifier (presque) à chaque fois à partir d'une certaine longueur si c'est une roation) car ça rendra plus de temps. Seulement, j'ai pensé que dans notre cas, c'était approprié

# IV. Justifications des parties coûteuses

Les parties les plus coûteuses de attack.cpp sont les fonctions où on doit traiter le texte crypté en entier. Par exemple lorsqu'on doit diviser le texte, ou trouver la lettre la plus fréquente. Cependant nous sommes bien obligés de le faire de cette façon.

Concernant la méthode « *divide_text* », sa complexité est en $\theta(n)\ t.q\ n = longueur\ du\ texte$. Etant donné qu'on parcourt une seule fois le texte et qu'on répartit instantanément le texte en L colonnes nous n'avons pas besoin de parcourir plusieurs fois le texte pour créer les différentes colonnes. Pour un long texte ça peut être assez long cependant nous ne voyons pas d'autres solutions.

Pour la fonction « *findMostOccurence* », sa complexité est en $O(n)\ t.q\ n = longueur\ du\ string$ sachant qu'on doit l'appeler sur toutes les colonnes du texte crypté ce qui a la longue va prendre un certain temps.

Comme nous le montre le profiler gprof (voir annexe D), la fonction « *divide_text* » occupe 68% du temps d'exécution et 31% pour la fonction « *findMostOccurence* ». Ce sont donc bien les parties du programme les plus coûteuses (en temps).

Selon moi, ce sont les seules parties coûteuses de notre programme. Car nous ne stockons pas de grandes valeurs (juste pour le vecteur de colonnes et le contenu de texte). Nous n'utilisons plus des strings pour stocker les phases de création d'un mot de passe étant donné que nous utilisons les indices du texte et du mot de passe pour ensuite écrire le résultat, par la formule annexe E, dans le fichier décrypté (donc plus de strings pour stocker le texte décrypté).

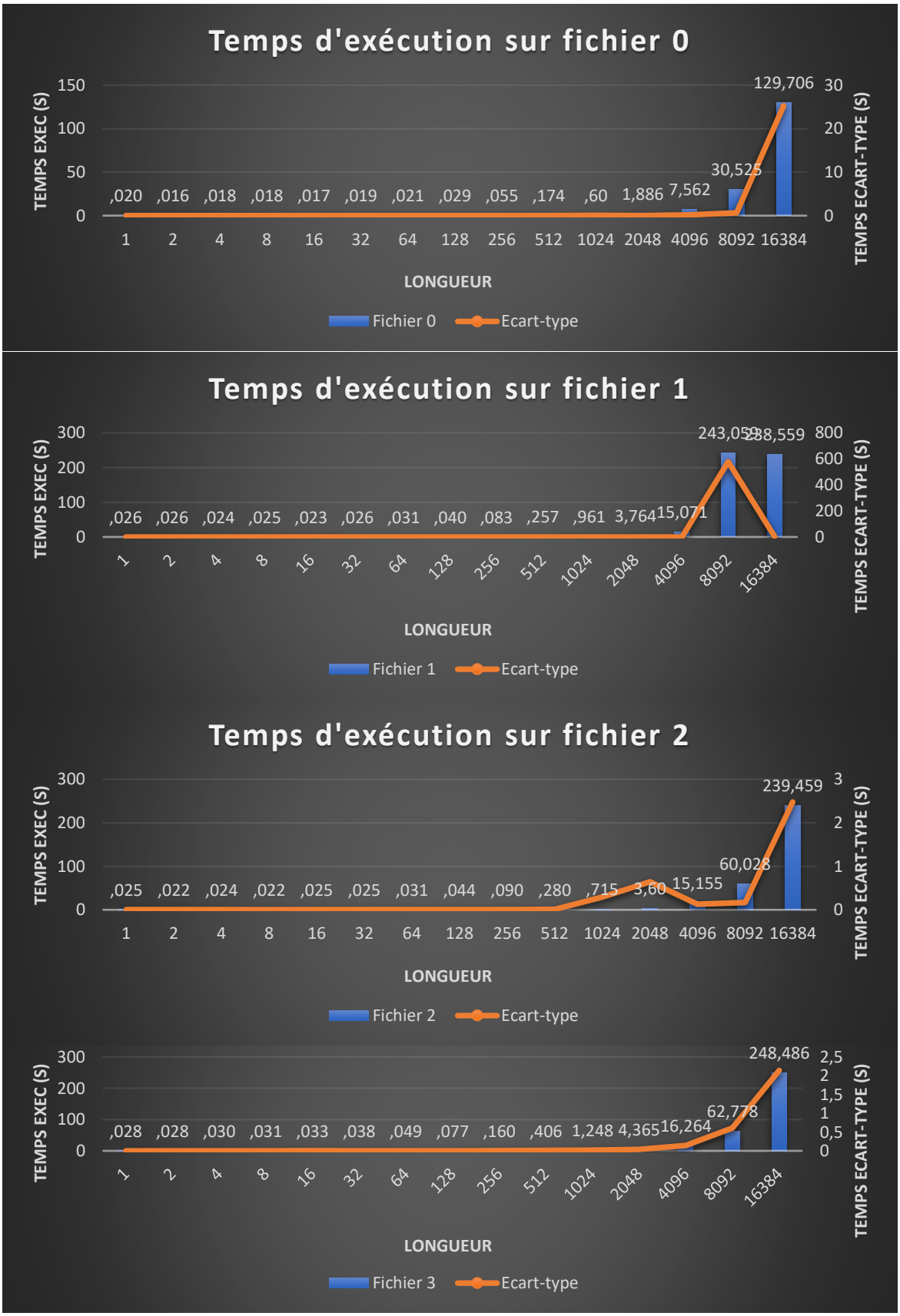
# V. Annexes

## A. Figure 1 :

lettre en clair

| lettre de la clé | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| B | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A |
| C | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B |
| D | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C |
| E | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D |
| F | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E |
| G | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F |
| H | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G |
| I | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H |
| J | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I |
| K | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J |
| L | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K |
| M | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L |
| N | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M |
| O | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N |
| P | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| Q | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P |
| R | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q |
| S | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R |
| T | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S |
| U | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T |
| V | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U |
| W | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V |
| X | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W |
| Y | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X |
| Z | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y |

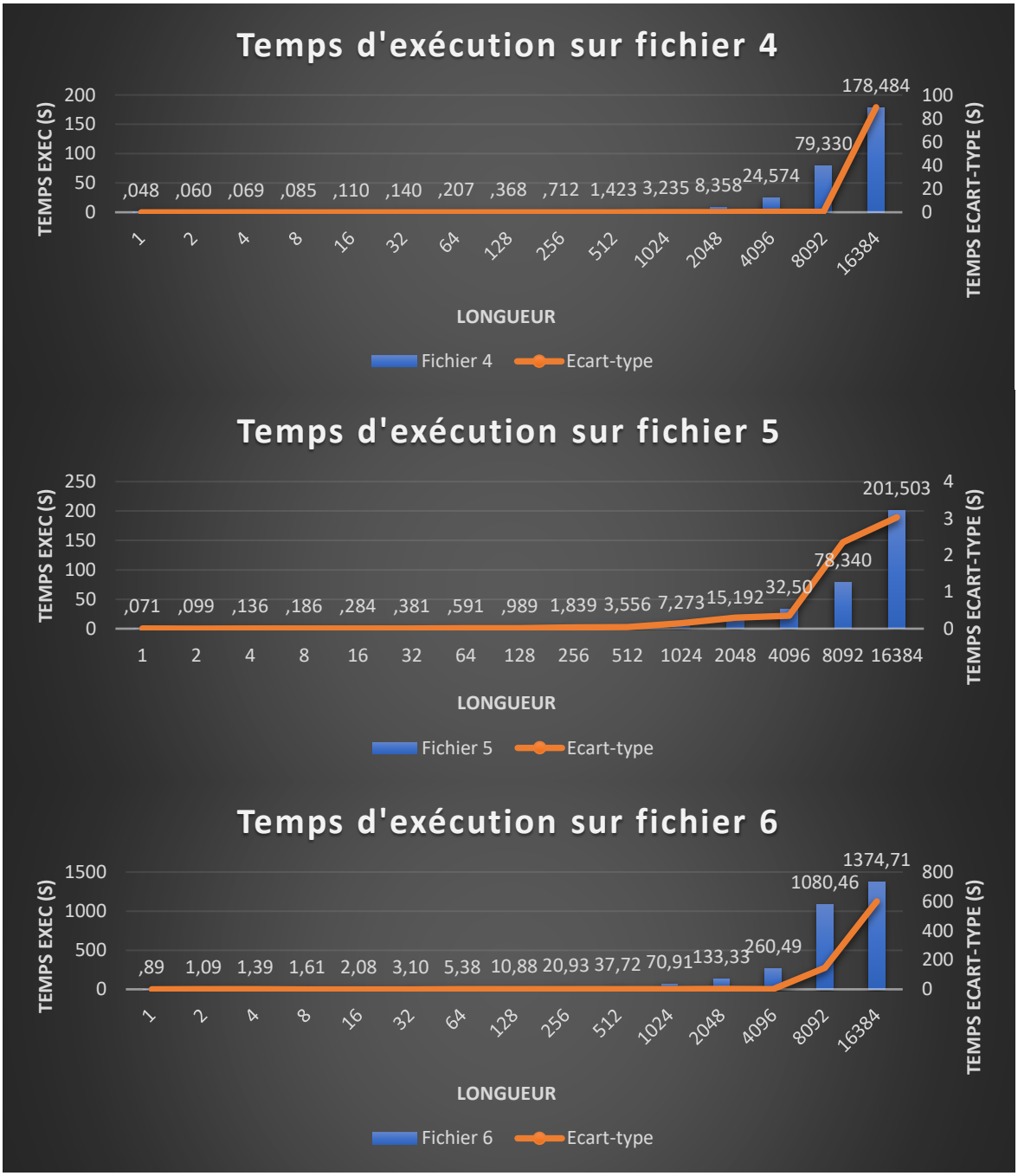
## B. Figure 2 : [10]




[10] Note : Les nombre à l'intérieur du graphique appartiennent aux bâtonnets et non à la courbe.

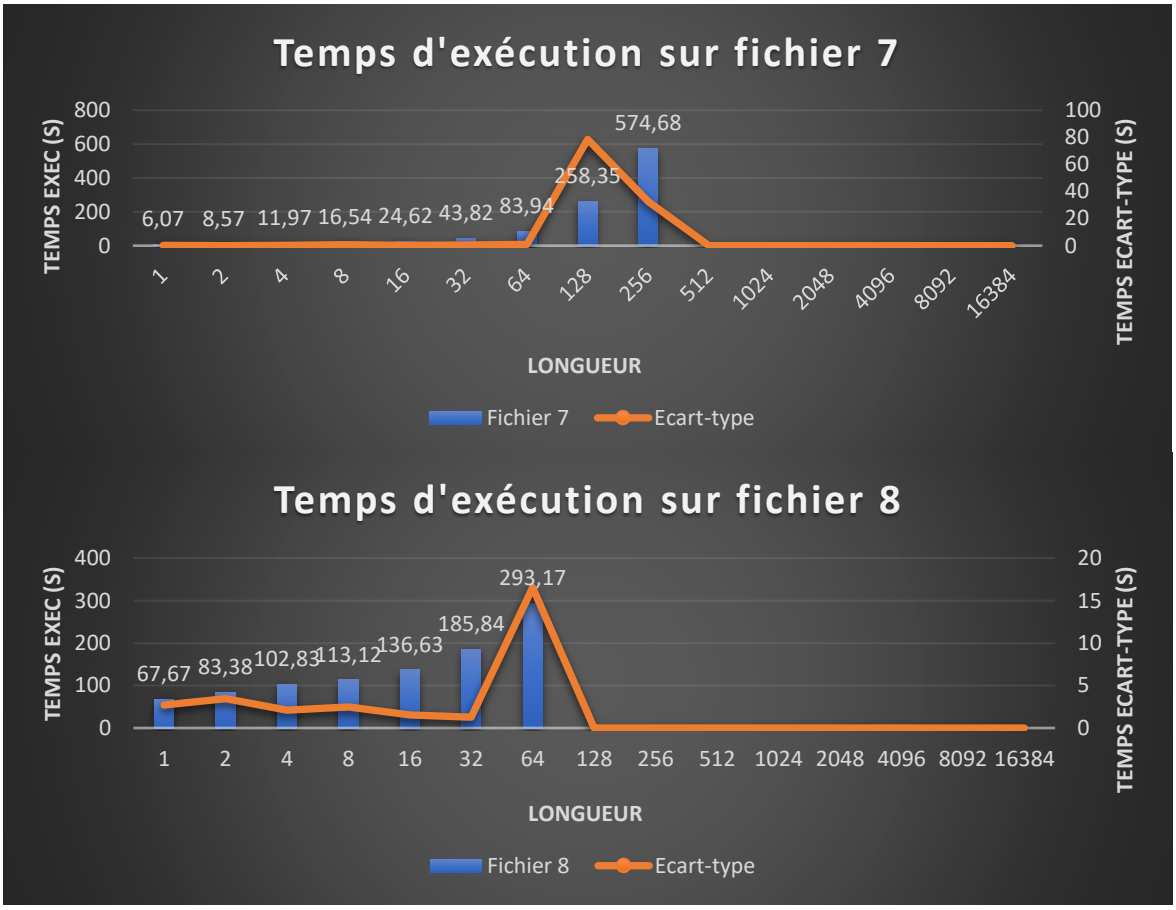
Rayan Contuliano Bravo 24/05/2022

N° ordre = 0000537050 BA1-INFO

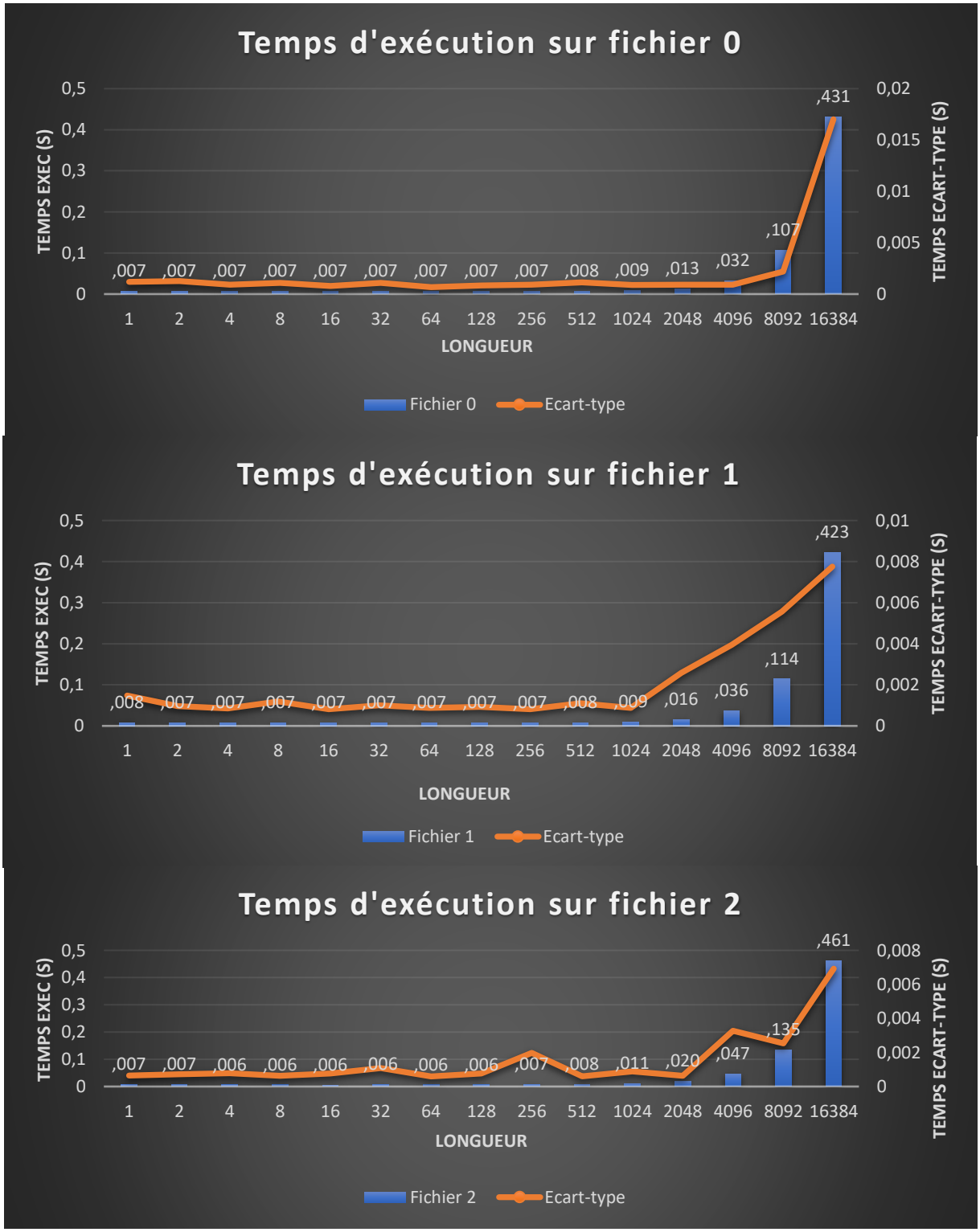


Rayan Contuliano Bravo 24/05/2022

N° ordre = 0000537050 BA1-INFO

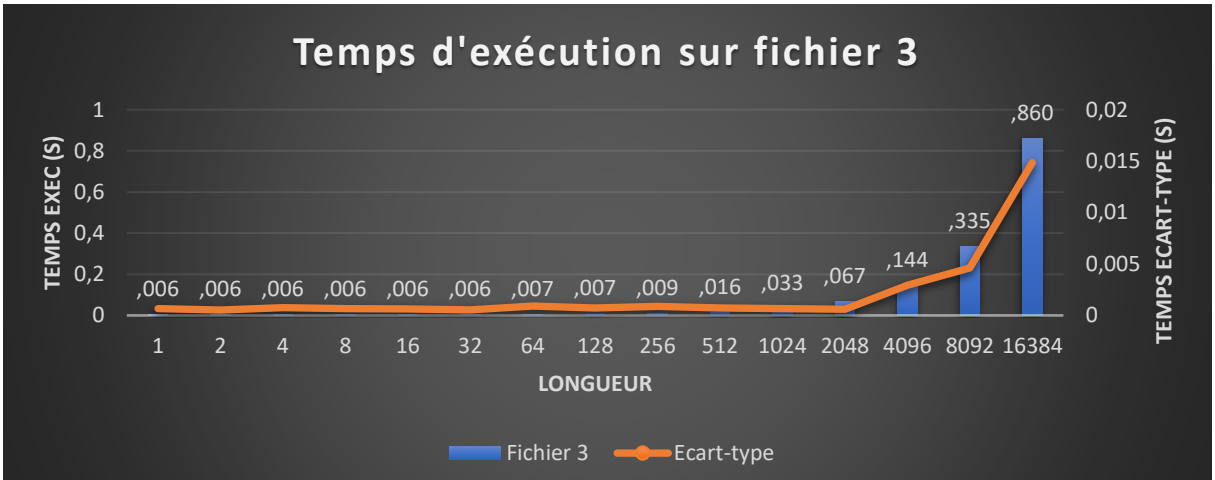


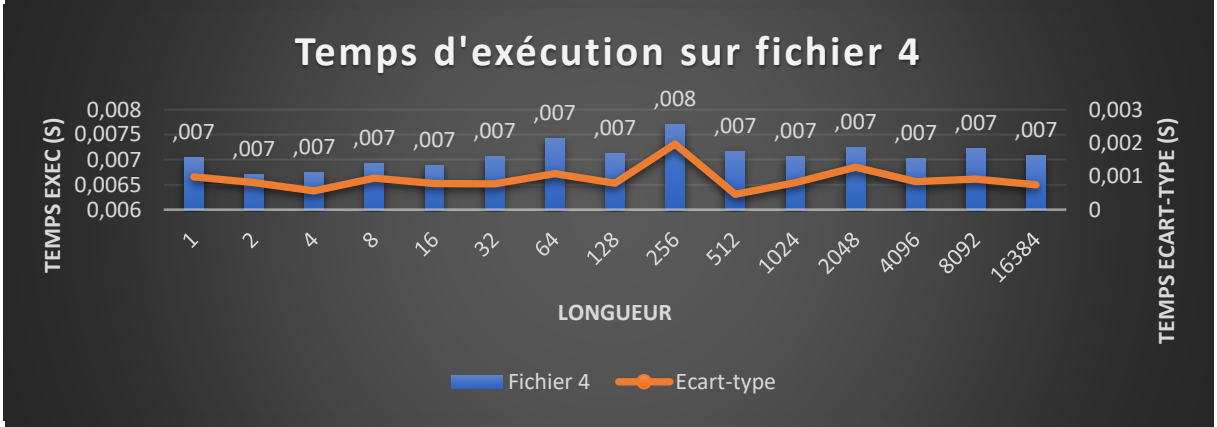
## C. Figure 3 :

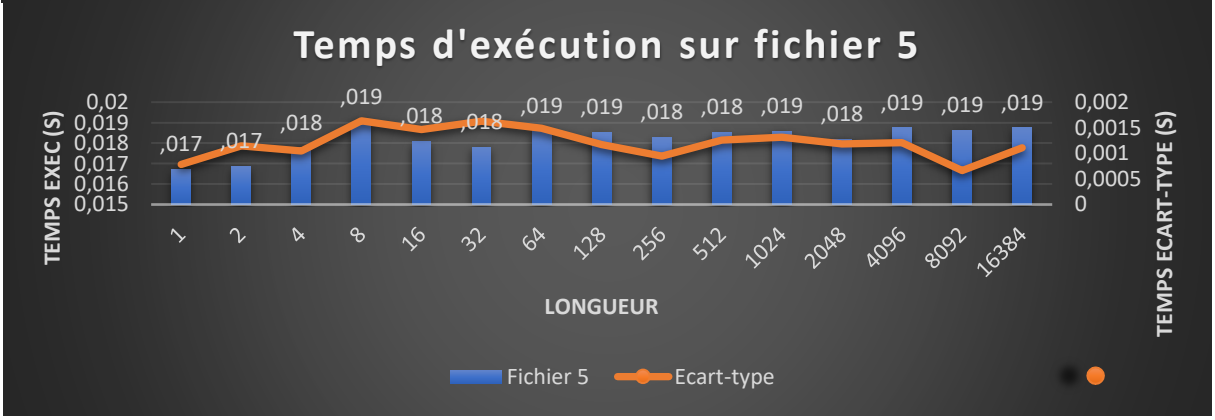

Temps d'exécution sur fichier 0
TEMPS EXEC (S)
TEMPS ECART-TYPE (S)
LONGUEUR
1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8092 16384
,007 ,007 ,007 ,007 ,007 ,007 ,007 ,007 ,007 ,008 ,009 ,013 ,032 ,107 ,431
Fichier 0
Ecart-type
Temps d'exécution sur fichier 1
TEMPS EXEC (S)
TEMPS ECART-TYPE (S)
LONGUEUR
1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8092 16384
,008 ,007 ,007 ,007 ,007 ,007 ,007 ,007 ,007 ,008 ,009 ,016 ,036 ,114 ,423
Fichier 1
Ecart-type
Temps d'exécution sur fichier 2
TEMPS EXEC (S)
TEMPS ECART-TYPE (S)
LONGUEUR
1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8092 16384
,007 ,007 ,006 ,006 ,006 ,006 ,006 ,006 ,007 ,008 ,011 ,020 ,047 ,135 ,461
Fichier 2
Ecart-type

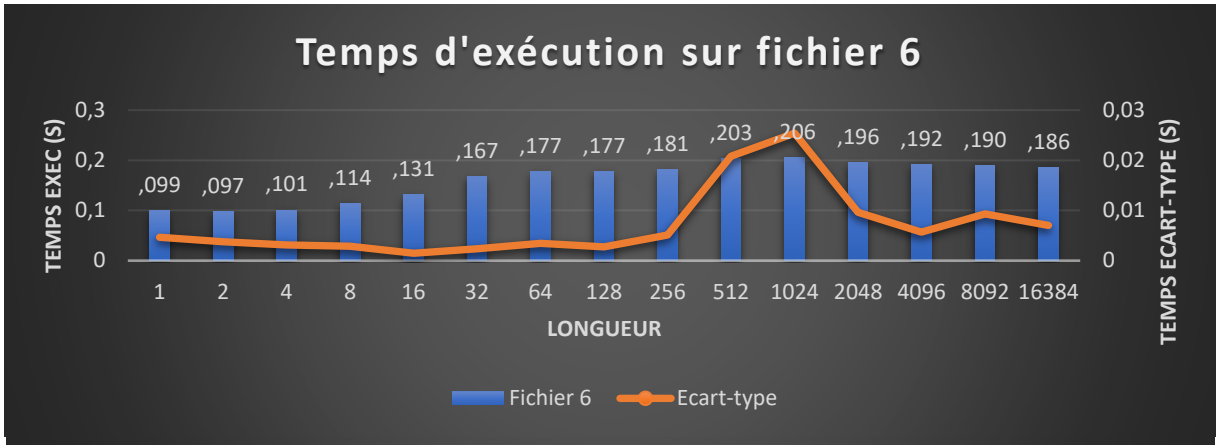
Rayan Contuliano Bravo 24/05/2022

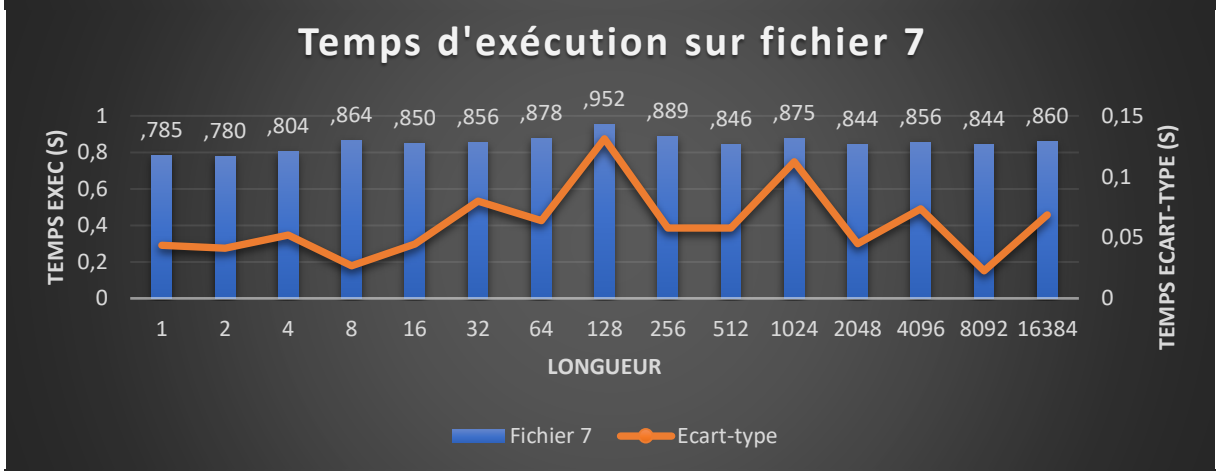
N° ordre = 0000537050 BA1-INFO

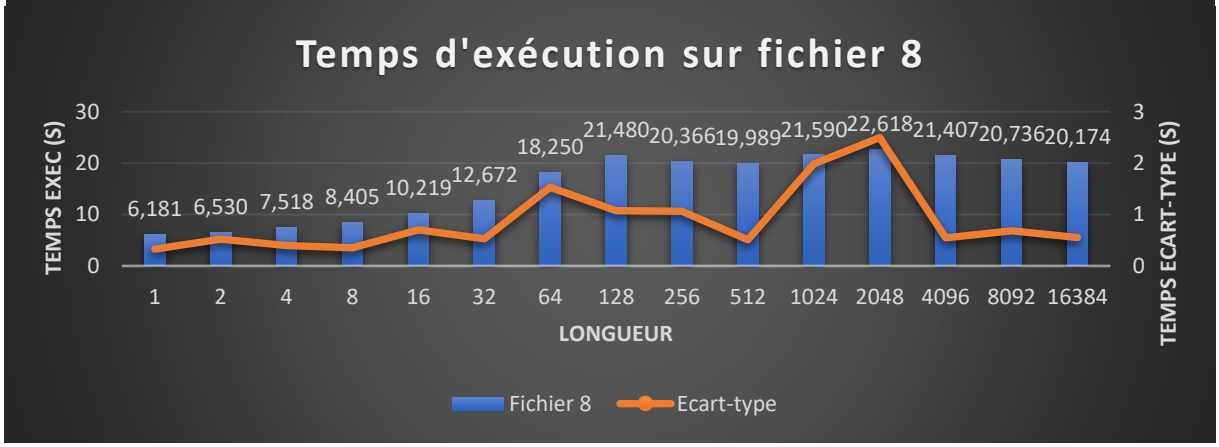





## D. Figure 4 :

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 68.38      9.04     9.04       87   103.91   103.91  File::divideText[abi:cxx11](unsigned long const&) const
 30.90     13.12     4.08     3828     1.07     1.07  findMostOccurence(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >&)
```

## E. Figure 5 :

```
char((((letter - clef->clef[idxMdp]) + 26) % 26) + 'A');
```