|  |  |
| --- | --- |
|  | **SIGLE DU COURS** |
| Titre du cours |
| ***Série A, B, C ou D, etc., s’il y a lieu*** |

**Travail noté**

Titre du travail (Pondération)

|  |
| --- |
| ■ Remplissez soigneusement cette feuille d’identité.  ■ Rédigez votre travail, en commençant à la page suivante.  ■ Sauvegardez votre travail de cette façon : SIGLEDUCOURS\_TN1\_VOTRENOM.  ■ Utilisez le *Dépôt des travaux* pour acheminer votre travail à votre professeur ou son délégué. <http://www.teluq.ca/mateluq/> |

Feuille d’identité

Nom : Romero Francia Prénom :Gonzalo Alfredo

Numéro d’étudiant : 21394980 Trimestre : Hiver 2024

Courriel : [romero\_francia.gonzalo\_alfredo@univ.teluq.ca](mailto:romero_francia.gonzalo_alfredo@univ.teluq.ca)

Nom du professeur ou son délégué : Habib Louafi

**Réservé à l’usage du professeur ou son délégué**

Date de réception       Date de retour

Note

Date d’envoi

**Tables des matières**

Introduction3

Planification de la réalisation du projet de fin d’études nommé Implémentation et comparaison des algorithmes de chiffrage asymétrique – Continuation de la réalisation de ce projet de fin d’études4

Description du problème principal, qui devient aussi l’objectif principal de la réalisation de la phase 5, du projet de fin d’études nommé « Implémentation et comparaison des algorithmes de chiffrage asymétrique »……………………………………………………………………………………………………………....5

Modélisation de la solution au problème principal, qui devient aussi l’objectif principal de la réalisation de la phase 5, du projet de fin d’études nommé « Implémentation et comparaison des algorithmes de chiffrage asymétrique »……………………………………………………………………………………………………………....6

Phase 5 – Explication de l’approche de programmation utilisé, pour rendre le code python associé à l’Algorithme RSA, paramétrable, distribuable, corrigé, amélioré et optimisé7

Explication de l’approche de programmation utilisé, pour rendre le code python de base associé à l’Algorithme RSA, en un code python corrigé, amélioré et optimisé7

Comparaison du code python de base implémentant l’algorithme RSA, montré dans le travail noté #1 d’INF 1430, avec le code python corrigé, amélioré et optimisé, implémentant cet algorithme, qui est en version 18

Comparaison du code python implémentant l’algorithme RSA, corrigé, amélioré et optimisé, qui est en version 1, avec le code python implémentant l’algorithme RSA, corrigé, amélioré et optimisé, qui est en version 2.24

Comparaison du code python implémentant l’algorithme RSA, corrigé, amélioré et optimisé, qui est en version 2, avec le code python implémentant l’algorithme RSA, corrigé, amélioré et optimisé, qui est en version 343

Comparaison du code python de base implémentant l’algorithme El-Gamal, montré dans le travail noté #1 d’INF 1430, avec le code python corrigé, amélioré et optimisé, implémentant cet algorithme, qui est en version 1.66

Comparaison du code python implémentant l’algorithme El-Gamal, corrigé, amélioré et optimisé, qui est en version 1, avec le code python implémentant l’algorithme El-Gamal, corrigé, amélioré et optimisé, qui est en version 2. 85

Comparaison du code python implémentant l’algorithme El-Gamal, corrigé, amélioré et optimisé, qui est en version 2, avec le code python implémentant l’algorithme El-Gamal, corrigé, amélioré et optimisé, qui est en version 3. 99

Comparaison du code python implémentant l’algorithme El-Gamal, corrigé, amélioré et optimisé, qui est en version 3, avec le code python implémentant l’algorithme El-Gamal, corrigé, amélioré et optimisé, qui est en version 4. 116

Comparaison du code python implémentant l’algorithme El-Gamal, corrigé, amélioré et optimisé, qui est en version 4, avec le code python implémentant l’algorithme El-Gamal, corrigé, amélioré et optimisé, qui est en version 5. 130

Comparaison du code python implémentant l’algorithme El-Gamal, corrigé, amélioré et optimisé, qui est en version 5, avec le code python implémentant l’algorithme El-Gamal, corrigé, amélioré et optimisé, qui est en version 6. 149

Comparaison du code python de base implémentant l’algorithme ECC, montré dans le travail noté #1 d’INF 1430, avec le code python corrigé, amélioré et optimisé, implémentant cet algorithme, qui est en version 1. 173

Comparaison du code python implémentant l’algorithme ECC, corrigé, amélioré et optimisé, qui est en version 1, avec le code python implémentant l’algorithme ECC, corrigé, amélioré et optimisé, qui est en version 2. 183

Comparaison du code python implémentant l’algorithme ECC, corrigé, amélioré et optimisé, qui est en version 2, avec le code python implémentant l’algorithme ECC, corrigé, amélioré et optimisé, qui est en version 3. 195

Comparaison du code python implémentant l’algorithme ECC, corrigé, amélioré et optimisé, qui est en version 3, avec le code python implémentant l’algorithme ECC, corrigé, amélioré et optimisé, qui est en version 4. 207

Comparaison du code python implémentant l’algorithme ECC, corrigé, amélioré et optimisé, qui est en version 4, avec le code python implémentant l’algorithme ECC, corrigé, amélioré et optimisé, qui est en version 5. 225

Conclusion……………………………………………………………………………………………………………….247

**Introduction**

Le présent travail noté vise à montrer, de manière détaillé, structurée et organisée, toutes les approches et techniques de programmation et toutes les modélisations des calculs mathématiques et arithmétiques et informatiques nécessaires, pour implémenter, en langage python 3.12.1, les 10 critères d’évaluation décrites à la page 7 du document Word du travail noté #2 d’INF 1430, et qui sont les suivants :

* Temps d'exécution.
* Utilisation de la mémoire.
* Taille des clés générées.
* Complexité algorithmique.
* Gestion des grandes quantités de données.
* Comparaison des tailles de clés.
* Comparaison des temps d'exécution avec différentes tailles de clés.
* Quantité de mémoire utilisée lors de la génération de clés.
* Comparaison de la taille des messages clairs et pleins utilisés, pour bien sécuriser les messages cryptés et décryptés
* L'effet avalanche.

Gonzalo Alfredo Romero Francia, l’auteur du présent document, a décidé de prendre, comme base et point de départ, pour réaliser toutes les tâches de conception, programmation, développement, test de fonctionnement et amélioration continue des codes python produites, les codes python RSA, El-Gamal et ECC, en version 3, 6 et 5 respectivement, fournis par la section Annexe du travail noté #2 d’INF 1430, ces trois codes python RSA, El-Gamal et ECC sont déjà codés et testés lors de la réalisation des tâches d’optimisation profonde et exhaustive des codes python RSA, El-Gamal et ECC fournis par ce travail noté #2 d’INF 1430, ces codes python sont déjà fonctionnels, paramétrables et distribuables, corrigés, améliorés et optimisés au complet, ce fait est confirmé par tous les test de fonctionnement faits lors de la réalisation du travail noté #2 d’INF 1430.

Maintenant, pour réaliser le présent travail noté #3 d’INF 1430, et pour continuer la réalisation du projet de fin d’études nommé «  Implémentation et comparaison des algorithmes de chiffrage asymétrique », Gonzalo Alfredo Romero Francia a décidé, tout en réalisant plusieurs recherches d’informations, sur les types de critères d’évaluation utilisés en général, pour évaluer la performance et l’optimisation de codes python programmés pour implémenter les algorithmes RSA, El-Gamal et ECC, de choisir les 10 critères d’évaluation de ses trois codes python implémentant les algorithmes RSA, El-Gamal et ECC, qui sont montrés juste ci-haut dans la présente page du présent document, pour les implémenter aussi en langage python de manière très efficace et très optimale, dans le but d’évaluer la performance, l’efficacité et l’optimisation de toutes les lignes de code python de ces trois codes python RSA, El-Gamal et ECC, afin de réaliser les analyses profondes et exhaustives qui seront réalisées lors de la réalisation du travail noté #4 d’INF 1430.

Maintenant que l’auteur du présent document a déterminé, lors de la réalisation du travail noté #1 d’INF 1430, les 10 critères d’évaluation de ses trois codes python RSA, El-Gamal et ECC, maintenant il est très important de noter et de remarquer, que l’objectif ultime du présent travail noté #3 d’INF 1430, est d’implémenter en langage python, dans le main des trois codes python programmés par Gonzalo Alfredo Romero Francia, toutes les lignes de code permettant d’implémenter ce que chacun des 10 critères d’évaluation nommés ci-haut dans la présente page, effectue comme type d’évaluation à ces trois codes python RSA, El-Gamal et ECC, codés et programmés par Gonzalo Alfredo Romero Francia.

L’auteur du présent document a décidé, afin de satisfaire à l’objectif ultime du présent travail noté #3 d’INF 1430, qui est décrite juste dans le paragraphe montré ci-dessus, de réaliser un nombre énorme de recherches d’information sur Google et autres moteurs de recherche, et réaliser des recherches d’informations sur les 10 critères d’évaluation nommés à la page 3 du présent document, dans le but d’obtenir des informations permettant à l’auteur du présent document de coder et programmer les lignes de code python implémentant, de manière très efficace et très optimale, chacun de ces 10 critères d’évaluation déjà nommée ci-dessus, ces lignes de code python seront codés et programmés dans la classe main déjà présente dans ses trois codes python RSA, El-Gamal et ECC déjà en version distribuable, paramétrable, corrigée, améliorée et optimisée lors de la réalisation du travail noté #2 d’INF 1430, ces trois codes python sont en version 3, 5 et 6 respectivement

Il est à noter que les codes python implémentant les algorithmes de chiffrement et de déchiffrement asymétriques RSA, El-Gamal et ECC, montrés dans les sections suivantes du présent document, seront utilisés comme codes python codés et programmés en version finale et fonctionnelle, distribuable, paramétrable, corrigée, améliorée et optimisée, implémentant chacun de ces trois codes python les 10 critères d’évaluation nommés à la partie supérieure de la page 3 du présent document, et le présent travail noté #3 d’INF 1430 va expliquer, avec luxe de détails, les techniques et approches de programmation et les techniques et approches de modélisation des calculs mathématiques, arithmétiques et informatiques en lien avec ces 10 critères d’évaluation, en termes de conception, de codage et programmation, de développement et de test de fonctionnalité de codes python produits.

**Projet de fin d’études - Implémentation des dix critères d’évaluation des algorithmes de chiffrage asymétrique - Planification de la réalisation du projet de fin d’études nommé Implémentation et comparaison des algorithmes de chiffrage asymétrique – Continuation de la réalisation de ce projet de fin d’études**

Lors de la réalisation de toutes les activités et tâches associées à l’étape de planification du projet de fin d’études nommé « Implémentation et comparaison des algorithmes de chiffrage asymétrique », Gonzalo Alfredo Romero Francia a déjà fourni la planification totale et globale de la réalisation de ce projet de fin d’études, qui vise à montrer l'efficacité et l'efficience de l'implémentation des trois algorithmes de chiffrement et de déchiffrement asymétriques nommés RSA (Ron Rivest, Adi Shamir et Leonard Adleman), El-Gamal et ECC (Elliptic Curve Cryptography), afin de fournir et d'offrir des solutions à certains problèmes qui surviennent en général lorsque l'on les implémente en un langage de programmation donné (C, C#, C++, Java).

Afin de satisfaire à cet objectif général, global et primaire, Gonzalo Alfredo Romero Francia a décidé de concevoir, lors de la réalisation des premières activités et tâches en relation avec ce projet de fin d’études, pour la rédaction du document Word du travail noté #1 d’INF 1430, une planification de la réalisation de toutes les tâches de toutes les étapes de la réalisation du projet nommé « Implémentation et comparaison des algorithmes de chiffrage asymétrique », d’une manière très structuré et très organisé, ce projet étant le projet qu'il a choisi de réaliser dans le cadre de la réalisation de son cours INF 1430.

Afin de réaliser le projet nommé ci-dessus avec efficacité et efficience, et sans subir aucun retard ni aucun problème de réalisation de ce projet qui pourrait compromettre la réussite de la réalisation de ce projet, Gonzalo Alfredo Romero Francia a conçu, avec une énorme facilité, la planification de la réalisation de son projet nommé « Implémentation et comparaison des algorithmes de chiffrage asymétrique » suivante :

* Phase 1: faire des recherches sur Internet, sur les algorithmes de chiffrement asymétrique RSA, El-Gamal et ECC.
* Phase 2: me familiariser avec la théorie et les notions qu'utilisent ces trois algorithmes de chiffrement asymétrique, sous le plan purement théorique (lecture de pages web et visionnement de vidéos).
* Phase 3: me familiariser avec la théorie et les notions qu'utilisent ces trois algorithmes de chiffrement asymétrique, sous le plan purement mathématique et informatique (modélisation des formules et équations utilisées par ces trois algorithmes de chiffrement asymétrique en utilisant le logiciel Maple 2024, que je vais acheter prochainement).
* Phase 4: commencer à implémenter ces trois algorithmes de chiffrement asymétrique, tout en implémentant avec C++, chacune des formules et équations que ces trois algorithmes utilisent, tant pour chiffrer que pour déchiffrer des messages variés, afin d’obtenir une version paramétrable et distribuable du code Python associé à ces 3 algorithmes asymétriques.
* Phase 5: Terminer la programmation de l'implémentation des algorithmes RSA, El-Gamal et ECC, pour rendre les 3 programmes C++ en version finale.
* Phase 6: réaliser des mesures de performance à ces trois algorithmes de chiffrement asymétrique, en utilisant plusieurs critères qui ont été déterminées par moi en février dernier de l’année 2024. Ces critères d’évaluation ont été annoncés au professeur tuteur de Gonzalo Alfredo Romero Francia.
* Phase 7: rédiger le document Word nécessaire pour montrer le résultat de l'implémentation de ces 3 algorithmes, ainsi que les résultats obtenus.

Note : la phrase ombrée de couleur verte indique que Gonzalo Alfredo Romero Francia a déjà terminé la phase 6 de la réalisation de son projet de fin d’études nommé « Implémentation et comparaison des algorithmes de chiffrage asymétrique », au moment de la finalisation de l’amélioration et de l’optimisation des codes python montrés au travail noté#1 d’INF 1430, amélioration et optimisation qui a terminé le 24 Mai 2024 à 16:30 PM!

**Description du problème principal, qui devient aussi l’objectif principal de la réalisation de la phase 5, du projet de fin d’études nommé « Implémentation et comparaison des algorithmes de chiffrage asymétrique »**

Lors de la réalisation du travail noté #1 d’INF 1430, Gonzalo Alfredo Romero Francia avait réussi à rendre paramétrable et distribuable la version de base de codes python implémentant les algorithmes RSA, El-Gamal et ECC, qu’il avait trouvés sur internet, tout en réalisant des recherches d’informations sur comment implémenter, en langage python, ces algorithmes de chiffrement et de déchiffrement asymétrique .Le résultat final de toutes ces recherches d’informations sont les trois codes python implémentant les algorithmes RSA, El-Gamal et ECC, en version paramétrable et distribuable, ces trois codes python sont montrés dans la toute dernière section du document Word du travail noté #1.

Cependant, et en réalisant des observations rapides et approfondies, l’auteur du présent document s’est vite rendu compte que ces trois codes python, en version paramétrable et distribuable, ne sont pas du tout ni améliorés ni optimisés, ce qui rend les trois codes python déjà décrits ci-dessus, des versions pas corrigées, améliorées ni optimisées, qui répondraient de manière très efficace aux deux demandes de l’énoncé du projet de fin d’études nommé « Implémentation et comparaison des algorithmes de chiffrage asymétrique », qui sont les suivantes :

* Ces algorithmes doivent être paramétrables. Les paramètres attendus sont : la taille du bloc de données (64 bits, 128 bits, etc.), la taille de la clé (64 bits, 128 bits, etc.).
* En fin, l’étudiant doit fournir un tableau comparatif des performance (temps d’exécution, l’effet avalanche, etc.) de ces algorithmes

Quoi que les trois codes python implémentant les algorithmes RSA, El-Gamal et ECC, sont déjà paramétrables et distribuables, ces trois codes python ne contiennent aucune ligne de code python qui offre des choix de paramétrisation de la taille des messages clairs et plein, et des choix de paramétrisation de la taille des clés publique et privée générée par ces trois algorithmes de chiffrement et de déchiffrement asymétrique, De plus ces trois codes python déjà décrits ci-haut ne contiennent pas des lignes de code qui réalisent la logique informatique de gestion de ces choix de paramétrisation de la taille des messages clairs et plein, ni celle des choix de paramétrisation de la taille des clés publique et privée générée par ces trois algorithmes de chiffrement et de déchiffrement asymétrique.

C’est pour les deux raison déjà décrites dans le paragraphe ci-dessus, que Gonzalo Alfredo Romero Francia a décidé de travailler et de retravailler trop de fois, le nombre de fois que soit nécessaire, ses trois codes python implémentant les algorithmes de chiffrement et de déchiffrement asymétrique RSA, El-Gamal et ECC, afin de corriger, améliorer et optimiser au complet ces trois codes python, en termes logiques et informatiques. Pour cela, il a décidé de changer et modifier, au besoin, des lignes de code python de ces trois codes implémentant les trois algorithmes de chiffrement et de déchiffrement asymétriques nommé RSA, El-Gamal et ECC, ou même toutes les lignes de code python de ces trois codes python implémentant ces algorithmes de chiffrement et de déchiffrement asymétriques, afin de les rendre en codes python paramétrables, distribuables, corrigés, améliorés et optimisés au complet.

De plus, l’auteur du présent document a décidé de concevoir et de créer une structure de code python, trop complet et trop structuré, permettant tant de présenter, de manière sommaire et détaillée, chacun des codes python déjà décrits dans le paragraphe ci-dessus, que de présenter les différents choix qu’offrent chacun de ces trois codes python, en ce qui a trait la taille des messages clairs et pleins, et la taille des clés publique et privée générées, aux utilisateurs de chacun des trois codes python implémentant les algorithmes de chiffrement et de déchiffrement asymétrique RSA, El-Gamal et ECC. La solution finale résultante des opérations décrites dans la présente section seront présentés et développés aux sections suivantes du présent document.

Maintenant que l’auteur du présent document a réussi à réaliser toutes les tâches décrites dans les paragraphes nommés ci-dessus, lors de la réalisation de son travail noté #2 d’INF 1430, il est arrivé à l’étape suivante de la réalisation de son projet de fin d’études nommé «Implémentation et comparaison des algorithmes de chiffrage asymétrique » , **qui est l’implémentation, en langage python 3.12.1, des dix critères d’évaluation déjà nommées à la partie supérieure de la page 3 du présent document, et cela pour les trois codes python RSA, El-Gamal et ECC, en version 3, 5 et 6 respectivement, ces trois versions de codes python RSA, El-Gamal et ECC, sont déjà distribuables, paramétrables, corrigées, améliorées et optimisées,** alors il ne manque qu’implémenter à ces trois codes python déjà nommés ci-dessus, chacun de ces dix critères d’évaluation, qui sont les suivantes :

* Temps d'exécution.
* Utilisation de la mémoire.
* Taille des clés générées.
* Complexité algorithmique.
* Gestion des grandes quantités de données.
* Comparaison des tailles de clés.
* Comparaison des temps d'exécution avec différentes tailles de clés.
* Quantité de mémoire utilisée lors de la génération de clés.
* Comparaison de la taille des messages clairs et pleins utilisés, pour bien sécuriser les messages cryptés et décryptés
* L'effet avalanche.

Les prochaines sections du présent document vont montrer, de manière très claire, précise et détaillée, toutes les approches et techniques de codage et programmation, développement et test de fonctionnement de codes python produits, ainsi que toutes les approches et techniques de modélisation, en langage python 3.12.1, des calculs mathématiques, arithmétiques et informatiques réalisées, pour implémenter chacun des dix critères d’évaluation montrés ci-dessus, dans les trois codes python RSA, El-Gamal et ECC, en version 3, 5 et 6 respectivement, déjà codés et programmés et montrés dans l’annexe du travail noté #2 d’INF 1430, la description détaillé de ces techniques et approches d’implémentation de ces dix critères d’évaluation déjà décrite dans le présent paragraphe, seront abordées en détail dans les prochaines sections du présent document du travail noté #3 d’INF 1430.

**Modélisation de la solution au problème principal, qui devient aussi l’objectif principal de la réalisation de la phase 5, du projet de fin d’études nommé « Implémentation et comparaison des algorithmes de chiffrage asymétrique »**

Après avoir décrit, avec luxe de détails, le problème principal du projet de fin d’études nommé « Implémentation et comparaison des algorithmes de chiffrage asymétrique », Gonzalo Alfredo Romero Francia a décidé, afin de fournir une solution plus qu’efficace à ce problème déjà décrit à la section précédente, d’utiliser toutes les notions des approches de développement/programmation de programmes/applications informatiques nommés « Agile Programming » et « Extreme Engineering », qui sont deux philosophies de développement et de programmation de programmes et applications informatiques permettant de sauver du temps, des efforts et d’argent, en général, aux compagnies qui les utilisent, pour développer et programmer leurs applications informatiques.

La programmation agile (« Agile Programming ») et l’ingénierie extrême (« Extreme Engineering ») sont deux approches de développement logiciel qui visent à améliorer la flexibilité, la qualité et l'efficacité du processus de développement. La programmation agile met l'accent sur la collaboration interdisciplinaire, l'adaptation aux changements et la livraison continue de logiciels fonctionnels, tandis que "Extreme Engineering" met l'accent sur la rigueur technique, la simplicité, et l'automatisation des tests et du déploiement. Ces approches sont toutes deux largement utilisées dans l'industrie du développement logiciel pour répondre aux défis de la création de logiciels de qualité dans des délais serrés.

Alors, en utilisant ces deux approches et philosophies de développement/programmation de programmes et applications informatiques, l’auteur du présent document a réussi à concevoir à développer et à programmer ses trois codes python implémentant les algorithmes RSA, El-Gamal et ECC, tout en se concentrant sur un aspect ou une caractéristique mathématique, logicielle et informatique, ou sur plusieurs aspects ou caractéristiques mathématiques, logicielles et informatiques, de chacune des opérations mathématiques utilisées par ces trois algorithmes de chiffrement et de déchiffrement asymétrique.

Cela veut dire que, pour chaque aspect ou caractéristique mathématique, logicielle et informatique, ou pour un petit ensemble d’aspects et de caractéristiques mathématiques, logicielles et informatiques, de chacune des opérations mathématiques utilisées par ces trois algorithmes de chiffrement et de déchiffrement asymétrique, Gonzalo Alfredo Romero Francia a réussi à coder le code python en version finale, implémentant les algorithmes de chiffrement et de déchiffrement asymétrique RSA, El-Gamal et ECC, tout en créant plus de deux versions différentes des trois codes python implémentant ces trois algorithmes, chaque version de ces trois codes python corrige, améliore et optimise, un peu et pas à pas, chacune des trois codes python de base implémentant les algorithmes RSA, El-Gamal et ECC, déjà montrés à la dernière section du travail noté #1 d’INF 1430, et jusqu’à obtenir un rendement informatique et logicielle trop haute, de l’ordre de 97% et même plus.

En utilisant les approches de développement/programmation « Agile Programming » et « Extreme Engineering », l’auteur du présent document a réussi à coder, peu à peu, et en se concentrant sur un ou plusieurs aspects ou sur une ou plusieurs caractéristiques mathématiques, logicielles et informatiques, de chacun des trois codes python implémentant les algorithmes de chiffrement et de déchiffrement asymétriques RSA, El-Gamal et ECC, chacun des codes python implémentant ces trois algorithmes asymétriques de manière très efficace et optimale, ces trois codes python sont devenus, après toutes les modifications décrites ci-dessus, des codes python paramétrables, distribuables, corrigés, améliorés et optimisés, car ils donnent des rendements logicielles et informatiques supérieures à 97%, selon les centaines de tests logiciels qu’il a effectué à toutes les 3 versions du code python RSA, à toutes les 6 versions du code python El-Gamal, et à toutes les 5 versions du code python ECC. Les résultats finaux fournis par ces trois codes python seront montrés à la prochaine section.

Ensuite, et lors de la réalisation du présent travail noté #3 d’INF 1430, Gonzalo Alfredo Romero Francia a décidé d’utiliser la panoplie d’informations diverses qu’il a trouvées sur internet, en utilisant le moteur de recherche Google, Yahoo et bien d’autres, en lien avec l’implémentation en langage python 3.12.1, de chacun des dix critères d’évaluation déjà montrées à la partie inférieure de la section précédente, ces informations ont déjà été recueillis et stockés dans des documents/fichiers texte (format .txt) et ont été disponibles pour que Gonzalo Alfredo Romero Francia commence à coder et à programmer l’implémentation de ces 10 critères d’évaluation de ses trois codes python RSA en version 3, El-Gamal en version 6 et ECC en version 5, afin de réaliser des corrections, des améliorations et des optimisations à tous ses codes python produits, tout en utilisant les approches et techniques de programmation nommées « Programmation Agile » et « Ingénierie Extrême », qui sont des approches et des techniques de programmation, de développement et de test de fonctionnalité de codes python produits récursives, qui améliorent un code python d’une version X, en un code python amélioré et optimisé en version X+1.

**Phase 5 – Explication de l’approche de programmation utilisé, pour rendre le code python associé à l’Algorithme RSA, paramétrable, distribuable, corrigé, amélioré et optimisé**

**Explication de l’approche de programmation utilisé, pour rendre le code python de base associé à l’Algorithme RSA, en un code python corrigé, amélioré et optimisé**

Gonzalo Alfredo Romero Francia, lors de la réalisation de toutes les activités et tâches en lien aux phases 1 à 5, de la planification de son projet de fin d’études nommé « Implémentation et comparaison des algorithmes de chiffrage asymétrique », il a fourni à la dernière section du document Word du travail noté #2 du cours INF 1430, c-à-d à l’Annexe de ce document, les trois codes python implémentant les algorithmes RSA, El-Gamal et ECC, en version paramétrable et distribuable, corrigée, améliorée et optimisée, ces trois codes python en version finale sont aussi en version 3, 6 et 5 respectivement, ce qui a été déjà montré clairement par le document Word de ce travail noté #2 d’INF 1430.

Il est à noter que ces trois codes python implémentant les algorithmes RSA, El-Gamal et ECC, paramétrables et distribuables, encore performants en termes de rapidité de compilation et de calcul, et en termes de consommation mineure de la mémoire de la laptop de Gonzalo Alfredo Romero Francia, sont déjà aussi corrigés, améliorés et optimisés au complet, alors il a décidé d’utiliser les informations qu’il a trouvées au tout début de la réalisation de son projet de fin d’études nommé «Implémentation et comparaison des algorithmes de chiffrage asymétrique », ces informations lui ont permis de comprendre comment corriger, améliorer et optimiser au complet chacun des trois codes python de base implémentant les algorithmes RSA, El-Gamal et ECC, qui sont déjà paramétrables et distribuables, pour ensuite savoir comment implémenter, en langage python 3.12.1, chacun des dix critères d’évaluation suivantes, dans ses trois codes python RSA en version 3, El-Gamal en version 6 et ECC en version 5 :

* Temps d'exécution.
* Utilisation de la mémoire.
* Taille des clés générées.
* Complexité algorithmique.
* Gestion des grandes quantités de données.
* Comparaison des tailles de clés.
* Comparaison des temps d'exécution avec différentes tailles de clés.
* Quantité de mémoire utilisée lors de la génération de clés.
* Comparaison de la taille des messages clairs et pleins utilisés, pour bien sécuriser les messages cryptés et décryptés
* L'effet avalanche.

Alors, et afin de réussir à bien implémenter, en langage python 3.12.1, chacun de ces 10 critères d’évaluation, pour ajouter le bloc de code main implémentant ces 10 critères d’évaluation à ses codes python RSA, El-Gamal et ECC déjà décrits dans le paragraphe ci-dessus, Gonzalo Alfredo Romero Francia a décidé de mettre en pratique la technique de programmation modulaire nommé « Programmation petit à petit », c-à-d, qu’il a décidé de coder les lignes de code python associées à l’implémentation de chacun des dix critères d’évaluation montrés ci-haut, les insérer dans le main de ses codes python RSA en version 3, El-Gamal en version 6 et ECC en version 5, tester le code python modifié tout en le compilant dans la console de son laptop personnel, voir les résultats obtenus et au besoin, corriger, améliorer et optimiser une ou quelques lignes de code python, des codes python implémentant les algorithmes RSA, El-Gamal et ECC déjà décrits ci-haut, associé à un type de calcul mathématique, arithmétique et informatique, **en lien avec l’ensemble de calculs mathématiques, arithmétiques et informatiques utilisé par l’implémentation de chacun de ces 10 critères d’évaluation des algorithmes RSA, El-Gamal et ECC,** implémentés par les codes python déjà décrit ci-haut, pour ensuite corriger, améliorer et optimiser la ou les quelques lignes de code python associés aux autres types de calcul mathématique, arithmétique et informatique, **en lien avec l’ensemble de calculs mathématiques, arithmétiques et informatiques utilisé par l’implémentation ces 10 critères d’évaluation en python, ajoutée dans la classe main des trois codes python implémentant les algorithmes de chiffrement et de déchiffrement asymétriques RSA, El-Gamal et ECC.**

Afin de montrer très clairement la nature et la description des types d’implémentation logicielle et informatique, que Gonzalo Alfredo Romero Francia a décidé de fournir, tout en considérant les informations trouvées sur Internet, décrites dans les paragraphes montrés ci-dessus, pour implémenter, en langage python 3.12.1. chacun des dix critères d’évaluation de ses trois codes python RSA en version 3, El-Gamal en version 6 et ECC en version 5, déjà montrés à la partie supérieure de la présente page 8 du présent document, **des textes très clairs, très précis et très détaillés expliquant ce font chacune des lignes de code implémentant chacun de ces 10 critères d’évaluation, qui sont dans la classe main de ses trois codes python RSA en version 3, El-Gamal en version 6 et ECC en version 5, de même qu’il a décidé de fournir, au besoin, et lorsqu’un des codes python, qui sont les codes python RSA en version 3, El-Gamal version 6 et ECC version 5, a changé de quelques lignes de code ou d’une seule ligne de code python, Gonzalo Alfredo Romero Francia a décidé de décrire les changements et modifications faites, et cela de manière très claire, très précise et très détaillée.**

Afin d’être clairs, Gonzalo Alfredo Romero Francia a décidé de réaliser des analyses complètes entre le code python implémentant les algorithmes RSA en version 3, El-Gamal en version 6 et ECC en version 5, qui sont des codes python distribuables, paramétrables, corrigés, améliorés et optimisés, et les codes python RSA en version 3, El-Gamal en version 6 et ECC en version 5, aussi en version distribuables, paramétrables, corrigées, améliorées et optimisées, et implémentant, dans la classe main de ces trois codes python, pour expliquer comment l’auteur du présent document à réussi d’implémenter, en langage python 3.12.1, chacun des dix critères d’évaluation décrites à la partie supérieure de la page 9 du présent document, de même qu’il a décidé de montrer les résultats fournis par l’implémentation de ces dix critères d’évaluations qu’il a ajoutées à ses trois codes python RSA en version 3, El-Gamal en version 6 et ECC en version 5, tout en ajoutant deux figures montrant le fonctionnement de compilation de chacun des 30 codes python implémentant ces 10 critères d’évaluation, car l’auteur du présent document a les trois codes python déjà nommés ci-dessus, et chacun de ces trois codes python implémentera chacun des dix critères d’évaluation montrés à la page 9 du présent document. Le voici la liste complète de ces analyses de la nature et de la qualité d’implémentation de ces dix critères d’évaluation, ajoutées au main des codes python RSA en version 3, El-Gamal en version 6 et ECC en version 5 :

**Analyse complète de la nature et de la qualité d’implémentation de ces dix critères d’évaluation, ajoutées au main du code python RSA en version 3 :**

* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Temps d’exécution », ajoutée au main du code python RSA en version 3.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Utilisation de la mémoire », ajoutée au main du code python RSA en version 3.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Taille des clés générées », ajoutée au main du code python RSA en version 3.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Complexité algorithmique », ajoutée au main du code python RSA en version 3.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Gestion de grandes quantités de données », ajoutée au main du code python RSA en version 3.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Comparaison des tailles des clés », ajoutée au main du code python RSA en version 3.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Comparaison des temps d’exécution avec différentes tailles de clés », ajoutée au main du code python RSA en version 3.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Quantité de mémoire utilisée lors de la génération de clés », ajoutée au main du code python RSA en version 3.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Comparaison de la taille des messages clairs et pleins utilisés, pour bien sécuriser les messages cryptés et décryptés », ajoutée au main du code python RSA en version 3.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Effet Avalanche », ajoutée au main du code python RSA en version 3.

**Analyse complète de la nature et de la qualité d’implémentation de ces dix critères d’évaluation, ajoutées au main du code python El-Gamal en version 6 :**

* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Temps d’exécution », ajoutée au main du code python El-Gamal en version 6.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Utilisation de la mémoire », ajoutée au main du code python El-Gamal en version 6.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Taille des clés générées », ajoutée au main du code python El-Gamal en version 6.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Complexité algorithmique », ajoutée au main du code python El-Gamal en version 6.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Gestion de grandes quantités de données », ajoutée au main du code python El-Gamal en version 6.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Comparaison des tailles des clés », ajoutée au main du code python El-Gamal en version 6.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Comparaison des temps d’exécution avec différentes tailles de clés », ajoutée au main du code python El-Gamal en version 6.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Quantité de mémoire utilisée lors de la génération de clés », ajoutée au main du code python El-Gamal en version 6.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Comparaison de la taille des messages clairs et pleins utilisés, pour bien sécuriser les messages cryptés et décryptés », ajoutée au main du code python El-Gamal en version 6.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Effet Avalanche », ajoutée au main du code python El-Gamal en version 6.

**Analyse complète de la nature et de la qualité d’implémentation de ces dix critères d’évaluation, ajoutées au main du code python ECC en version 5 :**

* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Temps d’exécution », ajoutée au main du code python ECC en version 5.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Utilisation de la mémoire », ajoutée au main du code python ECC en version 5.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Taille des clés générées », ajoutée au main du code python ECC en version 5.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Complexité algorithmique », ajoutée au main du code python ECC en version 5.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Gestion de grandes quantités de données », ajoutée au main du code python ECC en version 5.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Comparaison des tailles des clés », ajoutée au main du code python ECC en version 5.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Comparaison des temps d’exécution avec différentes tailles de clés », ajoutée au main du code python ECC en version 5.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Quantité de mémoire utilisée lors de la génération de clés », ajoutée au main du code python ECC en version 5.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Comparaison de la taille des messages clairs et pleins utilisés, pour bien sécuriser les messages cryptés et décryptés », ajoutée au main du code python ECC en version 5.
* Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Effet Avalanche », ajoutée au main du code python ECC en version 5.

Le présent travail noté #3 va aborder, développer et montrer, de manière très détaillée et très précise, l’analyse complète de chacune des lignes de code python implémentant les algorithmes de chiffrement et de déchiffrement asymétrique RSA, El-Gamal et ECC, en version 3, 5 et 6 respectivement, de sorte que l’on réussisse à faciliter la compréhension de comment Gonzalo Alfredo Romero Francia a réussi à modifier ses trois codes python implémentant les algorithmes RSA en version 3, El-Gamal en version 6 et ECC en version 5, afin d’implémenter et d’ajouter, dans ses trois codes python, les blocs de code python implémentant les dix critères d’évaluation suivantes, afin d’évaluer la performance et le rendement fournis par ces trois codes python qui sont en version finale et fonctionnelle, distribuable, paramétrable, corrigée, améliorée et optimisée (RSA en version 3, El-Gamal en version 6 et ECC en version 5) :

* Temps d'exécution.
* Utilisation de la mémoire.
* Taille des clés générées.
* Complexité algorithmique.
* Gestion des grandes quantités de données.
* Comparaison des tailles de clés.
* Comparaison des temps d'exécution avec différentes tailles de clés.
* Quantité de mémoire utilisée lors de la génération de clés.
* Comparaison de la taille des messages clairs et pleins utilisés, pour bien sécuriser les messages cryptés et décryptés
* L'effet avalanche.

**Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Temps d’exécution », ajoutée au main du code python RSA en version 3.**

En réalisant une forte quantité de recherches d’informations, Gonzalo Alfredo Romero Francia a réussi d’améliorer, de corriger, et d’optimiser son code python de base implémentant l’algorithme de chiffrement et de déchiffrement RSA, pour ensuite obtenir des versions de ce code python plus performant, plus robustes et plus structurés et mieux organisées, ce qui a déjà été réalisé au travail noté #2 d’INF 1430. Maintenant que l’auteur du présent document a en main son code python RSA en version 3, il est prêt à utiliser et mettre en pratique toutes les informations qu’il a trouvées en février dernier, lui enseignant comment implémenter, en langage python 3.12.1, le critère d’évaluation nommé « Temps d’exécution », ces informations qu’il a trouvées en février dernier, sont les suivantes :

* Importer les modules random, math et time au début du code python RSA en version 3.
* Envelopper les parties du code python RSA en version 3 que l’on souhaite évaluer, dans des blocs de code python commençant par la ligne de code python « start\_time = time.time() » et terminant par la ligne de code python « end\_time = time.time() ».
* Calculer la différence entre ce que calcule la ligne de code « end\_time » et « start\_time » pour obtenir le temps d'exécution de la partie du code que l’on a implémenté la mesure du temps d’exécution.

En utilisant ces informations trouvées sur internet en février dernier, Gonzalo Alfredo Romero Francia a réussi à modifier et à obtenir le code python RSA en version 3 qui est montré à la section « Annexe » du présent document. Maintenant que l’auteur du présent document a son code python RSA en version 3 implémentant le critère d’évaluation nommé « Temps d’exécution », il a décidé de fournir les informations suivantes, expliquant comment il a réussi à implémenter, en langage python 3.12.1, le critère d’évaluation de ce code python nommé « Temps d’exécution » :

**Modifications apportées au code Python pour l'évaluation Du calcul du temps d’exécution**

**1. Importation des bibliothèques**

Ajout de la bibliothèque time :

**Premier code :**

import random

**Deuxième code :**

import random

import time

**2. Fonction de décodage de message**

Modification de la fonction decode\_message pour retourner une chaîne de caractères numériques au lieu des caractères ASCII :

**Premier code :**

def decode\_message(encoded\_message):

return ''.join(chr(char) for char in encoded\_message)

**Deuxième code :**

def decode\_message(encoded\_message):

return ''.join(str(char) for char in encoded\_message)

**3. Mesure du temps d'exécution**

Ajout des mesures de temps d'exécution pour les différentes parties du processus RSA :

**Deuxième code (section ajoutée) :**

if \_\_name\_\_ == '\_\_main\_\_':

# Code block that measures all the execution times associated to the RSA algorithm

# Measure the total execution time

start\_time\_total = time.time()

# Measure the key generation time

start\_time\_key\_generation = time.time()

public\_key, private\_key = generate\_keys(user\_response\_key\_size // 4)

end\_time\_key\_generation = time.time()

key\_generation\_time = end\_time\_key\_generation - start\_time\_key\_generation

# Print/display the key generation time

print("\nKey generation time: {:.6f} seconds ({:.6f} milliseconds)".format(key\_generation\_time, key\_generation\_time \* 1000))

# Measure the encoding time

start\_time\_encoding = time.time()

encoded\_message = encode\_message(message)

end\_time\_encoding = time.time()

encoding\_time = end\_time\_encoding - start\_time\_encoding

# Print/display the encoding time

print("\nEncoding time: {:.6f} seconds ({:.6f} milliseconds)".format(encoding\_time, encoding\_time \* 1000))

# Measure the decoding time

start\_time\_decoding = time.time()

decrypted\_message = [decrypt(char, private\_key) for char in encrypted\_message]

decoded\_message = decode\_message(decrypted\_message)

end\_time\_decoding = time.time()

decoding\_time = end\_time\_decoding - start\_time\_decoding

# Print/display the decoding time

print("\nDecoding time: {:.6f} seconds ({:.6f} milliseconds)".format(decoding\_time, decoding\_time \* 1000))

# Print/display the total execution time

end\_time\_total = time.time()

total\_time = end\_time\_total - start\_time\_total

print("\nTotal execution time: {:.6f} seconds ({:.6f} milliseconds)".format(total\_time, total\_time \* 1000))

**4. Position du code de génération des clés et de chiffrement/déchiffrement**

Déplacement du code de génération des clés et de chiffrement/déchiffrement à l'intérieur du bloc if \_\_name\_\_ == '\_\_main\_\_' :

**Premier code :**

print("Generating RSA keys...")

public\_key, private\_key = generate\_keys(user\_response\_key\_size // 4)

print("RSA keys generated successfully.")

message = msg

encoded\_message = encode\_message(message)

encrypted\_message = [encrypt(char, public\_key) for char in encoded\_message]

decrypted\_message = [decrypt(char, private\_key) for char in encrypted\_message]

decoded\_message = decode\_message(decrypted\_message)

print("\nInitial message:")

print(message)

print("\nPublic key:")

print(public\_key)

print("\nPrivate key:")

print(private\_key)

print("\nEncoded message (encrypted by public key):")

print(encrypted\_message)

print("\nDecoded message (decrypted by private key):")

print(decoded\_message)

**Deuxième code :**

if \_\_name\_\_ == '\_\_main\_\_':

# Code block that measures all the execution times associated to the RSA algorithm

...

print("Generating RSA keys...")

public\_key, private\_key = generate\_keys(user\_response\_key\_size // 4)

print("RSA keys generated successfully.")

message = msg

encoded\_message = encode\_message(message)

encrypted\_message = [encrypt(char, public\_key) for char in encoded\_message]

decrypted\_message = [decrypt(char, private\_key) for char in encrypted\_message]

decoded\_message = decode\_message(decrypted\_message)

print("\nInitial message:")

print(message)

print("\nPublic key:")

print(public\_key)

print("\nPrivate key:")

print(private\_key)

print("\nEncoded message (encrypted by public key):")

print(encrypted\_message)

print("\nDecoded message (decrypted by private key):")

print(decoded\_message)

# Measure the total execution time

end\_time\_total = time.time()

total\_time = end\_time\_total - start\_time\_total

print("\nTotal execution time: {:.6f} seconds ({:.6f} milliseconds)".format(total\_time, total\_time \* 1000))

**5. Affichage des temps d'exécution**

Ajout des impressions pour afficher les temps d'exécution pour chaque étape (génération de clés, encodage et décodage) et le temps total :

**Deuxième code (section ajoutée) :**

# Print/display the key generation time

print("\nKey generation time: {:.6f} seconds ({:.6f} milliseconds)".format(key\_generation\_time, key\_generation\_time \* 1000))

# Print/display the encoding time

print("\nEncoding time: {:.6f} seconds ({:.6f} milliseconds)".format(encoding\_time, encoding\_time \* 1000))

# Print/display the decoding time

print("\nDecoding time: {:.6f} seconds ({:.6f} milliseconds)".format(decoding\_time, decoding\_time \* 1000))

# Print/display the total execution time

print("\nTotal execution time: {:.6f} seconds ({:.6f} milliseconds)".format(total\_time, total\_time \* 1000))

**Résumé des modifications :**

Importation de la librairie python « time » pour mesurer les temps d'exécution du processus de la génération des clés, d'encodage et de décodage de messages clairs et pleins.

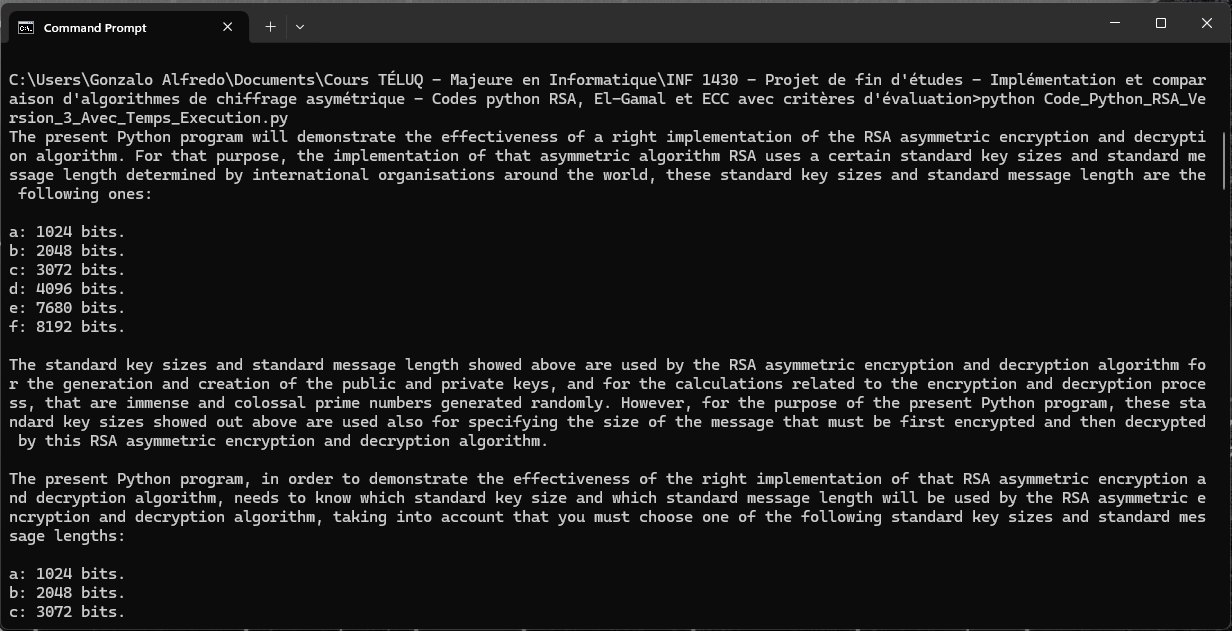
Modification de la fonction python « decode\_message » pour qu’elle retourne une chaîne de caractères numériques.

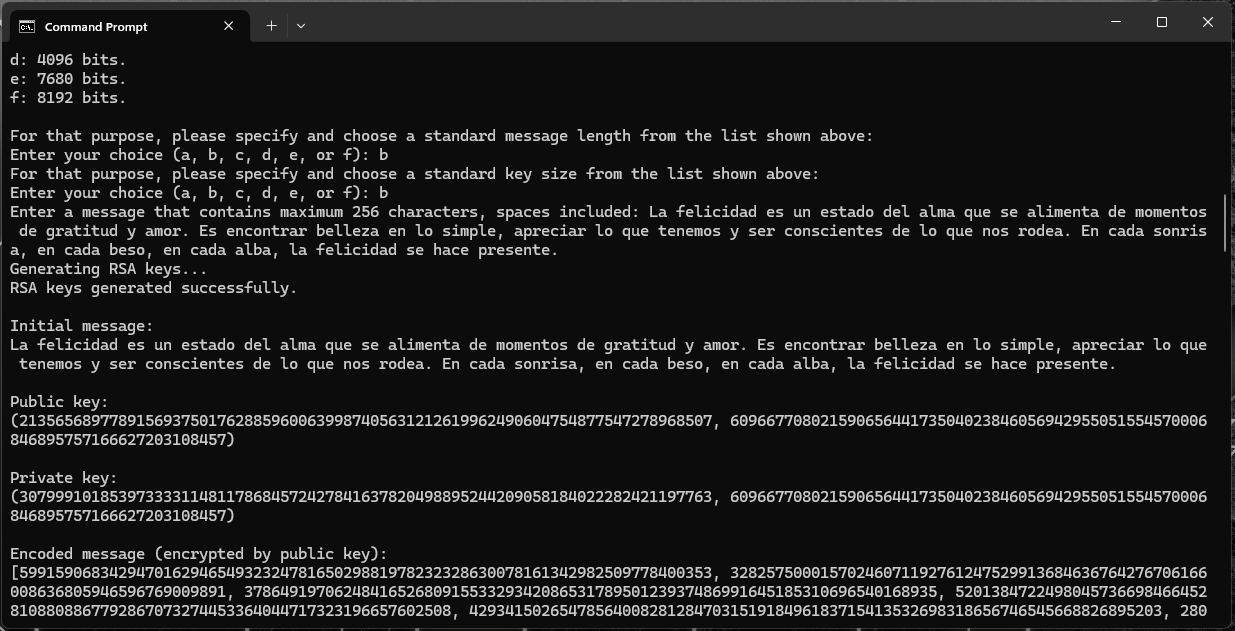
Ajout du bloc (classe python) if \_\_name\_\_ == '\_\_main\_\_' pour mesurer les temps d'exécution et organiser le flux principal du programme.

Ajout des mesures de temps pour la génération des clés, l'encodage, le décodage, et le temps total d'exécution de ces processus de calcul mathématiques, arithmétiques et informatiques du code python RSA. En version 3.

Déplacement et regroupement du code de génération des clés et de chiffrement/déchiffrement à l'intérieur du bloc principal, qui est au-dessus du bloc (classe) if \_\_name\_\_ == '\_\_main\_\_'.

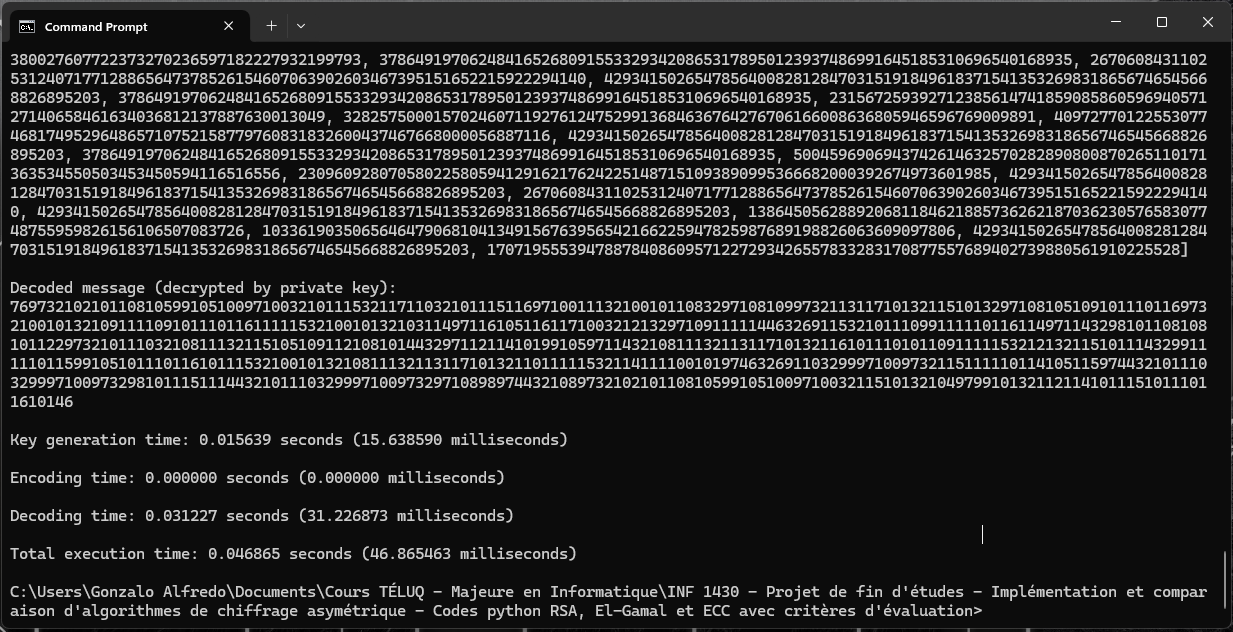
Ces modifications décrites dans le paragraphe montré ci-dessus ajoutent l’implémentation d’un nouveau critère d'évaluation pour mesurer l'efficacité de l'implémentation RSA en termes de temps d'exécution pour différentes étapes du processus de génération de clés publiques et privées, celles du chiffrement et celles du déchiffrement de messages clairs et pleins. Les modifications décrites avec luxe de détails, et montrée dans les paragraphes ci-haut, ont donné de très bons résultats, car Gonzalo Alfredo Romero Francia peut maintenant mesurer, avec beaucoup de précision, le temps que prend son code python RSA en version 3, à générer des clés publique et privée, à chiffrer et à déchiffrer des messages clairs et pleins spécifié par lui, lors de la compilation de ce code python RSA. Le voici la sortie que donne le code python RSA implémentant le critère d’évaluation nommé « Temps d’exécution » :











Les figures montrées ci-dessus montrent très bien que les modifications que Gonzalo Alfredo Romero Francia a faits, a permis à son code python RSA en version 3, d’implémenter le critère d’évaluation nommé « Temps d’exécution », de manière très rapide, très efficace et très optimale, et produisant des résultats excellents, efficaces, optimales et remarquables.

**Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Utilisation de la mémoire », ajoutée au main du code python RSA en version 3.**

En réalisant une forte quantité de recherches d’informations, Gonzalo Alfredo Romero Francia a réussi d’améliorer, de corriger, et d’optimiser son code python de base implémentant l’algorithme de chiffrement et de déchiffrement RSA, pour ensuite obtenir des versions de ce code python plus performant, plus robustes et plus structurés et mieux organisées, ce qui a déjà été réalisé au travail noté #2 d’INF 1430. Maintenant que l’auteur du présent document a en main son code python RSA en version 3, il est prêt à utiliser et mettre en pratique toutes les informations qu’il a trouvées en février dernier, lui enseignant comment implémenter, en langage python 3.12.1, le critère d’évaluation nommé « Utilisation de la mémoire », ces informations qu’il a trouvées en février dernier, sont les suivantes :

* Pour mesurer l'utilisation de la mémoire dans votre programme Python, il faut utiliser le module ou la librairie python « resource ». Ce module ou cette librairie python permet d'interroger et de contrôler les ressources utilisées par un processus associé à l’algorithme RSA, implémenté par le code python RSA en version 3 codé par l’auteur du présent document.
* Il a décidé alors d'utiliser ce module ou cette librairie python nommé « resource » pour implémenter, en langage python 3.12.1, la mesure de la quantité de mémoire utilisée avant et après l'exécution de tous les blocs de code de son code python RSA en version 3.

En utilisant ces informations trouvées sur internet en février dernier, Gonzalo Alfredo Romero Francia a réussi à modifier et à obtenir le code python RSA en version 3 qui est montré à la section « Annexe » du présent document. Maintenant que l’auteur du présent document a son code python RSA en version 3 implémentant le critère d’évaluation nommé « Utilisation de la mémoire », il a décidé de fournir les informations suivantes, expliquant comment il a réussi à implémenter, en langage python 3.12.1, le critère d’évaluation de ce code python nommé « Utilisation de la mémoire » :

**Modifications apportées au code Python RSA pour l'évaluation de l'utilisation de la mémoire**

**1.Importation des bibliothèques :**

Ajout de la bibliothèque psutil pour mesurer la consommation de mémoire :

Premier code :

import random

Deuxième code :

import random

import psutil

**2. Fonction pour mesurer l'utilisation de la mémoire :**

**Ajout de la fonction get\_memory\_usage() pour mesurer l'utilisation de la mémoire avant l'exécution du code :**

**Ajout de la fonction :**

def get\_memory\_usage():

process = psutil.Process()

return process.memory\_info().rss / 1024 # Convertir en kilo-octets

**Exemple d'utilisation de la fonction :**

before\_memory = get\_memory\_usage()

**3. Mesure de l'utilisation de la mémoire :**

**Ajout de la mesure de l'utilisation de la mémoire avant et après l'exécution du code RSA :**

# Example of using the function

before\_memory = get\_memory\_usage()

# Function to calculate the Greatest Common Divisor (GCD) of two numbers

def gcd(a, b):

while b != 0:

a, b = b, a % b

return a

# Function to calculate the modular inverse of a number modulo m

def modinv(a, m):

m0, x0, x1 = m, 0, 1

while a > 1:

q = a // m

m, a = a % m, m

x0, x1 = x1 - q \* x0, x0

return x1 + m0 if x1 < 0 else x1

# Function to test if a number is prime (using the Miller-Rabin primality test)

def is\_prime(n, k=5):

if n <= 1:

return False

if n <= 3:

return True

if n % 2 == 0:

return False

d = n - 1

s = 0

while d % 2 == 0:

d //= 2

s += 1

for \_ in range(k):

a = random.randint(2, n - 1)

x = pow(a, d, n)

if x == 1 or x == n - 1:

continue

for \_ in range(s - 1):

x = pow(x, 2, n)

if x == n - 1:

break

else:

return False

return True

# Function to generate a potential prime number candidate

def generate\_prime\_candidate(length):

p = random.getrandbits(length)

p |= (1 << length - 1) | 1

return p

# Function to generate a prime number close to the specified length

def generate\_prime(length):

p = generate\_prime\_candidate(length)

while not is\_prime(p):

p = generate\_prime\_candidate(length)

return p

# Function to generate RSA keys with a specified key length

def generate\_keys(key\_length):

p = generate\_prime(key\_length // 4)

q = generate\_prime(key\_length // 4)

n = p \* q

phi = (p - 1) \* (q - 1)

e = random.randint(2, phi) # Start from 2 to ensure coprimality with phi

while gcd(e, phi) != 1:

e = random.randint(2, phi)

d = modinv(e, phi)

return ((e, n), (d, n))

# Function to encrypt a message

def encrypt(m, public\_key):

e, n = public\_key

return pow(m, e, n)

# Function to decrypt a message

def decrypt(c, private\_key):

d, n = private\_key

return pow(c, d, n)

# Function to encode a message into a list of integers representing the ASCII codes of characters

def encode\_message(message):

return [ord(char) for char in message]

# Function to decode a message encoded into a string of characters

def decode\_message(encoded\_message):

return ''.join(str(char) for char in encoded\_message)

# This part of the Python program that implements effectively the RSA algorithm, displays the presentation of that Python program (that takes charge of the right choosing of the offered options described below, for the standard key sizes)

print("The present Python program will demonstrate the effectiveness of a right implementation of the RSA asymmetric encryption and decryption algorithm. For that purpose, the implementation of that asymmetric algorithm RSA uses a certain standard key sizes and standard message length determined by international organisations around the world, these standard key sizes and standard message length are the following ones:\n")

print("a: 1024 bits.")

print("b: 2048 bits.")

print("c: 3072 bits.")

print("d: 4096 bits.")

print("e: 7680 bits.")

print("f: 8192 bits.\n")

print("The standard key sizes and standard message length showed above are used by the RSA asymmetric encryption and decryption algorithm for the generation and creation of the public and private keys, and for the calculations related to the encryption and decryption process, that are immense and colossal prime numbers generated randomly. However, for the purpose of the present Python program, these standard key sizes showed out above are used also for specifying the size of the message that must be first encrypted and then decrypted by this RSA asymmetric encryption and decryption algorithm.\n")

print("The present Python program, in order to demonstrate the effectiveness of the right implementation of that RSA asymmetric encryption and decryption algorithm, needs to know which standard key size and which standard message length will be used by the RSA asymmetric encryption and decryption algorithm, taking into account that you must choose one of the following standard key sizes and standard message lengths:\n")

print("a: 1024 bits.")

print("b: 2048 bits.")

print("c: 3072 bits.")

print("d: 4096 bits.")

print("e: 7680 bits.")

print("f: 8192 bits.\n")

print("For that purpose, please specify and choose a standard message length from the list shown above:")

user\_response\_standard\_message\_length = input("Enter your choice (a, b, c, d, e, or f): ")

if user\_response\_standard\_message\_length == "a":

user\_response\_message\_length = 128

elif user\_response\_standard\_message\_length == "b":

user\_response\_message\_length = 256

elif user\_response\_standard\_message\_length == "c":

user\_response\_message\_length = 384

elif user\_response\_standard\_message\_length == "d":

user\_response\_message\_length = 512

elif user\_response\_standard\_message\_length == "e":

user\_response\_message\_length = 960

elif user\_response\_standard\_message\_length == "f":

user\_response\_message\_length = 1024

while user\_response\_standard\_message\_length not in {"a", "b", "c", "d", "e", "f"}:

print("Invalid option. Please choose a valid option (a, b, c, d, e, or f).")

user\_response\_standard\_message\_length = input("Enter your choice (a, b, c, d, e, or f): ")

print("For that purpose, please specify and choose a standard key size from the list shown above:")

user\_response\_standard\_key\_size = input("Enter your choice (a, b, c, d, e, or f): ")

if user\_response\_standard\_key\_size == "a":

user\_response\_key\_size = 128\*8

elif user\_response\_standard\_key\_size == "b":

user\_response\_key\_size = 256\*8

elif user\_response\_standard\_key\_size == "c":

user\_response\_key\_size = 384\*8

elif user\_response\_standard\_key\_size == "d":

user\_response\_key\_size = 512\*8

elif user\_response\_standard\_key\_size == "e":

user\_response\_key\_size = 960\*8

elif user\_response\_standard\_key\_size == "f":

user\_response\_key\_size = 1024\*8

while user\_response\_standard\_key\_size not in {"a", "b", "c", "d", "e", "f"}:

print("Invalid option. Please choose a valid option (a, b, c, d, e, or f).")

user\_response\_standard\_key\_size = input("Enter your choice (a, b, c, d, e, or f): ")

# Checking the consistency between the message length and the key size

while True:

if user\_response\_standard\_message\_length == user\_response\_standard\_key\_size:

break

else:

print("Error: The selected message length must match the selected key size.")

print("Please choose the same option (a, b, c, d, e, or f) for both message length and key size.")

user\_response\_standard\_message\_length = input("Enter your choice for message length (a, b, c, d, e, or f): ")

user\_response\_standard\_key\_size = input("Enter your choice for key size (a, b, c, d, e, or f): ")

if user\_response\_standard\_message\_length == "a":

user\_response\_message\_length = 128

elif user\_response\_standard\_message\_length == "b":

user\_response\_message\_length = 256

elif user\_response\_standard\_message\_length == "c":

user\_response\_message\_length = 384

elif user\_response\_standard\_message\_length == "d":

user\_response\_message\_length = 512

elif user\_response\_standard\_message\_length == "e":

user\_response\_message\_length = 960

elif user\_response\_standard\_message\_length == "f":

user\_response\_message\_length = 1024

if user\_response\_standard\_key\_size == "a":

user\_response\_key\_size = 128\*8

elif user\_response\_standard\_key\_size == "b":

user\_response\_key\_size = 256\*8

elif user\_response\_standard\_key\_size == "c":

user\_response\_key\_size = 384\*8

elif user\_response\_standard\_key\_size == "d":

user\_response\_key\_size = 512\*8

elif user\_response\_standard\_key\_size == "e":

user\_response\_key\_size = 960\*8

elif user\_response\_standard\_key\_size == "f":

user\_response\_key\_size = 1024\*8

if user\_response\_standard\_key\_size == "a":

msg = input("Enter a message that contains maximum 128 characters, spaces included: ")

elif user\_response\_standard\_key\_size == "b":

msg = input("Enter a message that contains maximum 256 characters, spaces included: ")

elif user\_response\_standard\_key\_size == "c":

msg = input("Enter a message that contains maximum 384 characters, spaces included: ")

elif user\_response\_standard\_key\_size == "d":

msg = input("Enter a message that contains maximum 512 characters, spaces included: ")

elif user\_response\_standard\_key\_size == "e":

msg = input("Enter a message that contains maximum 960 characters, spaces included: ")

elif user\_response\_standard\_key\_size == "f":

msg = input("Enter a message that contains maximum 1024 characters, spaces included: ")

while len(msg) > user\_response\_message\_length:

print(f"The message written down has more than {user\_response\_message\_length} characters, including spaces!")

msg = input(f"Please write down a message that contains {user\_response\_message\_length} characters, including spaces:")

print("Generating RSA keys...")

public\_key, private\_key = generate\_keys(user\_response\_key\_size // 4)

print("RSA keys generated successfully.")

message = msg

encoded\_message = encode\_message(message)

encrypted\_message = [encrypt(char, public\_key) for char in encoded\_message]

decrypted\_message = [decrypt(char, private\_key) for char in encrypted\_message]

decoded\_message = decode\_message(decrypted\_message)

print("\nInitial message:")

print(message)

print("\nPublic key:")

print(public\_key)

print("\nPrivate key:")

print(private\_key)

print("\nEncoded message (encrypted by public key):")

print(encrypted\_message)

print("\nDecoded message (decrypted by private key):")

print(decoded\_message)

after\_memory = get\_memory\_usage()

memory\_used = after\_memory - before\_memory

**4. Section ajoutée dans le bloc if \_\_name\_\_ == '\_\_main\_\_', avec une structure conditionnelle pour vérifier si les clés ont déjà été générées :**

if \_\_name\_\_ == '\_\_main\_\_':

# Vérifie si les clés ont déjà été générées

if 'public\_key' not in globals() or 'private\_key' not in globals():

# Si les clés n'ont pas été générées, génère une nouvelle paire de clés

public\_key, private\_key = generate\_keys(user\_response\_key\_size // 4)

else:

# Si les clés existent déjà, récupère les clés existantes

pass

# Mesure de l'utilisation de la mémoire après l'exécution du code RSA

after\_memory = get\_memory\_usage()

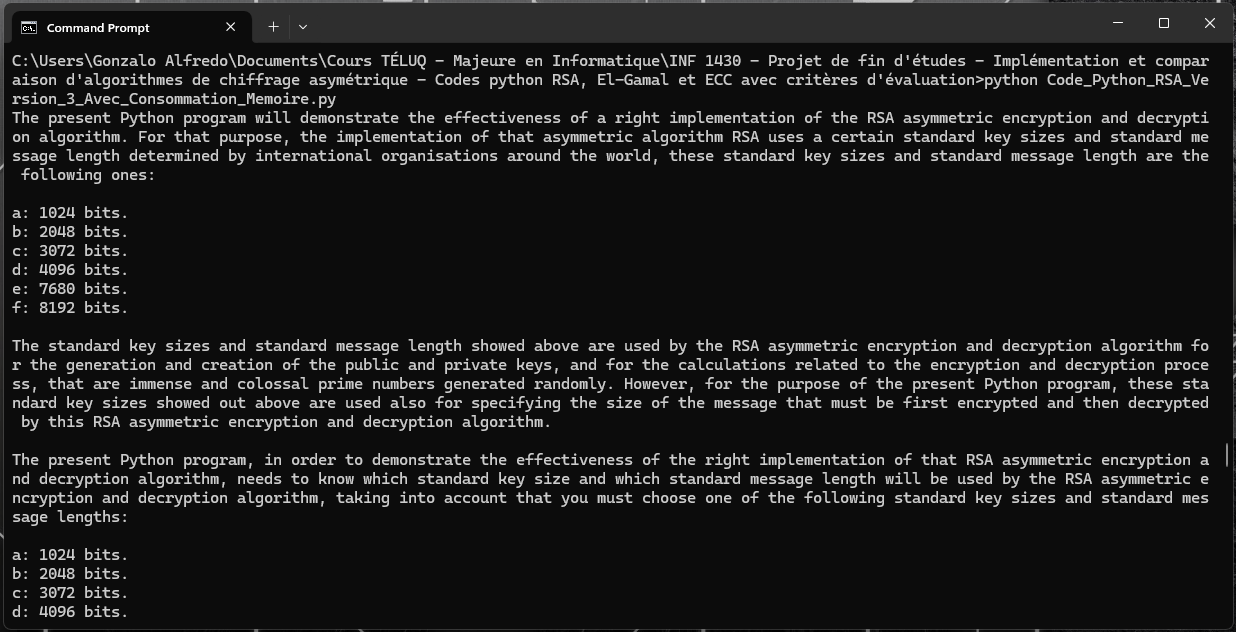
memory\_used = after\_memory - before\_memory

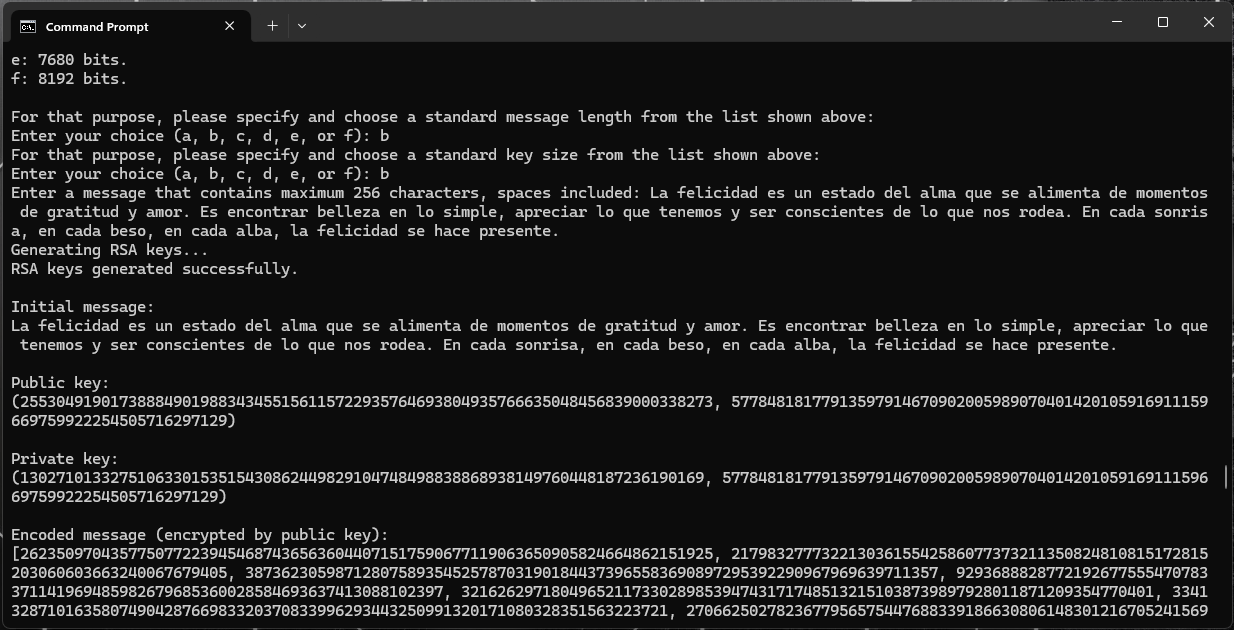
print("Memory used :", memory\_used, "kilobytes")

**5. Résumé des modifications :**

* Importation de la bibliothèque psutil pour mesurer l'utilisation de la mémoire.
* Ajout d'une fonction nommée get\_memory\_usage() pour mesurer l'utilisation de la mémoire avant et après l'exécution du code python RSA en version 3.
* Intégration de la mesure de l'utilisation de la mémoire dans le bloc principal du programme du code python RSA en version 3, avec une structure conditionnelle pour vérifier si les clés ont déjà été générées, afin d'afficher la quantité de mémoire utilisée après l'exécution du code RSA.

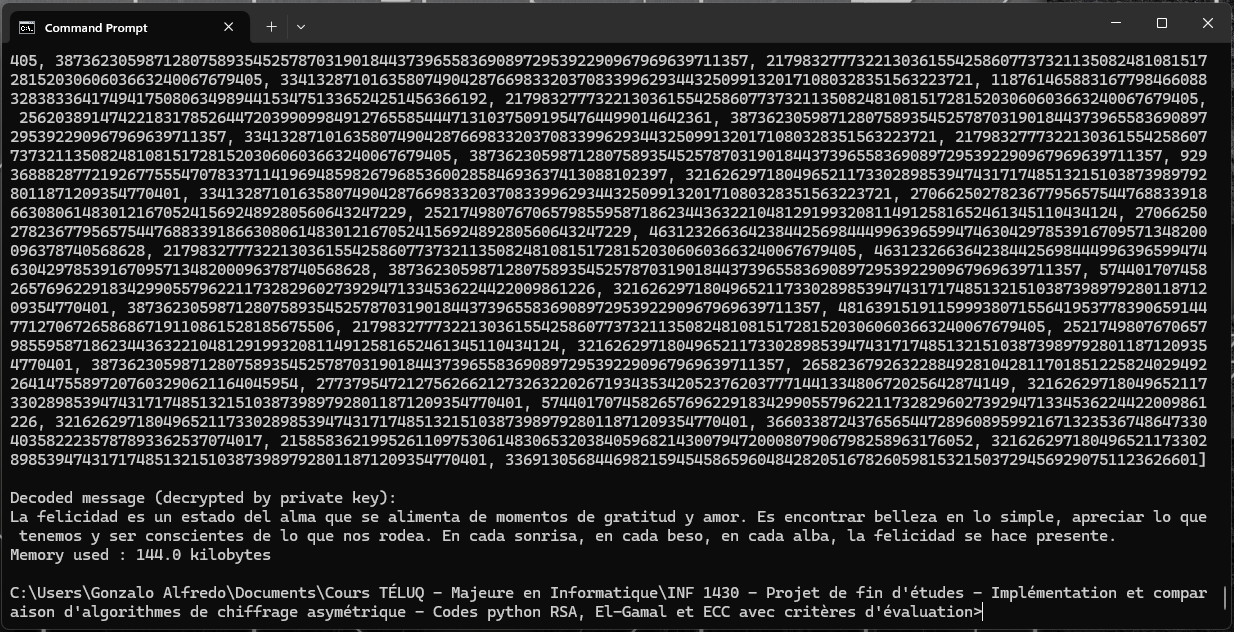
Ces modifications décrites dans le paragraphe montré ci-dessus ajoutent l’implémentation d’un nouveau critère d'évaluation pour mesurer l'efficacité de l'implémentation RSA en termes d’utilisation de la mémoire causée par différentes étapes du processus de génération de clés publiques et privées, celles du chiffrement et celles du déchiffrement de messages clairs et pleins. Les modifications décrites avec luxe de détails, et montrée dans les paragraphes ci-haut, ont donné de très bons résultats, car Gonzalo Alfredo Romero Francia peut maintenant mesurer, avec beaucoup de précision, le temps que prend son code python RSA en version 3, à générer des clés publique et privée, à chiffrer et à déchiffrer des messages clairs et pleins spécifié par lui, lors de la compilation de ce code python RSA. Le voici la sortie que donne le code python RSA implémentant le critère d’évaluation nommé « Utilisation de la mémoire » :











Les figures montrées ci-dessus montrent très bien que les modifications que Gonzalo Alfredo Romero Francia a faits, a permis à son code python RSA en version 3, d’implémenter le critère d’évaluation nommé « Utilisation de la mémoire », de manière très rapide, très efficace et très optimale, et produisant des résultats excellents, efficaces, optimales et remarquables.

**Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Taille des clés générées », ajoutée au main du code python RSA en version 3.**

En réalisant une forte quantité de recherches d’informations, Gonzalo Alfredo Romero Francia a réussi d’améliorer, de corriger, et d’optimiser son code python de base implémentant l’algorithme de chiffrement et de déchiffrement RSA, pour ensuite obtenir des versions de ce code python plus performant, plus robustes et plus structurés et mieux organisées, ce qui a déjà été réalisé au travail noté #2 d’INF 1430. Maintenant que l’auteur du présent document a en main son code python RSA en version 3, il est prêt à utiliser et mettre en pratique toutes les informations qu’il a trouvées en février dernier, lui enseignant comment implémenter, en langage python 3.12.1, le critère d’évaluation nommé « Taille des clés générées », ces informations qu’il a trouvées en février dernier, sont les suivantes :

* Pour mesurer la taille des clés générées dans le code python RSA en version 3, il faut simplement afficher la longueur en bits des clés publiques et privées une fois qu'elles ont été générées par les fonctions python generate\_keys().

En utilisant ces informations trouvées sur internet en février dernier, Gonzalo Alfredo Romero Francia a réussi à modifier et à obtenir le code python RSA en version 3 qui est montré à la section « Annexe » du présent document. Maintenant que l’auteur du présent document a son code python RSA en version 3 implémentant le critère d’évaluation nommé « Taille des clés générées », il a décidé de fournir les informations suivantes, expliquant comment il a réussi à implémenter, en langage python 3.12.1, le critère d’évaluation de ce code python nommé « Taille des clés générées » :

**Modifications apportées au code Python pour l'évaluation de la taille des clés générées**

**1. Importation des bibliothèques**

**Ajout de la bibliothèque nécessaire pour la gestion de la génération des clés :**

**Premier code :**

import random

**Deuxième code :**

import random

**2. Fonction de décodage de message**

**Modification spécifique pour la taille des clés générées dans la fonction de décodage, en lien avec le changement de la fonction chr() par la fonction str() :**

**Premier code :**

def decode\_message(encoded\_message):

return ''.join(chr(char) for char in encoded\_message)

**Deuxième code :**

def decode\_message(encoded\_message):

return ''.join(str(char) for char in encoded\_message)

**3. Mesure de la taille des clés générées**

**Ajout de l'évaluation et de l'affichage de la taille des clés générées dans le processus RSA :**

**Deuxième code (section ajoutée) :**

if \_\_name\_\_ == '\_\_main\_\_':

# Code block that measures the size of the keys generated by the python code implementing the RSA algorithm

# Check if keys have already been generated

if 'public\_key' not in globals() or 'private\_key' not in globals():

# If keys have not been generated, generate a new key pair

public\_key, private\_key = generate\_keys(user\_response\_key\_size // 4)

else:

# If keys already exist, retrieve the existing keys

pass

# Calculate the size of the public and private keys in bits

public\_key\_size = math.log2(public\_key[1])

private\_key\_size = math.log2(private\_key[1])

print("\nThe generated public key (size in bits):")

print(public\_key, "(", public\_key\_size, "bits)")

print("\nThe generated private key (size in bits):")

print(private\_key, "(", private\_key\_size, "bits)")

**4. Position du code de génération des clés et de chiffrement/déchiffrement**

**Déplacement du code de génération des clés et de chiffrement/déchiffrement à l'intérieur du bloc if \_\_name\_\_ == '\_\_main\_\_' :**

**Premier code :**

print("Generating RSA keys...")

public\_key, private\_key = generate\_keys(user\_response\_key\_size // 4)

print("RSA keys generated successfully.")

message = msg

encoded\_message = encode\_message(message)

encrypted\_message = [encrypt(char, public\_key) for char in encoded\_message]

decrypted\_message = [decrypt(char, private\_key) for char in encrypted\_message]

decoded\_message = decode\_message(decrypted\_message)

print("\nInitial message:")

print(message)

print("\nPublic key:")

print(public\_key)

print("\nPrivate key:")

print(private\_key)

print("\nEncoded message (encrypted by public key):")

print(encrypted\_message)

print("\nDecoded message (decrypted by private key):")

print(decoded\_message)

**Deuxième code :**

if \_\_name\_\_ == '\_\_main\_\_':

# Code block that measures the size of the keys generated by the python code implementing the RSA algorithm

# Check if keys have already been generated

if 'public\_key' not in globals() or 'private\_key' not in globals():

# If keys have not been generated, generate a new key pair

public\_key, private\_key = generate\_keys(user\_response\_key\_size // 4)

else:

# If keys already exist, retrieve the existing keys

pass

# Calculate the size of the public and private keys in bits

public\_key\_size = math.log2(public\_key[1])

private\_key\_size = math.log2(private\_key[1])

print("\nThe generated public key (size in bits):")

print(public\_key, "(", public\_key\_size, "bits)")

print("\nThe generated private key (size in bits):")

print(private\_key, "(", private\_key\_size, "bits)")

**5. Affichage des tailles des clés**

**Ajout des impressions pour afficher les tailles des clés générées :**

**Deuxième code (section ajoutée) :**

# Calculate the size of the public and private keys in bits

public\_key\_size = len(str(public\_key))

private\_key\_size = len(str(private\_key))

print("\nThe generated public key (size in bits):")

print(public\_key, "(", public\_key\_size, "bits)")

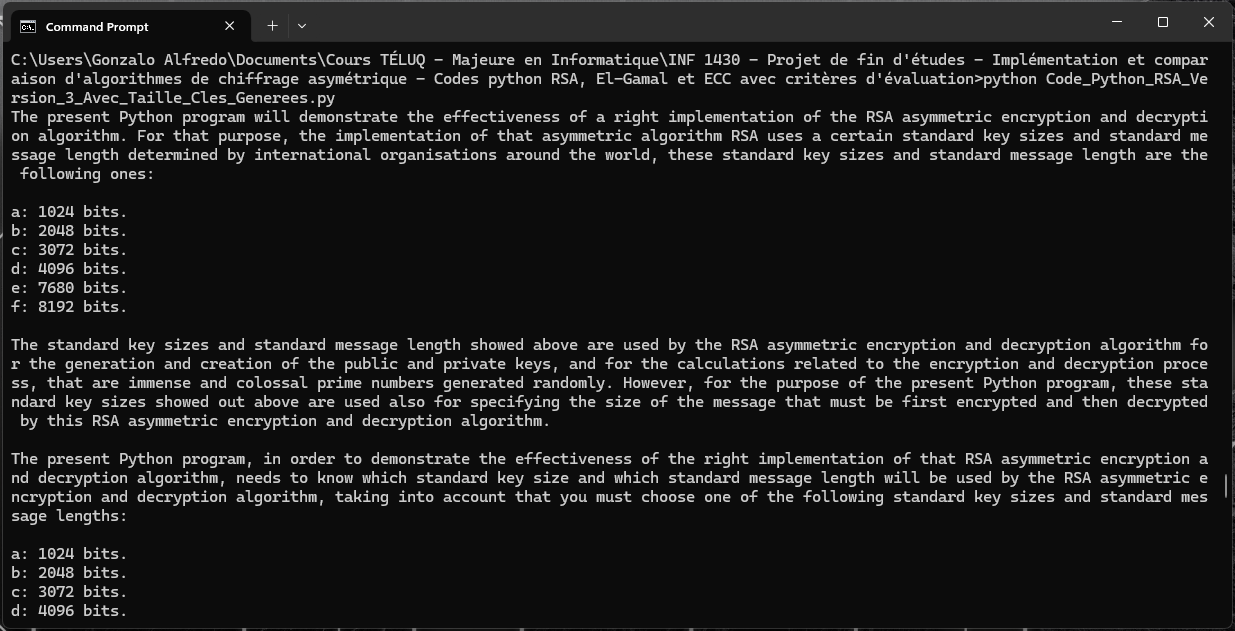
print("\nThe generated private key (size in bits):")

print(private\_key, "(", private\_key\_size, "bits)")

**6. Résumé des modifications :**

* Importation des bibliothèques : Pas de modification particulière pour ce critère.
* Fonction de décodage de message : Pas de modification spécifique pour ce critère.
* Ajout de mesures de la taille des clés générées : Pour évaluer et afficher les tailles des clés RSA générées, ce qui permet de vérifier si la taille des clés est conforme aux attentes et aux exigences de sécurité.
* Déplacement du code de génération des clés et de chiffrement/déchiffrement : À l'intérieur du bloc if \_\_name\_\_ == '\_\_main\_\_' pour organiser le flux principal du programme et permettre une exécution correcte du script principal.
* Affichage des tailles des clés : Pour fournir une évaluation claire et détaillée des tailles des clés RSA générées.

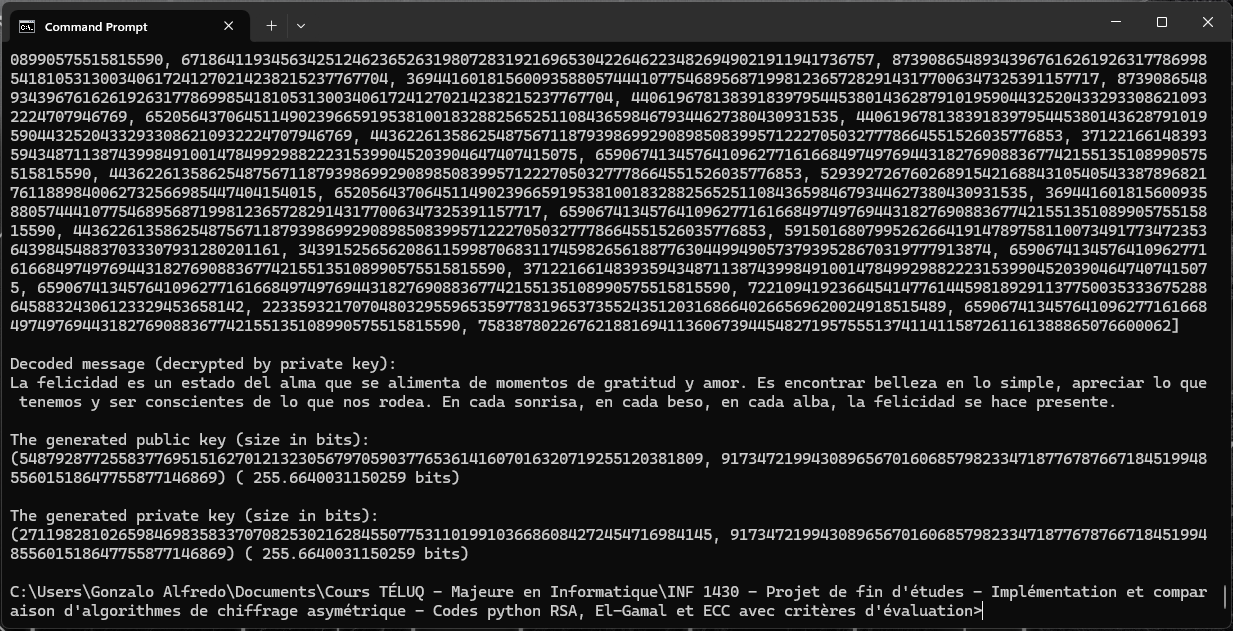
Ces modifications décrites dans le paragraphe montré ci-dessus ajoutent l’implémentation d’un nouveau critère d'évaluation pour mesurer l'efficacité de l'implémentation RSA en termes de tailles de clés générées par différents calculs arithmétiques, mathématiques et informatiques effectués par la fonction generate\_keys(), du processus de génération de clés publiques et privées RSA, pour ensuite réaliser tous les calculs mathématiques, arithmétiques et informatiques en lien aux processus de chiffrement et de déchiffrement de messages clairs et pleins. Les modifications décrites avec luxe de détails, et montrée dans les paragraphes ci-haut, ont donné de très bons résultats, car Gonzalo Alfredo Romero Francia peut maintenant mesurer, avec beaucoup de précision, le temps que prend son code python RSA en version 3, à générer des clés publique et privée, à chiffrer et à déchiffrer des messages clairs et pleins spécifié par lui, lors de la compilation de ce code python RSA. Le voici la sortie que donne le code python RSA implémentant le critère d’évaluation nommé « Taille des clés générées » :











Les figures montrées ci-dessus montrent très bien que les modifications que Gonzalo Alfredo Romero Francia a faits, a permis à son code python RSA en version 3, d’implémenter le critère d’évaluation nommé « Taille des clés générées », de manière très rapide, très efficace et très optimale, et produisant des résultats excellents, efficaces, optimales et remarquables.

**Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Complexité algorithmique », ajoutée au main du code python RSA en version 3.**

En réalisant une forte quantité de recherches d’informations, Gonzalo Alfredo Romero Francia a réussi d’améliorer, de corriger, et d’optimiser son code python de base implémentant l’algorithme de chiffrement et de déchiffrement RSA, pour ensuite obtenir des versions de ce code python plus performant, plus robustes et plus structurés et mieux organisées, ce qui a déjà été réalisé au travail noté #2 d’INF 1430. Maintenant que l’auteur du présent document a en main son code python RSA en version 3, il est prêt à utiliser et mettre en pratique toutes les informations qu’il a trouvées en février dernier, lui enseignant comment implémenter, en langage python 3.12.1, le critère d’évaluation nommé « Complexité Algorithmique », ces informations qu’il a trouvées en février dernier, sont les suivantes :

* La complexité algorithmique de votre implémentation RSA dépend principalement des opérations de génération de clés, de chiffrement et de déchiffrement. Voici comment on peut évaluer la complexité algorithmique de ces parties du code python RSA en version 3 :
* Complexité de la génération de clés (setkeys) :
* La génération de clés dans le code python RSA en version 3 implique de choisir deux nombres premiers aléatoires et de calculer les clés publique et privée.
* La complexité de cette opération dépend principalement du temps nécessaire pour trouver deux nombres premiers de grande taille et pour calculer les clés publique et privée en fonction de ces nombres.
* On peut évaluer cette complexité en mesurant le temps d'exécution de la fonction setkeys() pour différentes tailles de clés.
* Complexité du chiffrement (encrypt) :
* Le chiffrement dans le code python RSA en version 3 implique de calculer la puissance modulaire d'un nombre (le message) avec la clé publique.
* La complexité de cette opération dépend principalement de la taille du message et de la taille de la clé publique.
* On peut évaluer cette complexité en mesurant le temps d'exécution de la fonction encrypt() pour différentes tailles de messages et de clés publiques.
* Complexité du déchiffrement (decrypt) :
* Le déchiffrement dans le code python RSA en version 3 implique de calculer la puissance modulaire d'un nombre (le texte chiffré) avec la clé privée.
* La complexité de cette opération dépend principalement de la taille du texte chiffré et de la taille de la clé privée.
* On peut évaluer cette complexité en mesurant le temps d'exécution de la fonction decrypt() pour différentes tailles de texte chiffré et de clés privées.
* En mesurant le temps d'exécution de ces opérations pour différentes tailles de données et de clés, on peut obtenir des informations sur la complexité algorithmique de votre implémentation RSA.

En utilisant ces informations trouvées sur internet en février dernier, Gonzalo Alfredo Romero Francia a réussi à modifier et à obtenir le code python RSA en version 3 qui est montré à la section « Annexe » du présent document. Maintenant que l’auteur du présent document a son code python RSA en version 3 implémentant le critère d’évaluation nommé « Complexité Algorithmique », il a décidé de fournir les informations suivantes, expliquant comment il a réussi à implémenter, en langage python 3.12.1, le critère d’évaluation de ce code python nommé « Complexité Algorithmique » :

**Modifications apportées au code Python pour l'évaluation de la Complexité Algorithmique**

**1. Importation des bibliothèques**

**Ajout des bibliothèques psutil et time pour mesurer la consommation de mémoire et le temps nécessaire à l'exécution des différentes étapes du processus RSA :**

**Premier code :**

import random

Deuxième code :

import random

import psutil

import time

import math

**2. Fonction pour mesurer l'utilisation de la mémoire**

**Ajout de la fonction get\_memory\_usage pour mesurer l'utilisation de la mémoire en kilobytes :**

**Deuxième code (section ajoutée) :**

def get\_memory\_usage():

process = psutil.Process()

return process.memory\_info().rss / 1024 # Convert to kilobytes

**3. Mesure de la complexité algorithmique**

**Ajout des fonctions pour mesurer la complexité algorithmique (temps et mémoire) de la génération de clés, du chiffrement et du déchiffrement :**

**Deuxième code (section ajoutée) :**

def measure\_key\_generation(key\_length):

start\_memory = get\_memory\_usage()

start\_time = time.time()

generate\_keys(key\_length // 4)

end\_time = time.time()

end\_memory = get\_memory\_usage()

return end\_time - start\_time, end\_memory - start\_memory

def measure\_encryption(message, public\_key):

start\_time = time.time()

encrypt(message, public\_key)

end\_time = time.time()

return end\_time - start\_time

def measure\_decryption(encrypted\_message, private\_key):

start\_time = time.time()

decrypt(encrypted\_message, private\_key)

end\_time = time.time()

return end\_time - start\_time

**4. Bloc principal de mesure de la complexité**

**Déplacement du code de génération des clés et de chiffrement/déchiffrement à l'intérieur du bloc if \_\_name\_\_ == '\_\_main\_\_', avec des mesures de temps et de mémoire pour chaque étape :**

**Premier code :**

print("Generating RSA keys...")

public\_key, private\_key = generate\_keys(user\_response\_key\_size // 4)

print("RSA keys generated successfully.")

message = msg

encoded\_message = encode\_message(message)

encrypted\_message = [encrypt(char, public\_key) for char in encoded\_message]

decrypted\_message = [decrypt(char, private\_key) for char in encrypted\_message]

decoded\_message = decode\_message(decrypted\_message)

print("\nInitial message:")

print(message)

print("\nPublic key:")

print(public\_key)

print("\nPrivate key:")

print(private\_key)

print("\nEncoded message (encrypted by public key):")

print(encrypted\_message)

print("\nDecoded message (decrypted by private key):")

print(decoded\_message)

**Deuxième code :**

if \_\_name\_\_ == '\_\_main\_\_':

# Code block that measures the algorithmic complexity of the keys generated and the encryption and decryption process provided by the Python code implementing the RSA algorithm

if 'public\_key' not in globals() or 'private\_key' not in globals():

public\_key, private\_key = generate\_keys(user\_response\_key\_size // 4)

else:

pass

if 'public\_key' in globals() and 'private\_key' in globals():

key\_gen\_time, key\_gen\_memory = measure\_key\_generation(user\_response\_key\_size)

key\_gen\_time\_str = "{:.8f}".format(max(0.00000001, key\_gen\_time))

key\_gen\_memory\_str = "{:.8f}".format(max(0.00000001, key\_gen\_memory))

print("\nComplexity of key generation:")

print("Time taken:", key\_gen\_time\_str, "seconds")

print("Memory used:", key\_gen\_memory\_str, "KB")

print("\nMeasuring encryption complexity...")

encrypted\_time = measure\_encryption(123456, public\_key)

encryption\_memory\_used = get\_memory\_usage()

encrypted\_time\_str = "{:.8f}".format(max(0.00000001, encrypted\_time))

print("Encryption time:", encrypted\_time\_str, "seconds")

print("Memory used during encryption:", "{:.8f}".format(encryption\_memory\_used), "KB")

print("\nMeasuring decryption complexity...")

decrypted\_time = measure\_decryption(encrypted\_time, private\_key)

decryption\_memory\_used = get\_memory\_usage()

decrypted\_time\_str = "{:.8f}".format(max(0.00000001, decrypted\_time))

print("Decryption time:", decrypted\_time\_str, "seconds")

print("Memory used during decryption:", "{:.8f}".format(decryption\_memory\_used), "KB")

**5. Affichage des résultats de la complexité algorithmique**

**Ajout des impressions pour afficher les résultats des mesures de complexité pour chaque étape (génération des clés, chiffrement, et déchiffrement) :**

**Deuxième code (section ajoutée) :**

# Print/display the complexity of key generation

print("\nComplexity of key generation:")

print("Time taken:", key\_gen\_time\_str, "seconds")

print("Memory used:", key\_gen\_memory\_str, "KB")

# Print/display the complexity of encryption

print("\nEncryption time:", encrypted\_time\_str, "seconds")

print("Memory used during encryption:", "{:.8f}".format(encryption\_memory\_used), "KB")

# Print/display the complexity of decryption

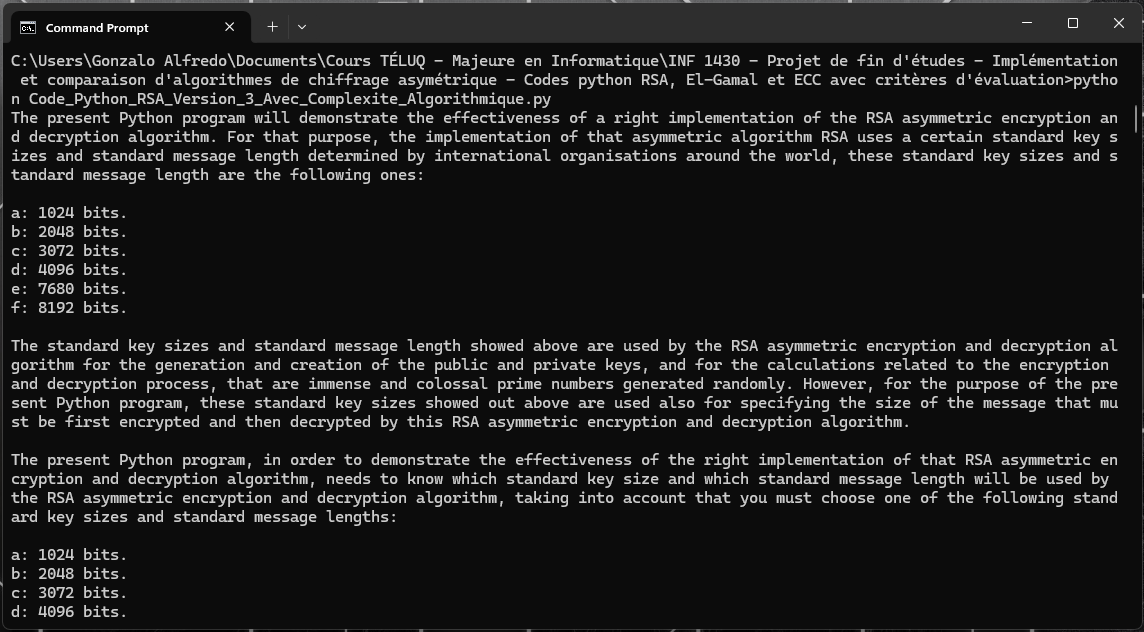
print("\nDecryption time:", decrypted\_time\_str, "seconds")

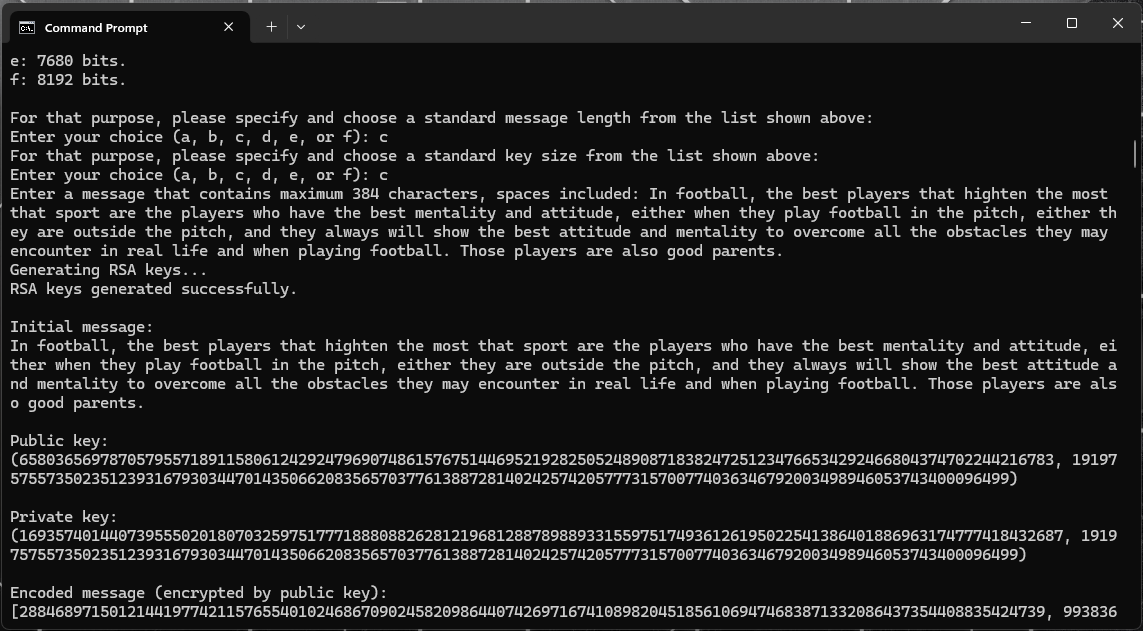
print("Memory used during decryption:", "{:.8f}".format(decryption\_memory\_used), "KB")

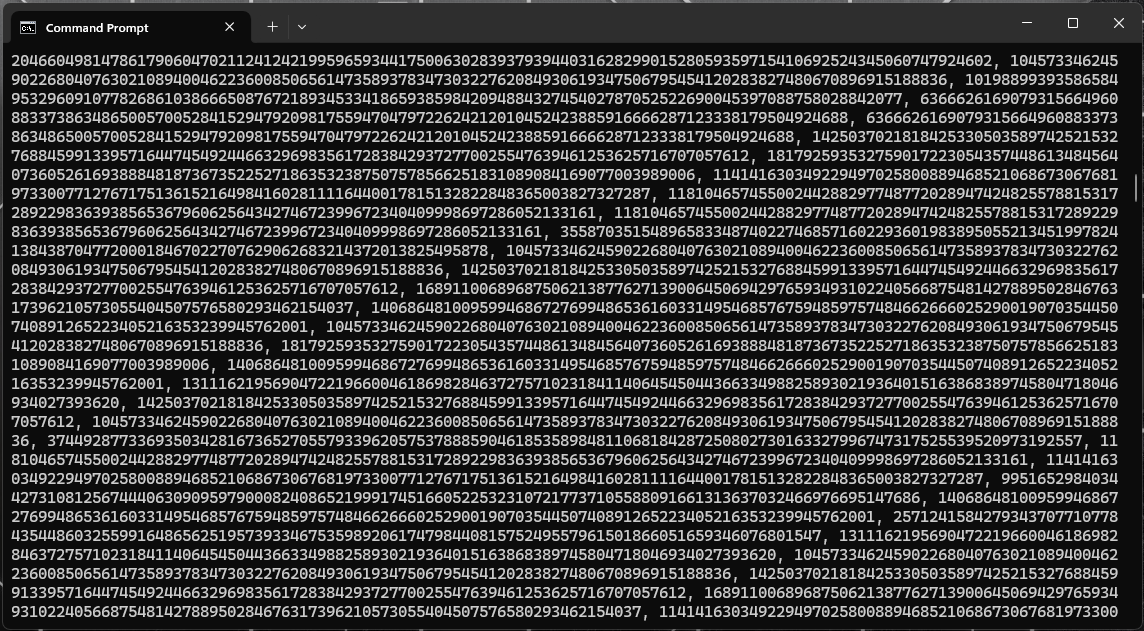
**Résumé des modifications :**

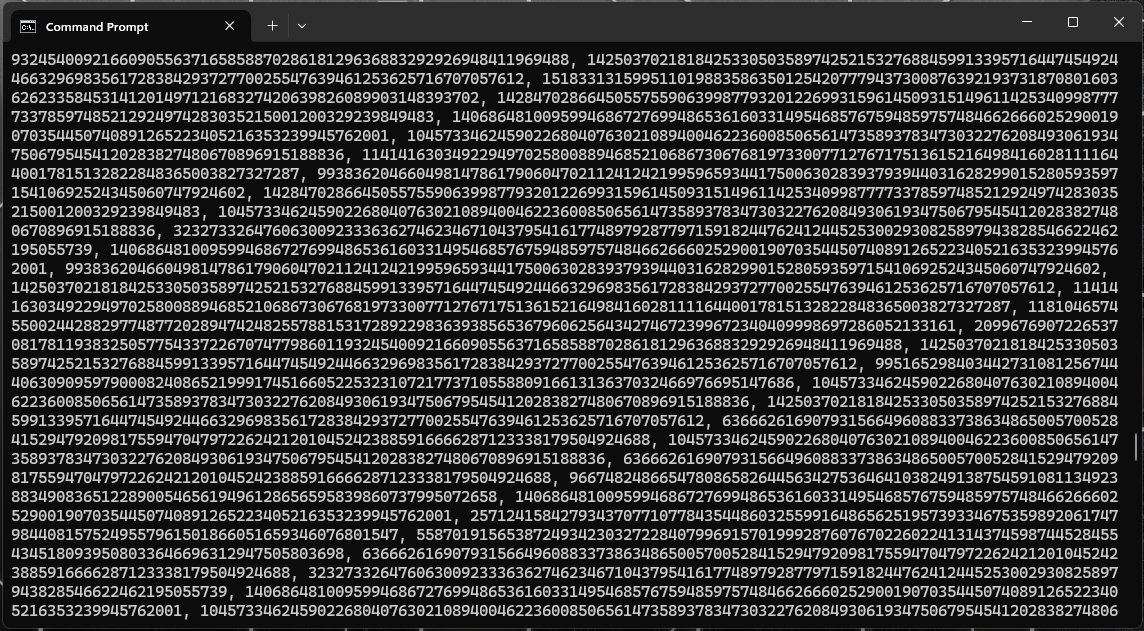
* Importation des librairies psutil et time pour mesurer la consommation de mémoire et le temps d'exécution des étapes du processus RSA.
* Ajout de la fonction get\_memory\_usage pour mesurer la mémoire utilisée par le processus en cours.
* Ajout de fonctions pour mesurer la complexité algorithmique de la génération des clés, du chiffrement et du déchiffrement, en termes de temps et de mémoire.
* Déplacement du code de génération des clés et de chiffrement/déchiffrement à l'intérieur du bloc principal if \_\_name\_\_ == '\_\_main\_\_', avec des mesures pour chaque étape.
* Ajout des impressions pour afficher les résultats des mesures de complexité, incluant le temps et la mémoire utilisés pour chaque étape du processus RSA.

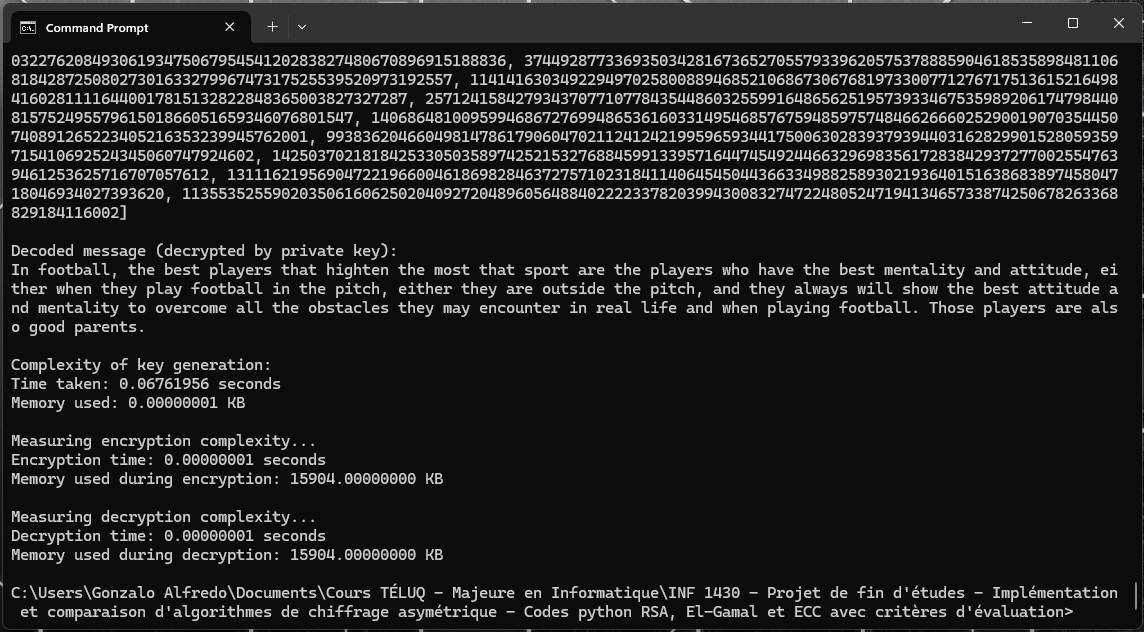
Ces modifications décrites dans le paragraphe montré ci-dessus ajoutent l’implémentation d’un nouveau critère d'évaluation pour mesurer l'efficacité de l'implémentation RSA en termes de complexité algorithmique de type temporelle (en unités de temps de type millisecondes) et en termes d’utilisation de la mémoire (en unités d’espace mémoire de type KB), causée par différentes étapes du processus de génération de clés publiques et privées, celles du chiffrement et celles du déchiffrement de messages clairs et pleins. Les modifications décrites avec luxe de détails, et montrée dans les paragraphes ci-haut, ont donné de très bons résultats, car Gonzalo Alfredo Romero Francia peut maintenant mesurer, avec beaucoup de précision, le temps que prend son code python RSA en version 3, à générer des clés publique et privée, à chiffrer et à déchiffrer des messages clairs et pleins spécifié par lui, lors de la compilation de ce code python RSA. Le voici la sortie que donne le code python RSA implémentant le critère d’évaluation nommé « Complexité Algorithmique » :











Les figures montrées ci-dessus montrent très bien que les modifications que Gonzalo Alfredo Romero Francia a faits, a permis à son code python RSA en version 3, d’implémenter le critère d’évaluation nommé « Complexité Algorithmique », de manière très rapide, très efficace et très optimale, et produisant des résultats excellents, efficaces, optimales et remarquables.

**Analyse complète de la nature et de la qualité d’implémentation du critère d’évaluation nommé « Gestion de grandes quantités de données », ajoutée au main du code python RSA en version 3.**

En réalisant une forte quantité de recherches d’informations, Gonzalo Alfredo Romero Francia a réussi d’améliorer, de corriger, et d’optimiser son code python de base implémentant l’algorithme de chiffrement et de déchiffrement RSA, pour ensuite obtenir des versions de ce code python plus performant, plus robustes et plus structurés et mieux organisées, ce qui a déjà été réalisé au travail noté #2 d’INF 1430. Maintenant que l’auteur du présent document a en main son code python RSA en version 3, il est prêt à utiliser et mettre en pratique toutes les informations qu’il a trouvées en février dernier, lui enseignant comment implémenter, en langage python 3.12.1, le critère d’évaluation nommé « Complexité Algorithmique », ces informations qu’il a trouvées en février dernier, sont les suivantes :

* Pour évaluer la gestion des grandes quantités de données dans votre implémentation RSA, vous pouvez générer des messages de tailles variées et mesurer le temps nécessaire pour chiffrer et déchiffrer ces messages.
* Créer une fonction qui mesure de la complexité de l'opération de chiffrement pour un message donné.
* Créer une fonction qui mesure de la complexité de l'opération de déchiffrement pour un message donné.
* Créer une ligne de code python qui mesure de la complexité de la génération de clés
* Créer des lignes de code python qui implémentent le chiffrement et le déchiffrement d’un message clair et plein.
* Créer une ligne de code python qui mesure de la complexité de l'opération de chiffrement.
* Créer une ligne de code python qui #mesure de la complexité de l'opération de déchiffrement.

En utilisant ces informations trouvées sur internet en février dernier, Gonzalo Alfredo Romero Francia a réussi à modifier et à obtenir le code python RSA en version 3 qui est montré à la section « Annexe » du présent document. Maintenant que l’auteur du présent document a son code python RSA en version 3 implémentant le critère d’évaluation nommé « Gestion de grandes quantités de données », il a décidé de fournir les informations suivantes, expliquant comment il a réussi à implémenter, en langage python 3.12.1, le critère d’évaluation de ce code python nommé « Gestion de grandes quantités de données » :

**Modifications apportées au code Python pour l'évaluation de la gestion de grandes quantités de données**

**1. Importation des bibliothèques**

**Ajout de la bibliothèque time :**

**Premier code :**

import random

**Deuxième code :**

import random

import time

**2. Fonction de décodage de message**

**Modification de la fonction decode\_message pour retourner une chaîne de caractères numériques au lieu des caractères ASCII :**

**Premier code :**

def decode\_message(encoded\_message):

return ''.join(chr(char) for char in encoded\_message)

**Deuxième code :**

def decode\_message(encoded\_message):

return ''.join(str(char) for char in encoded\_message)

**3. Position du code de génération des clés et de chiffrement/déchiffrement**

**Déplacement du code de génération des clés et de chiffrement/déchiffrement à l'intérieur du bloc if \_\_name\_\_ == '\_\_main\_\_' :**

**Premier code :**

print("Generating RSA keys...")

public\_key, private\_key = generate\_keys(user\_response\_key\_size // 4)

print("RSA keys generated successfully.")

message = msg

encoded\_message = encode\_message(message)

encrypted\_message = [encrypt(char, public\_key) for char in encoded\_message]

decrypted\_message = [decrypt(char, private\_key) for char in encrypted\_message]

decoded\_message = decode\_message(decrypted\_message)

print("\nInitial message:")

print(message)

print("\nPublic key:")

print(public\_key)

print("\nPrivate key:")

print(private\_key)

print("\nEncoded message (encrypted by public key):")

print(encrypted\_message)

print("\nDecoded message (decrypted by private key):")

print(decoded\_message)

**Deuxième code :**

if \_\_name\_\_ == '\_\_main\_\_':

# Code block that measures all the execution times associated to the RSA algorithm

...

print("Generating RSA keys...")

public\_key, private\_key = generate\_keys(user\_response\_key\_size // 4)

print("RSA keys generated successfully.")

message = msg

encoded\_message = encode\_message(message)

encrypted\_message = [encrypt(char, public\_key) for char in encoded\_message]

decrypted\_message = [decrypt(char, private\_key) for char in encrypted\_message]

decoded\_message = decode\_message(decrypted\_message)

print("\nInitial message:")

print(message)

print("\nPublic key:")

print(public\_key)

print("\nPrivate key:")

print(private\_key)

print("\nEncoded message (encrypted by public key):")

print(encrypted\_message)

print("\nDecoded message (decrypted by private key):")

print(decoded\_message)

**4. Boucle for pour mesurer le temps d'encryptage et le temps d’encodage**

**La boucle suivante mesure le temps d'encodage du message et la quantité de mémoire utilisée :**

# Measure encryption complexity for each test message

print("\nMeasuring encryption complexity for each test message...")

encryption\_times = [] # Store encryption times for each message

encrypted\_messages = [] # Store encrypted messages for each message

encoding\_memory\_used = [] # Store memory usage during encoding for each message

encryption\_memory\_used\_list = [] # Store memory usage during encryption for each message

for i, message in enumerate(test\_messages):

# Encode the message

encoded\_message = encode\_message(message)

# Measure memory usage during encoding

encoding\_memory\_used\_value = measure\_memory\_usage()

encoding\_memory\_used.append(encoding\_memory\_used\_value)

# Encrypt the message

encryption\_start\_time = time.time()

encrypted\_message = [encrypt(char, public\_key) for char in encoded\_message]

encryption\_end\_time = time.time()

encryption\_time = encryption\_end\_time - encryption\_start\_time

encryption\_times.append(encryption\_time)

encrypted\_messages.append(encrypted\_message)

# Measure memory usage during encryption

encryption\_memory\_used\_value = measure\_memory\_usage()

encryption\_memory\_used\_list.append(encryption\_memory\_used\_value)

# Display information

print(f"Message {i+1}:")

print("Clear message:", message)

print("Encrypted message:", encrypted\_message)

print(f"Encryption time for message {i+1}: {encryption\_time:.8f} seconds")

print("Memory used during encoding:", "{:.8f}".format(encoding\_memory\_used\_value), "KB")

print("Memory used during encryption:", "{:.8f}".format(encryption\_memory\_used\_value), "KB")

**Lignes de code associées (calcul du temps moyen d’encryptage et de encodage, et la quantité moyenne de mémoire utilisée lors du encryptage et lors du encodage de messages):**

# Calculate average encryption time

average\_encryption\_time = sum(encryption\_times) / len(encryption\_times)

print("\nAverage encryption time:", "{:.8f}".format(average\_encryption\_time), "seconds")

# Calculate average encoding time

average\_encoding\_time = sum(encryption\_times) / len(encryption\_times)

print("Average encoding time:", "{:.8f}".format(average\_encoding\_time), "seconds")

# Calculate average memory used during encryption

average\_encryption\_memory = sum(encryption\_memory\_used\_list) / len(encryption\_memory\_used\_list)

print("Average memory used during encryption:", "{:.8f}".format(average\_encryption\_memory), "KB")

# Calculate average memory used during encoding

average\_encoding\_memory = sum(encoding\_memory\_used) / len(encoding\_memory\_used)

print("Average memory used during encoding:", "{:.8f}".format(average\_encoding\_memory), "KB")

# Memory usage during encoding

encoding\_memory = sys.getsizeof(encoded\_message)

**5. Boucle for pour mesurer le temps de décryptage et le temps de décodage**

**La boucle suivante mesure le temps de décodage du message et la quantité de mémoire utilisée :**

# Measure decryption complexity for each test message

print("\nMeasuring decryption complexity for each test message...")

decryption\_times = [] # Store decryption times for each message

decrypted\_messages = [] # Store decrypted messages for each message

decoding\_memory\_used = [] # Store memory usage during decoding for each message

decryption\_memory\_used\_list = [] # Store memory usage during decryption for each message

for i, encrypted\_message in enumerate(encrypted\_messages):

# Decrypt the message

decryption\_start\_time = time.time()

decrypted\_message = [decrypt(char, private\_key) for char in encrypted\_message]

decryption\_end\_time = time.time()

decryption\_time = decryption\_end\_time - decryption\_start\_time

decryption\_times.append(decryption\_time)

decrypted\_messages.append(decrypted\_message)

# Measure memory usage during decryption

decryption\_memory\_used\_value = measure\_memory\_usage()

decryption\_memory\_used\_list.append(decryption\_memory\_used\_value)

# Measure memory usage during decoding

decoding\_memory\_used\_value = measure\_memory\_usage()

decoding\_memory\_used.append(decoding\_memory\_used\_value)

# Display information

print(f"Message {i+1}:")

print("Encrypted message:", encrypted\_message)

decrypted\_message\_text = ''.join([chr(char) if char < 1114111 else '?' for char in decrypted\_message])

print("Decrypted message:", decrypted\_message\_text)

print(f"Decryption time for message {i+1}: {decryption\_time:.8f} seconds")

print("Memory used during decryption:", "{:.8f}".format(decryption\_memory\_used\_value), "KB")

print("Memory used during decoding:", "{:.8f}".format(decoding\_memory\_used\_value), "KB")

Lignes de code associées **(calcul du temps moyen de décryptage et de décodage, et la quantité moyenne de mémoire utilisée lors du décryptage et lors du décodage de messages)** :

# Calculate average decryption time

average\_decryption\_time = sum(decryption\_times) / len(decryption\_times)

print("\nAverage decryption time:", "{:.8f}".format(average\_decryption\_time), "seconds")

# Calculate average decoding time

average\_decoding\_time = sum(decryption\_times) / len(decryption\_times)

print("Average decoding time:", "{:.8f}".format(average\_decoding\_time), "seconds")

# Calculate average memory used during decryption

average\_decryption\_memory = sum(decryption\_memory\_used\_list) / len(decryption\_memory\_used\_list)

print("Average memory used during decryption:", "{:.8f}".format(average\_decryption\_memory), "KB")

# Calculate average memory used during decoding

average\_decoding\_memory = sum(decoding\_memory\_used) / len(decoding\_memory\_used)

print("Average memory used during decoding:", "{:.8f}".format(average\_decoding\_memory), "KB")

**6. Affichage des temps d'exécution et des quantités de mémoire utilisées, pour les 5 messages clairs et pleins générées par les fonctions generate\_message() et generate\_test\_messages()**

**Ajout des impressions pour afficher les temps d'exécution pour chaque étape (génération de clés, chiffrement, déchiffrement encodage et décodage), les quantités de mémoire utilisées pendant le chiffrement, le déchiffrement, l'encodage et le décodage, et le temps total :**

# Generate a plaintext message

plaintext\_message = "The challenges facing humanity today are numerous and complex. From climate change to global poverty, to social inequalities and the rise of terrorism, our world is confronted with urgent problems that require immediate and concerted action. To overcome these challenges, it is essential that we work together, as a global population, to find sustainable solutions. This requires a firm commitment to international cooperation, as well as concrete policies and actions to promote sustainable development, equality, and peace. By joining forces and pooling our resources, humanity can create a better future for all, where each individual has the opportunity to realize their full potential and live in a safe, just, and prosperous world. However, for the nations of the world to become rich and prosperous, global poverty must be eradicated for real."

# Use plain text message to generate test messages

test\_messages = generate\_test\_messages(plaintext\_message, 5)

# Measure memory usage for generating test messages

memory\_used = measure\_memory\_usage()

print("\nMemory used for generating test messages:", "{:.8f}".format(memory\_used), "KB")

//Gros blocs de code en lien aux deux boucles déjà décrites aux points 4 et 5!

//….

//….

//….

//….

# Generate test message for encoding and decoding

test\_message = generate\_message(user\_response\_message\_length)

# Measure encoding complexity

print("\nMeasuring encoding complexity...")

encoding\_start\_time = time.time()

encoded\_message = encode\_message(test\_message)

encoding\_end\_time = time.time()

encoding\_time = encoding\_end\_time - encoding\_start\_time

encoding\_memory\_used = measure\_memory\_usage() # Measure memory usage during encoding

print("Encoding time:", "{:.8f}".format(encoding\_time), "seconds")

print("Memory used during encoding:", "{:.8f}".format(encoding\_memory\_used), "KB")

# Measure decoding complexity

print("\nMeasuring decoding complexity...")

decoding\_start\_time = time.time()

decoded\_message = decode\_message(encoded\_message)

decoding\_end\_time = time.time()

decoding\_time = decoding\_end\_time - decoding\_start\_time

decoding\_memory\_used = measure\_memory\_usage() # Measure memory usage during decoding

print("Decoding time:", "{:.8f}".format(decoding\_time), "seconds")

print("Memory used during decoding:", "{:.8f}".format(decoding\_memory\_used), "KB")

# Measure encryption complexity

print("\nMeasuring encryption complexity...")

encryption\_start\_time = time.time()

encrypted\_message = [encrypt(char, public\_key) for char in encoded\_message]

encryption\_end\_time = time.time()

encryption\_time = encryption\_end\_time - encryption\_start\_time

encryption\_memory\_used = measure\_memory\_usage() # Measure memory usage during encryption

print("Encryption time:", "{:.8f}".format(encryption\_time), "seconds")

print("Memory used during encryption:", "{:.8f}".format(encryption\_memory\_used), "KB")

# Measure decryption complexity

print("\nMeasuring decryption complexity...")

decryption\_start\_time = time.time()

decrypted\_message = [decrypt(char, private\_key) for char in encrypted\_message]

decryption\_end\_time = time.time()

decryption\_time = decryption\_end\_time - decryption\_start\_time

decryption\_memory\_used = measure\_memory\_usage() # Measure memory usage during decryption

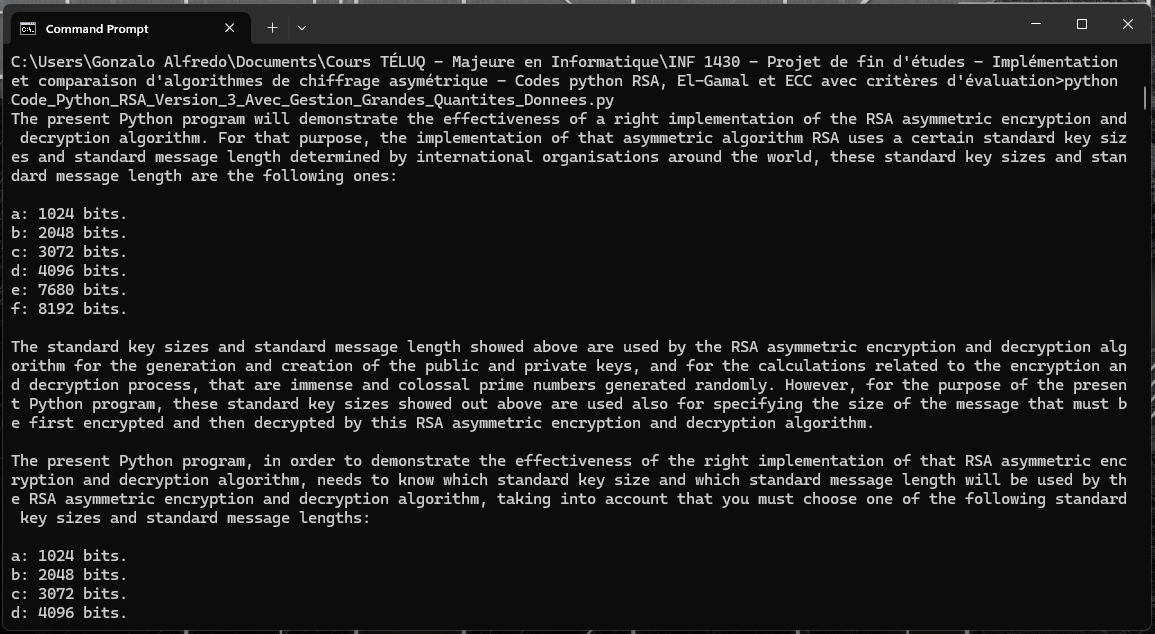
print("Decryption time:", "{:.8f}".format(decryption\_time), "seconds")

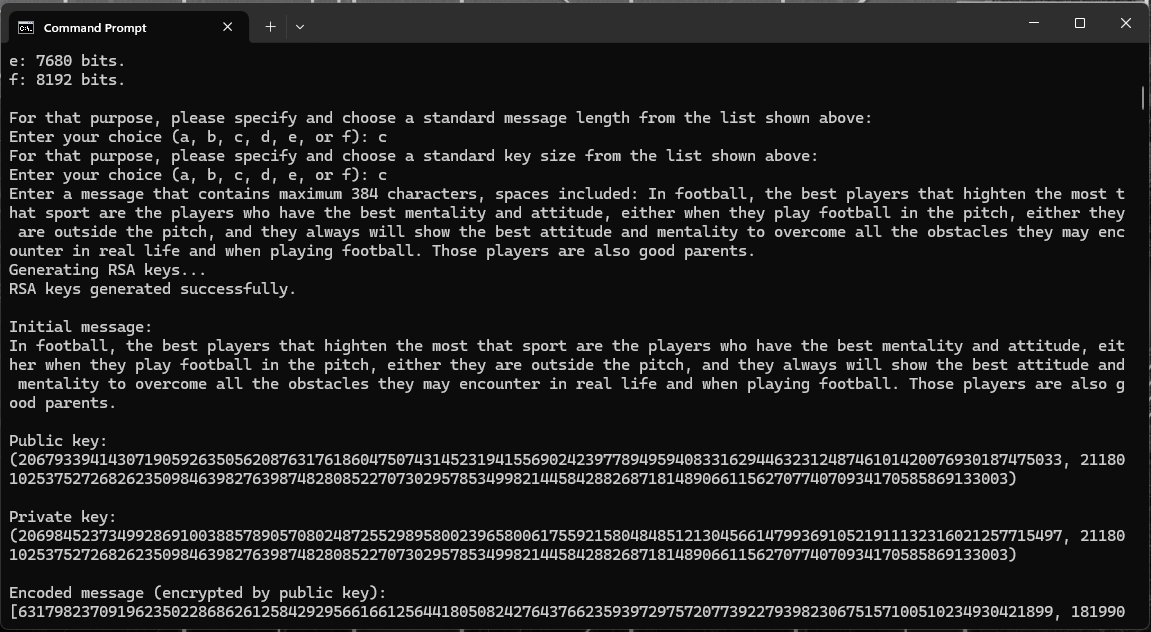
print("Memory used during decryption:", "{:.8f}".format(decryption\_memory\_used), "KB")

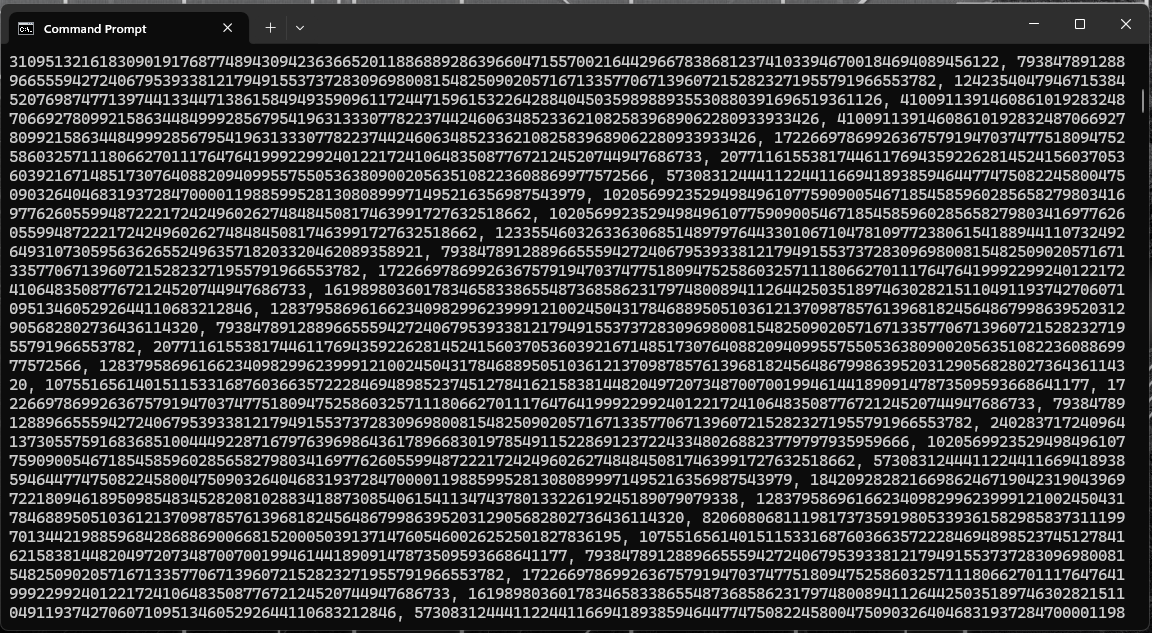
**Résumé des modifications :**

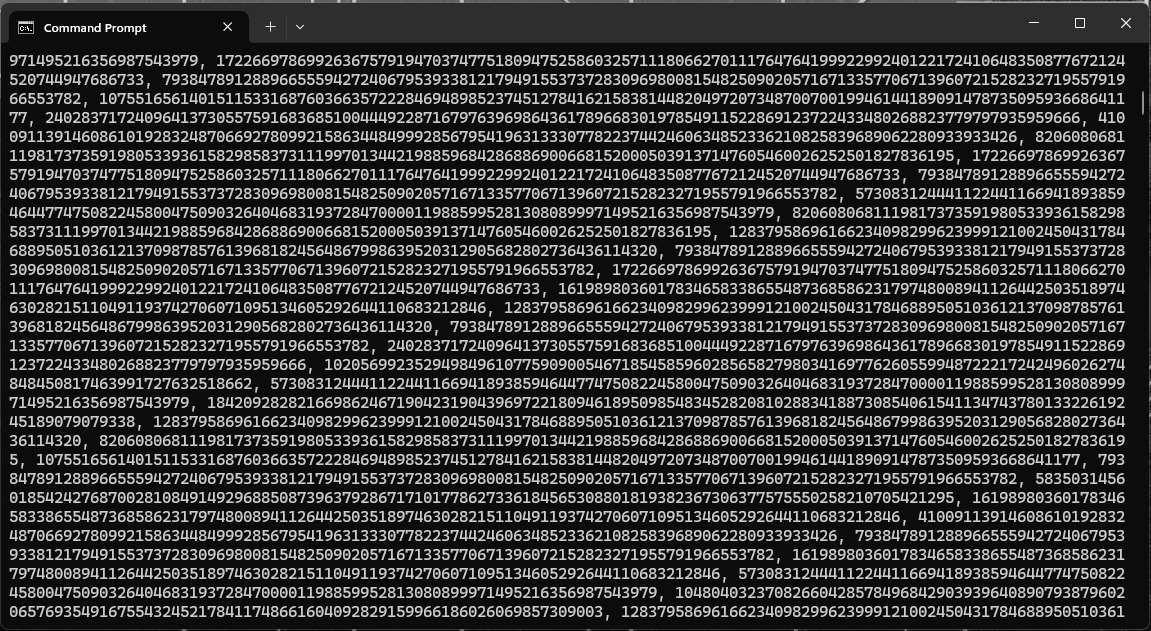
* Importation de la librairie Python **time** pour mesurer les temps d'exécution du processus de la génération des clés, d'encodage et de décodage de messages clairs et pleins.
* Modification de la fonction Python **decode\_message** pour qu'elle retourne une chaîne de caractères numériques.
* Ajout du bloc **if \_\_name\_\_ == '\_\_main\_\_'** pour mesurer les temps d'exécution et organiser le flux principal du programme.
* Ajout des mesures de temps pour la génération des clés, l'encodage, le décodage, et le temps total d'exécution de ces processus de calcul mathématiques, arithmétiques et informatiques du code Python RSA. En version 3.
* Déplacement et regroupement du code de génération des clés et de chiffrement/déchiffrement à l'intérieur du bloc principal, qui est au-dessus du bloc **if \_\_name\_\_ == '\_\_main\_\_'**.
* Mesure de la complexité de l'encodage, du décodage, de l'encryptage et du décryptage à l'aide de boucles **for** pour chaque message test généré, avec affichage des temps d'exécution et des quantités de mémoire utilisées.
* Affichage des informations suivantes :
* Average encoding time
* Average decoding time
* Average encryption time
* Average decryption time
* Average memory used during encoding
* Average memory used during decoding
* Average memory used during encryption
* Average memory used during decryption

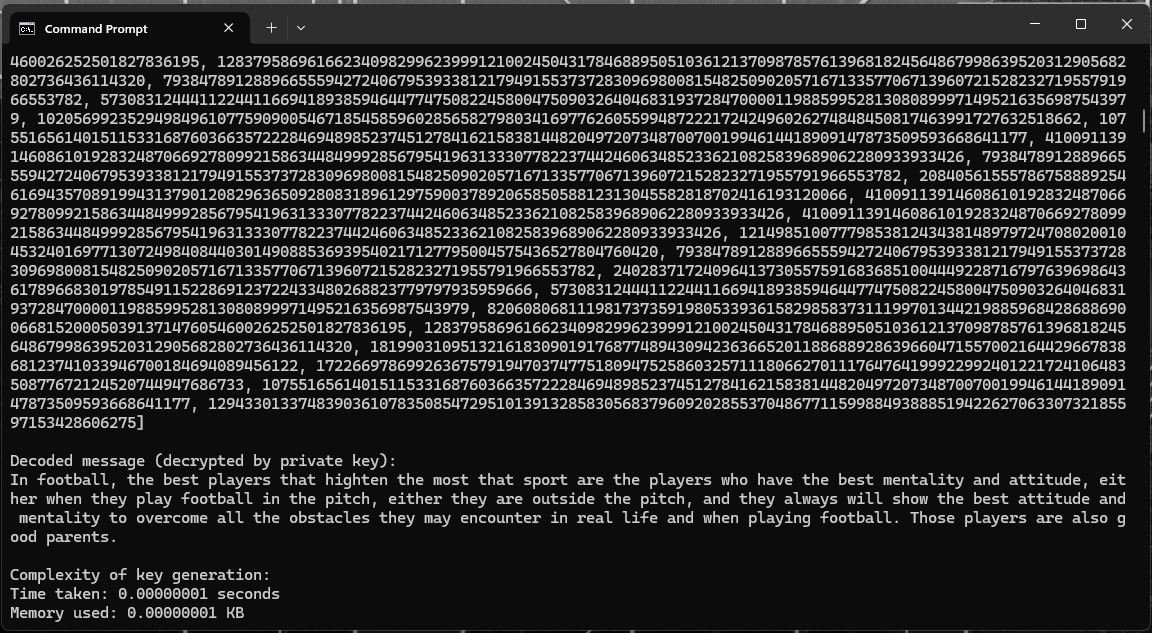
Ces modifications décrites dans le paragraphe montré ci-dessus ajoutent l’implémentation d’un nouveau critère d'évaluation pour mesurer l'efficacité de l'implémentation RSA en termes de complexité algorithmique de type temporelle (en unités de temps de type millisecondes) et en termes d’utilisation de la mémoire (en unités d’espace mémoire de type KB), causée par différentes étapes du processus de génération de clés publiques et privées, celles du chiffrement et celles du déchiffrement de messages clairs et pleins, **et cela, pour réaliser l’implémentation, en langage python 3.12.1, de la gestion de traitement/processus de plusieurs messages clairs et pleins, en lien avec l’algorithme asymétrique RSA.** Les modifications décrites avec luxe de détails, et montrée dans les paragraphes ci-haut, ont donné de très bons résultats, car Gonzalo Alfredo Romero Francia peut maintenant mesurer, avec beaucoup de précision, le temps que prend son code python RSA en version 3, à générer des clés publique et privée, à chiffrer et à déchiffrer des messages clairs et pleins spécifié par lui, lors de la compilation de ce code python RSA. Le voici la sortie que donne le code python RSA implémentant le critère d’évaluation nommé « Gestion de grandes quantités de données » :



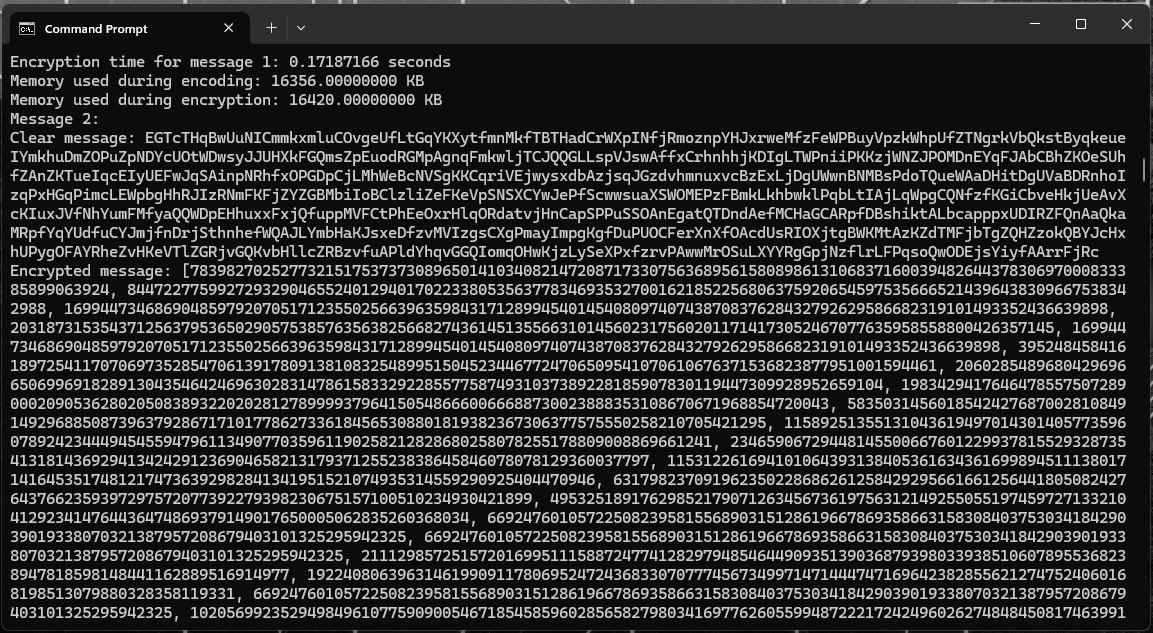


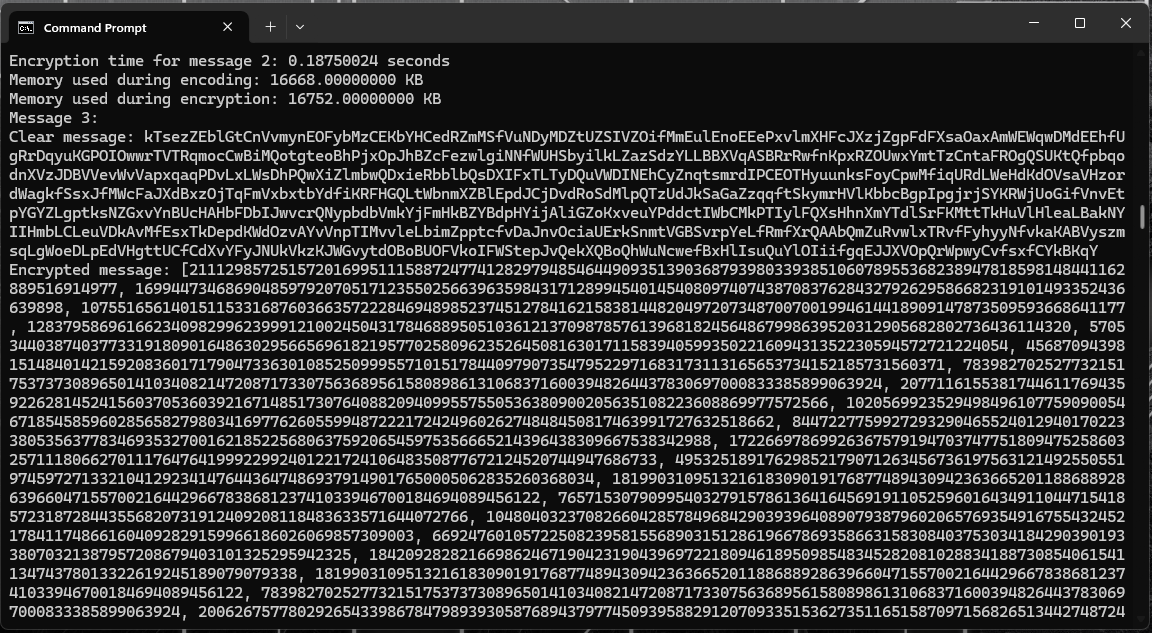


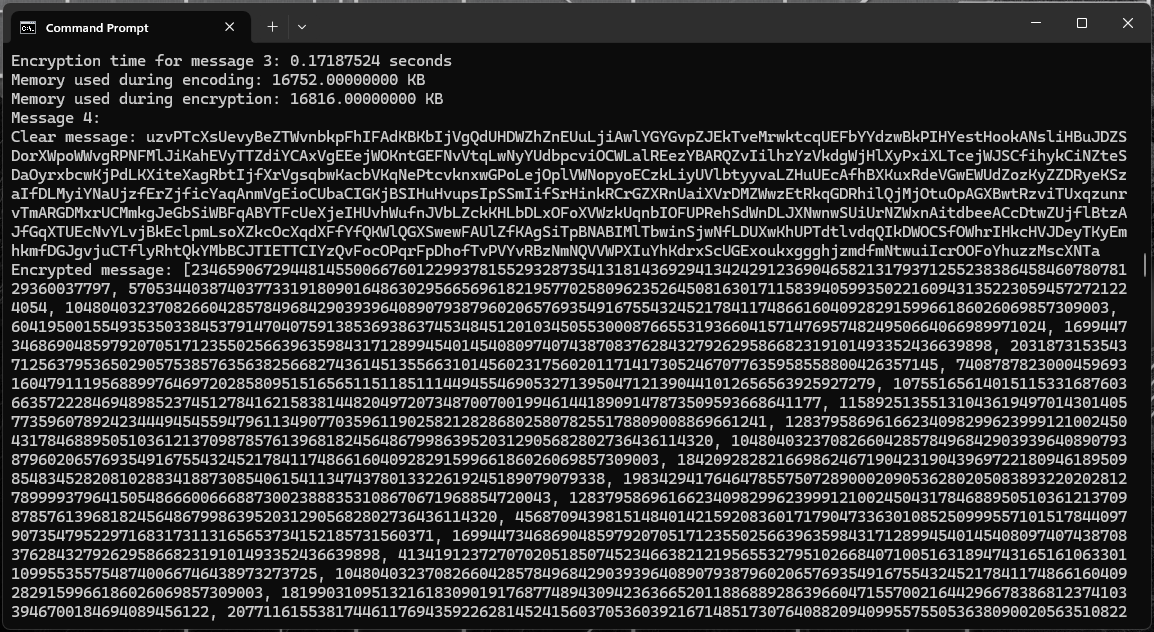


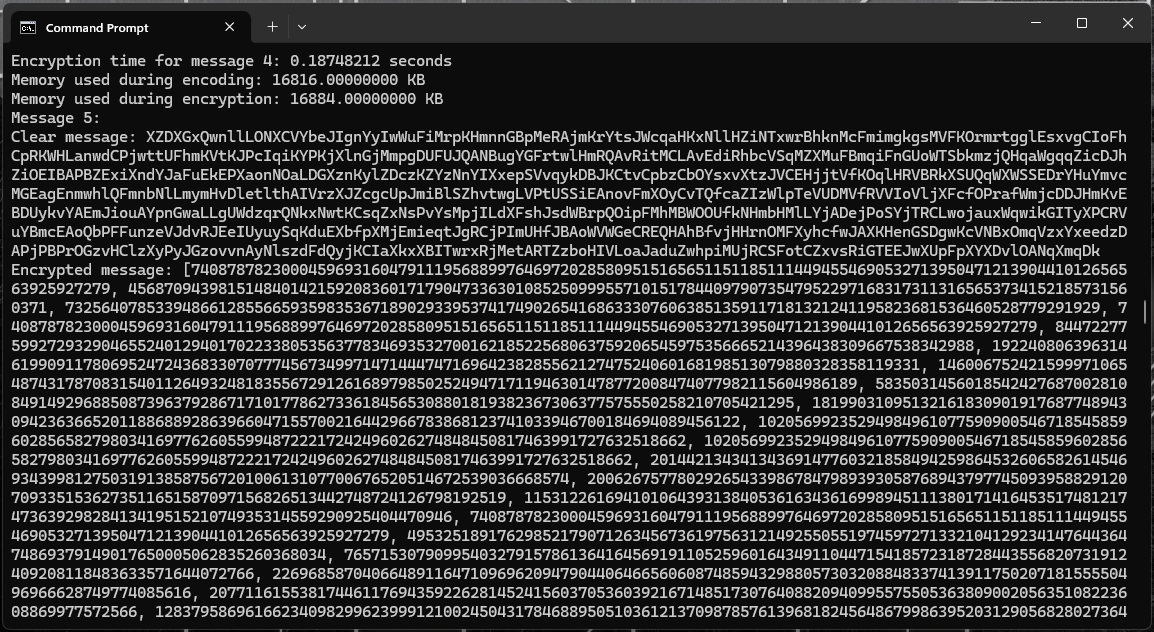


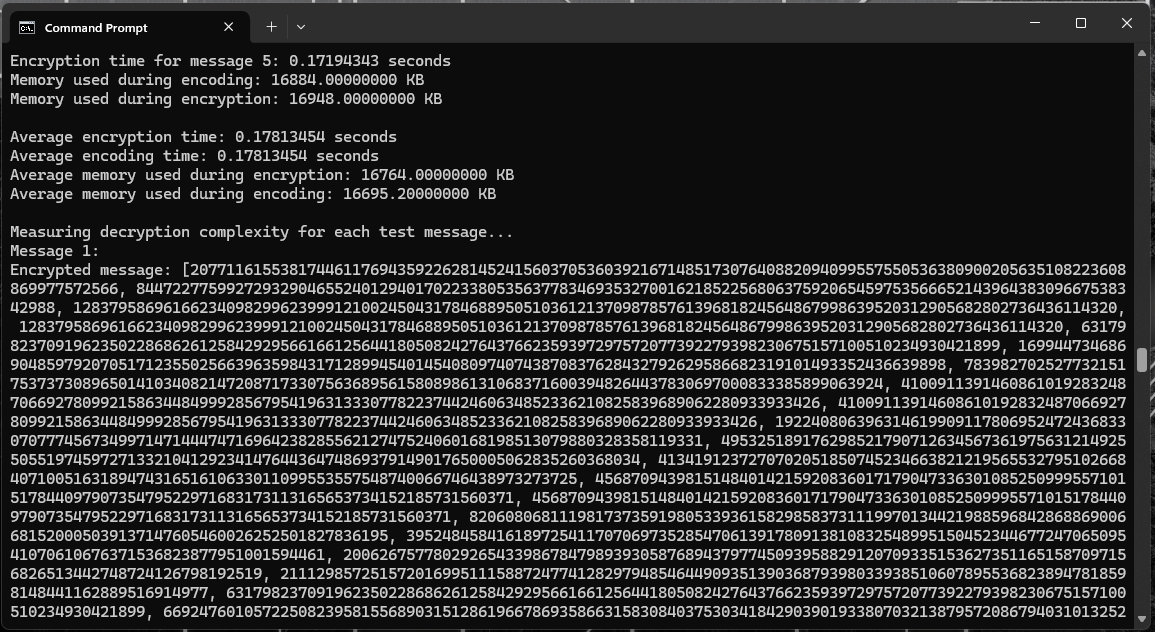


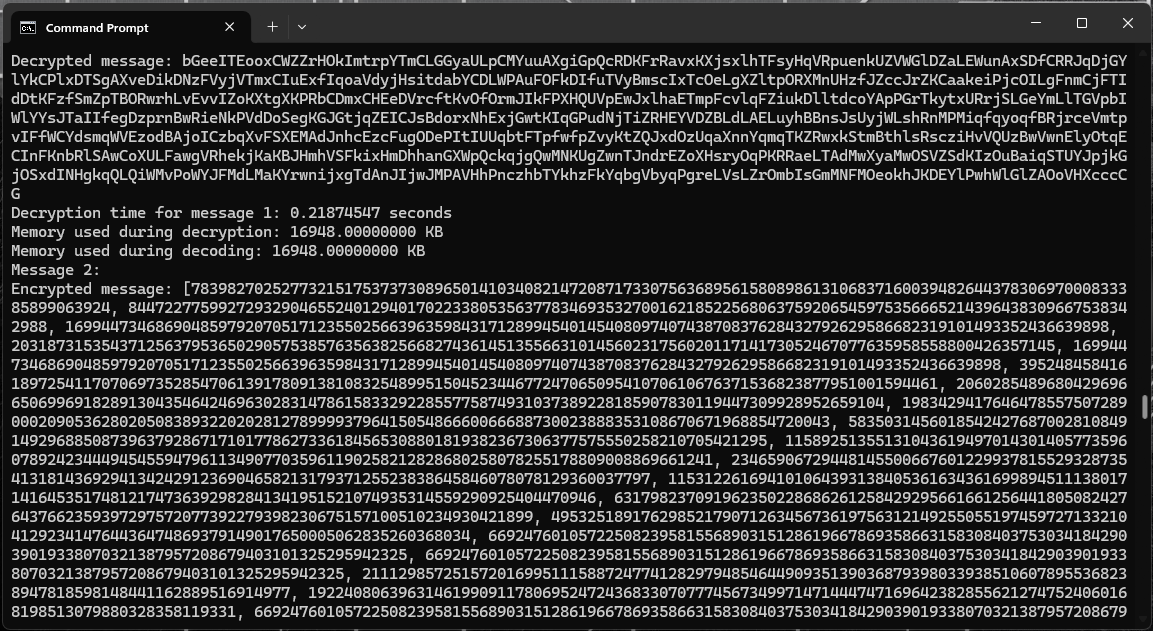


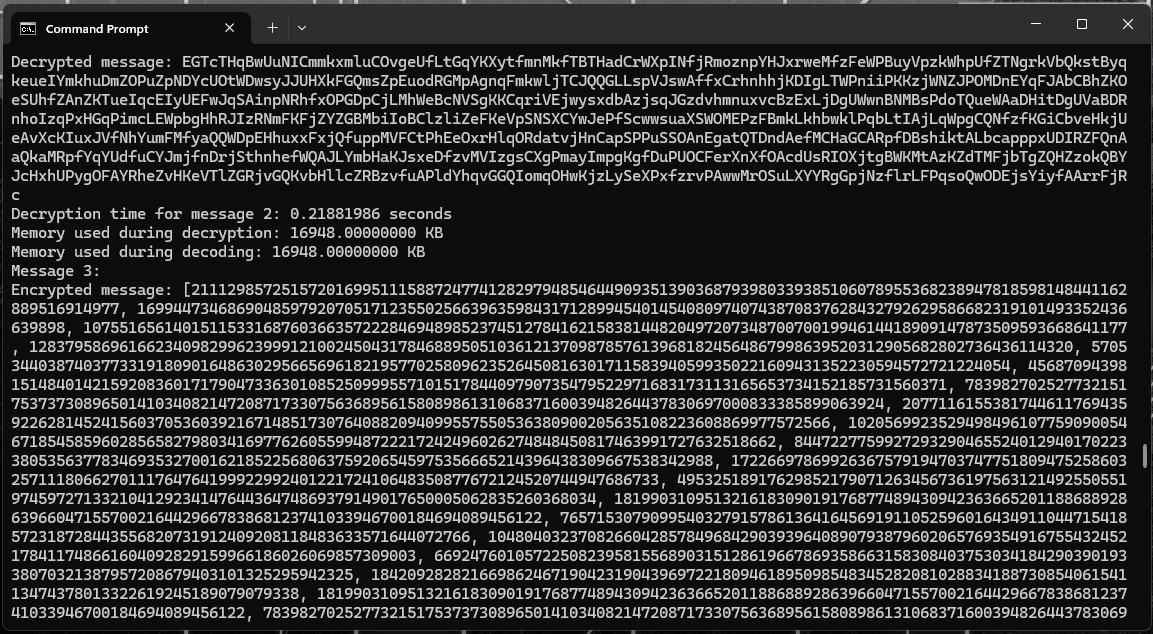


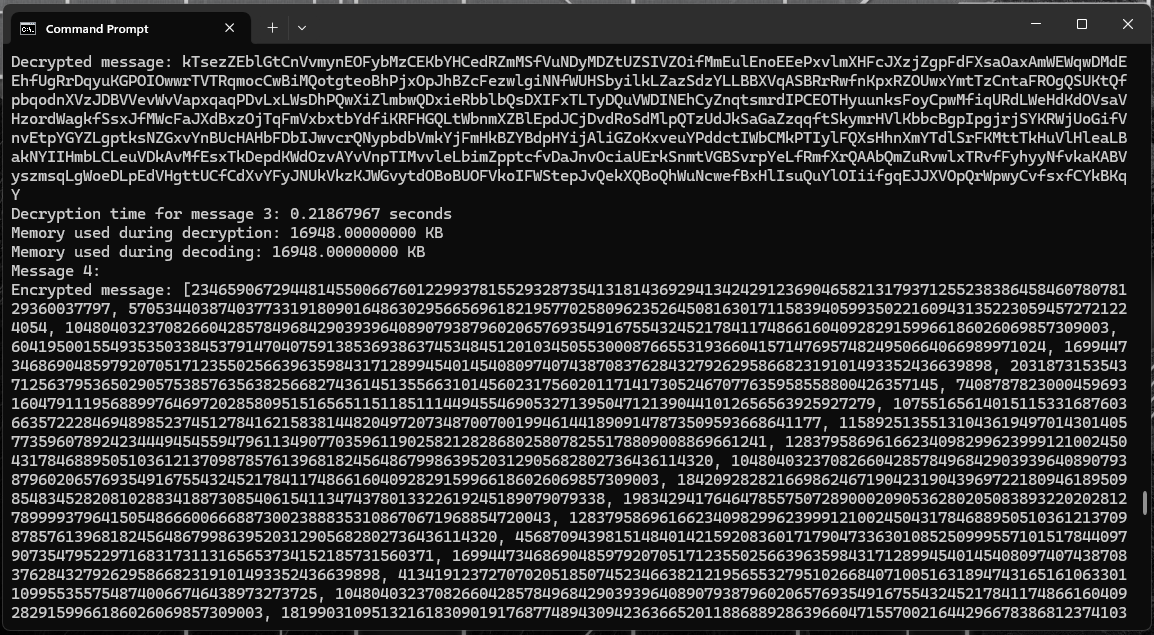


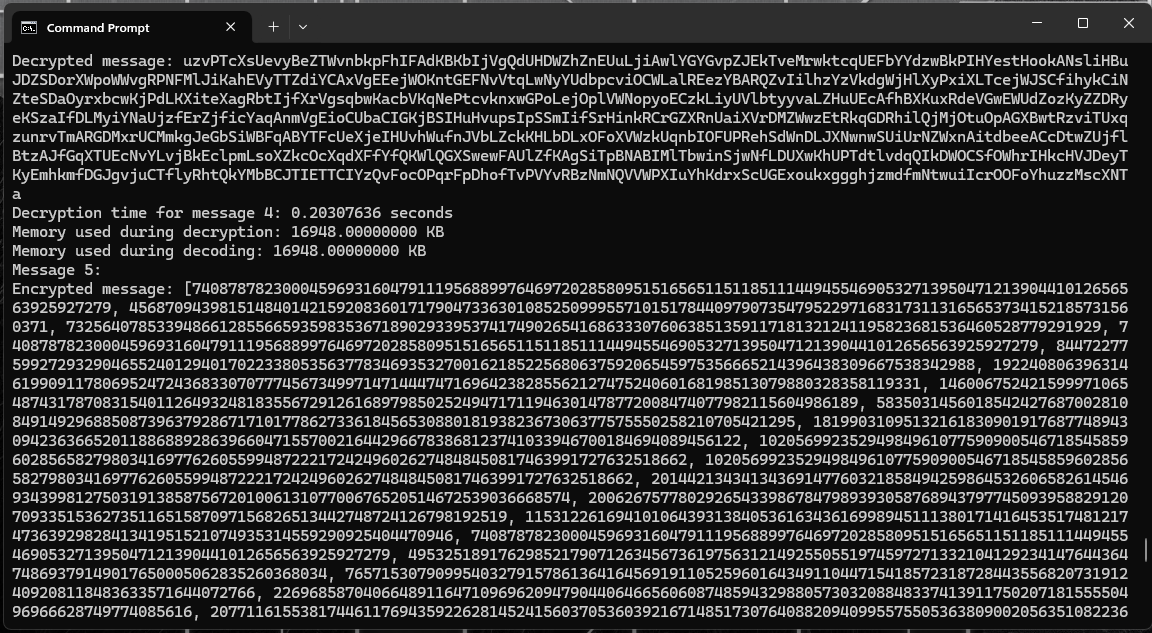


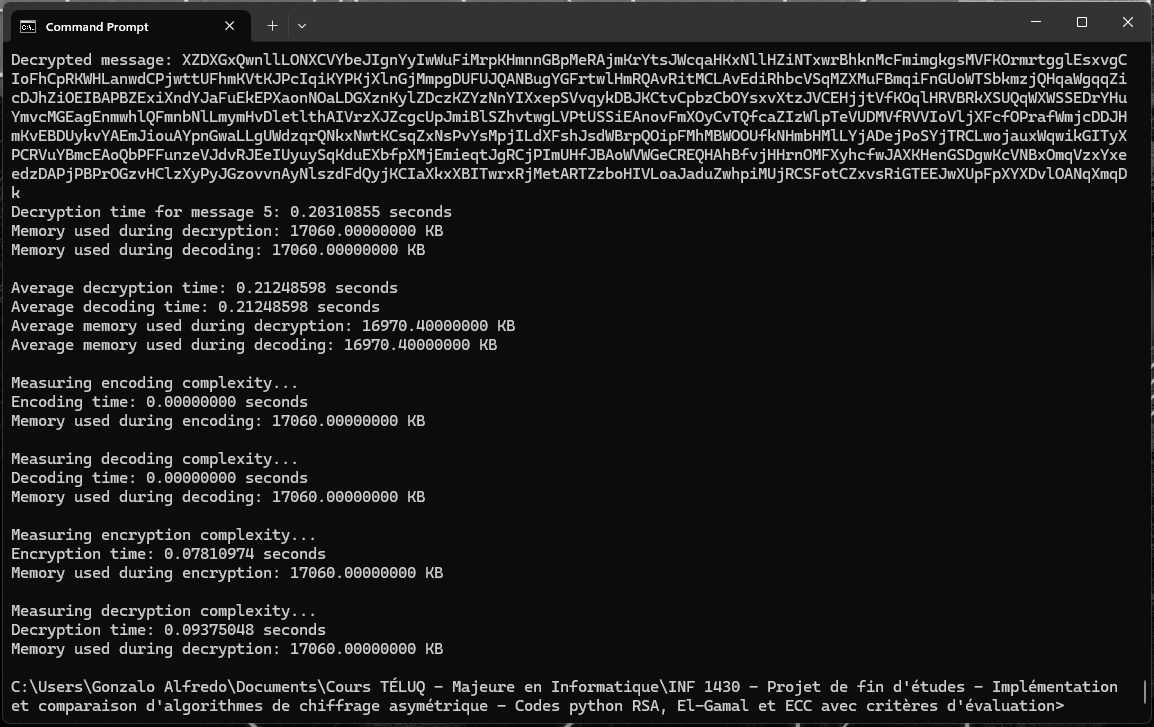












Les figures montrées ci-dessus montrent très bien que les modifications que Gonzalo Alfredo Romero Francia a faits, a permis à son code python RSA en version 3, d’implémenter le critère d’évaluation nommé « Gestion des grandes quantités de données », de manière très rapide, très efficace et très optimale, et produisant des résultats excellents, efficaces, optimales et remarquables.

//Continuer demain à rédiger les autres gros textes, pour les autres critères d’évaluation!!!!

**Conclusion**

Après avoir réalisé toutes les analyses de conception, développement et de programmation des codes python fournies par le document Word du travail noté #1 d’INF 1430, Gonzalo Alfredo Romero Francia s’est rendu compte que ses trois codes python implémentant les algorithmes RSA, El-Gamal et ECC, qu’il a fournis à la dernière section de ce document Word, de son travail noté #1 d’INF 1430, n’étaient pas ni corrigés, ni améliorés ni optimisés, car ces trois codes python, implémentant les algorithmes RSA, El-Gamal et ECC, donnaient de très bonnes performances, mais prenaient un certain petit temps, pour traiter des messages courts et moyens, et prenaient encore plus de temps pour traiter des messages longs ou très longs, ce qui rendait ces trois codes python pas optimales, pas corrigés et pas améliorés.

Afin de démontrer la validité du point déjà décrit dans le paragraphe ci-dessus, Gonzalo Alfredo Romero Francia a réalisé des tests de fonctionnement à ces trois codes python de base implémentant les algorithmes RSA, El-Gamal et ECC, afin de chronométrer le temps d’exécution de ces trois codes python. Et le voici les résultats obtenus :

**Code python RSA :** pour le traitement et la compilation d’un message clair et plein de 7680 bits, le temps d’exécution de la présentation de l’algorithme RSA, les choix offerts par ce code python, en ce qui a trait la paramétrisation de la taille des messages clairs et pleins et l’écriture d’un message clair et plein d’une certaine longueur, en bits, est de 1 minute 25 secondes en moyenne, et le temps d’exécution de tous les calculs mathématiques et de tous les opérations informatiques associés au processus de chiffrement et de déchiffrement de messages clairs et pleins, est de 2 minutes 10 secondes en moyenne

**Code python RSA :** pour le traitement et la compilation d’un message clair et plein de 8192 bits, le temps d’exécution de la présentation de l’algorithme RSA, les choix offerts par ce code python, en ce qui a trait la paramétrisation de la taille des messages clairs et pleins et l’écriture d’un message clair et plein d’une certaine longueur, en bits, est de 1 minute 29 secondes en moyenne, et le temps d’exécution de tous les calculs mathématiques et de tous les opérations informatiques associés au processus de chiffrement et de déchiffrement de messages clairs et pleins, est de 2 minutes 35 secondes en moyenne

**Code python El-Gamal :** pour le traitement et la compilation d’un message clair et plein de n’importe quelle taille, et pour le traitement de tous les opérations informatiques associés au processus de chiffrement et de déchiffrement de messages clairs et pleins, le code python de base El-Gamal est vite d’exécution, **quoi que ce code python n’est pas tout à fait optimale, mais il est possible de rendre ce code python encore plus vite et plus optimale, en termes mathématiques, logicielles et informatiques.**

**Code python ECC :** pour le traitement et la compilation d’un message clair et plein de n’importe quelle taille, et pour le traitement de tous les opérations informatiques associés au processus de chiffrement et de déchiffrement de messages clairs et pleins, le code python de base ECC est vite d’exécution, **quoi que ce code python n’est pas tout à fait optimale, mais il est possible de rendre ce code python encore plus vite et plus optimale, en termes mathématiques, logicielles et informatiques.**

L’auteur du présent document, en réalisant des centaines de recherches d’informations, permettant de corriger, d’améliorer et d’optimiser ses trois codes python RSA, El-Gamal et ECC, et en réalisant des centaines de tests de fonctionnalité d’exécution et de compilation à chacune des trois versions de son code python RSA, à chacune des six versions de son code python El-Gamal, et à chacune des cinq versions de son code python ECC, Gonzalo Alfredo Romero Francia a réussi d’appliquer, de manière très efficace, les approches et philosophies de développement/programmation nommés « Approche de programmation Agile » et « Ingénierie extrême », et le résultat final de toutes les étapes de développement et de programmation récursives, est le code python RSA en version 3, le code python El-Gamal en version 6, et le code python ECC en version 5, qui sont montrés dans les pages 11 à 17, 37 à 42 et 65 à 71 de l’Annexe du présent document, qui a été rédigé à part du présent document.

En guise de conclusion, et après avoir réalisé toutes les tâches d’analyse de lignes de code python, de correction, d’amélioration et d’optimisation de ses trois codes python implémentant les algorithmes de chiffrement et de déchiffrement asymétrique RSA, El-Gamal et ECC, l’auteur du présent document est arrivé à un ensemble de conclusions qui sont les suivantes :

* Il est très important, voire très essentiel et très vital, d’utiliser les approches et philosophies de développement/programmation de programmes et applications informatiques nommés « Approche de programmation Agile » et « Ingénierie extrême », car ils permettent de sauver des efforts de programmation, du temps de programmation et des ressources informatiques.
* Il est très important, voire très essentiel et très vital, de modulariser les blocs de code associés à des opérations et calculs mathématiques associés aux processus de génération de clés publiques et privées, au processus de chiffrement et au processus de déchiffrement, de chacun des trois algorithmes de chiffrement et de déchiffrement asymétrique nommés RSA, El-Gamal et ECC, car la modularisation de tous ces blocs de code permet de mieux programmer , mieux structurer et mieux organiser tous les blocs de code python déjà décrits ci-dessus.
* Il est très important, très essentiel et très vital de comprendre que la meilleure forme de modulariser les blocs de code associés à des opérations et calculs mathématiques associés aux processus de génération de clés publiques et privées, au processus de chiffrement et au processus de déchiffrement, des codes python implémentant les algorithmes asymétriques RSA, El-Gamal et ECC, est de modulariser des blocs de code de ces trois codes python, avec l’approche de développement/programmation nommé « Optimisation petit à petit », cette approche de développement/programmation permet de corriger, d’améliorer et d’optimiser des blocs complets de code python en parties, petit à petit, une ou plusieurs modifications de lignes de code python RSA, El-Gamal et ECC, pour ensuite faire des tests de fonctionnement de ces codes python, afin d’observer son comportement, et réaliser encore des modifications, corrections, améliorations et optimisations récursives, jusqu’à obtenir une version corrigée, améliorée et optimisée, ce qui se confirme avec la réalisation de ces tests de fonctionnement logiciel et informatique.
* Il est très important, très essentiel et très vital de comprendre qu’il doit y avoir une balance entre la rapidité d’exécution que donnent les trois codes python implémentant les algorithmes asymétriques RSA, El-Gamal et ECC, et la quantité de mémoire utilisée par ces trois codes python, car s’il n’y a pas de balancement entre ces deux paramètres de fonctionnement de ces trois codes python, il va se produire facilement des surutilisation de ressources informatiques et/ou la surconsommation de mémoire informatique, ce qui est à proscrire complètement.
* Les trois codes python implémentant les algorithmes de chiffrement et de déchiffrement asymétrique RSA, El-Gamal et ECC, montrés aux pages 11 à 17, 37 à 42 et 65 à 71 de l’Annexe du présent document, qui a été rédigé à part du présent document, sont des codes python déjà paramétrables et distribuables, et aussi déjà corrigés, améliorés et optimisés à 100%. Cependant, il est clair et évident que, lors de la réalisation de la phase 6 du projet de fin d’études nommé «  Implémentation et comparaison des algorithmes de chiffrage asymétrique », il va y avoir une ou deux modifications dans une ou deux fonctions des trois codes python déjà décrits ci-haut, quoi que ces modifications ne seront pas significatives qui changeraient, de manière significative ou au complet, le fonctionnement global de la compilation et de l’exécution des lignes de code des trois codes python RSA, El-Gamal et ECC.

* En général, l’implémentation de tout type de calcul mathématique et informatique, associé à des algorithmes de chiffrement et de déchiffrement asymétrique, requiert beaucoup de connaissances sur les algorithmes de chiffrement et de déchiffrement asymétriques, du temps et des efforts de recherche d’informations, du temps et des efforts d’analyse logique, mathématique et informatique, des capacités et habilétés de conception et de développement/programmation, ainsi que l’utilisation d’approches, philosophies et stratégies de conception, de développement et de programmation très complexes, car le domaine de la cryptographie est un domaine très complexe, très exigeant et très demandant d’utilisation d’habilétés et de capacités mentales, cognitives et intellectuelles, tant pour la réalisation d’analyses théoriques des algorithmes de chiffrement et de déchiffrement asymétrique, que pour l’implémentation en différents langages de programmation, de tous les opérations, calculs et processus associés à ces algorithmes de chiffrement et de déchiffrement asymétrique, tels les algorithmes RSA, El-Gamal et ECC.