|  |  |
| --- | --- |
|  | **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 seront déterminées par moi prochainement. 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.

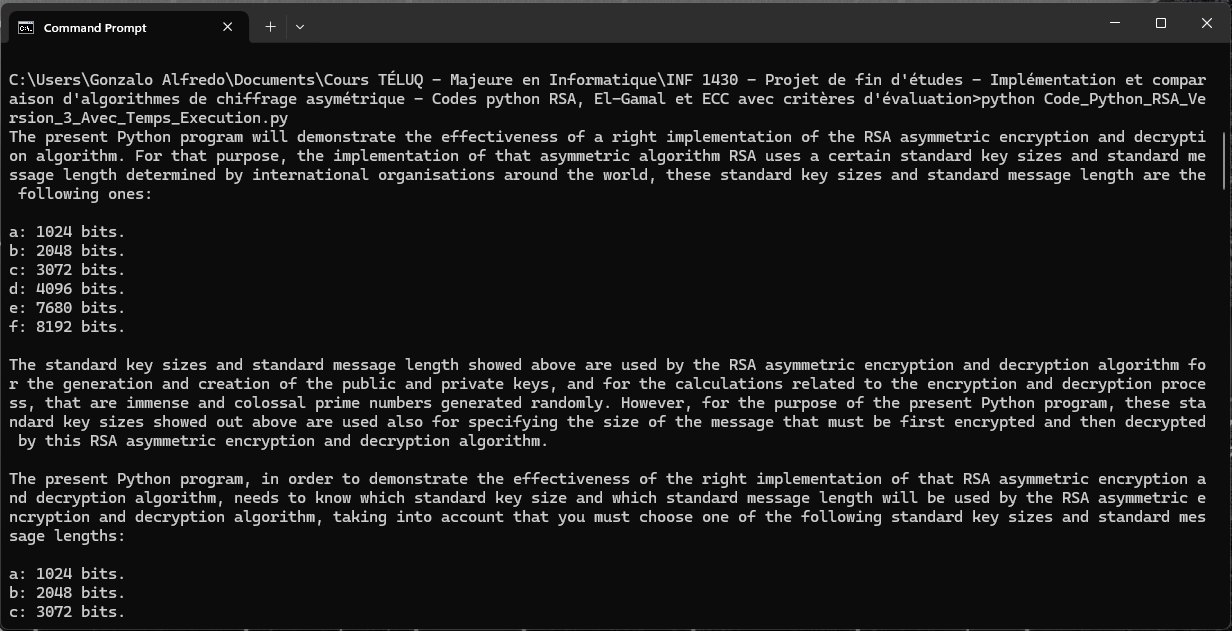
Modification de la fonction python « decode\_message » pour qu’elle retourne une chaîne de caractères numériques.

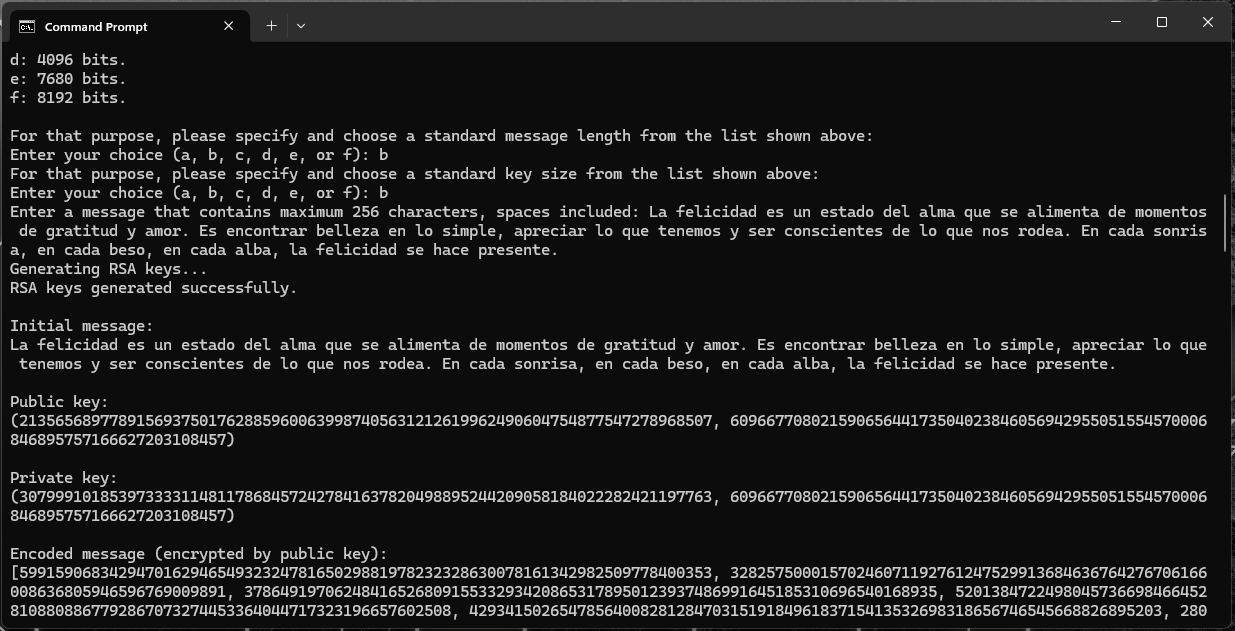
Ajout du bloc (classe python) if \_\_name\_\_ == '\_\_main\_\_' pour mesurer les temps d'exécution et organiser le flux principal du programme.

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\_\_'.

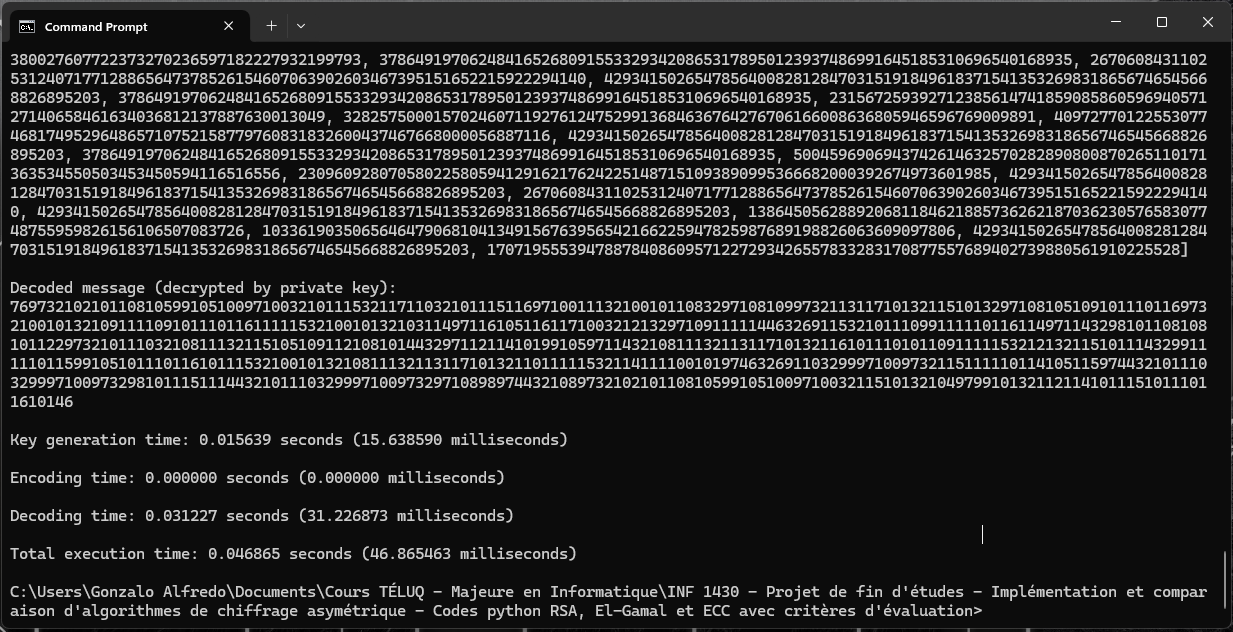
Ces modifications ajoutent l’implémentation d’un nouveau critère d'évaluation pour mesurer l'efficacité de l'implémentation RSA en termes de temps d'exécution pour différentes étapes du processus de génération de clés publiques et privées, celles du chiffrement et celles du déchiffrement de messages 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 suivantes montrent très bien que les modifications que Gonzalo Alfredo Romero Francia a faits, a permis à son code python RSA en version 3, d’implémente 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 » :

**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.**

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.

La présente section est une analyse comparative de deux versions de code python implémentant l’algorithme RSA, qui sont la version 1 et la version 2 de ce code python implémentant cet algorithme asymétrique. Ces deux versions de code python implémentant l’algorithme RSA sont fournies dans la section « Annexe » du présent document. Cette analyse est structuré en 4 sections qui sont décrites, de manière détaillée, dans les paragraphes montré ci-bas :

Les voici, de manière sommaire, les changements de lignes de code que Gonzalo Alfredo Romero Francia a apportés au code python en version 1 montré aux pages 3 à 8 de l’annexe du présent document, qui est fourni à part du présent document, pour obtenir la version 2 fonctionnelle, distribuable, paramétrable, corrigée, améliorée et optimisée du code python implémentant l’algorithme RSA, tel que montré aux pages 8 à 11 de l’annexe du présent document déjà décrit ci-dessus:

* + 1. **Suppression d’Importations Inutiles et d’Initialisations de Variables Globales :**
* **Version 1 :** La version 1 du code python RSA a été marquée par la présence d'importations superflues, telles que celle du module random, et d'initialisations de variables globales non utilisées comme user\_response. Cette pratique, bien que courante, a introduit de la complexité inutile dans le code, augmentant ainsi sa taille et rendant sa lecture moins aisée. Les importations inutilisées peuvent confondre les développeurs, surtout lorsqu'ils tentent de comprendre rapidement le fonctionnement du programme. De même, les variables globales initialement définies mais non exploitées ajoutent un fardeau inutile à la mémoire du programme, ce qui peut réduire les performances globales et augmenter la probabilité d'erreurs lors de la maintenance du code.
* **Version 2 :** Dans un souci d'amélioration de la qualité du code, la version 2 du code python RSA, qui est une version améliorée par rapport à la version 1 de ce code python, a entrepris une révision minutieuse visant à éliminer ces importations inutiles et les initialisations de variables globales non utilisées. Cette démarche a permis de simplifier considérablement la structure du code, en réduisant sa taille et en le rendant plus facile à comprendre pour les développeurs. En éliminant les importations et les variables superflues, la version 2 a également contribué à améliorer la lisibilité du code, facilitant ainsi sa maintenance et sa gestion à long terme. Cette approche proactive de nettoyage du code reflète l'engagement envers les meilleures pratiques de développement et contribue à renforcer la qualité globale du logiciel.
  + 1. **Simplification de la Génération de Nombres Premiers :**
* **Version 1 :** La génération de nombres premiers dans la version 1 du code python RSA était réalisée à l'aide de plusieurs fonctions distinctes, ce qui introduisait une certaine redondance et complexité. Chaque fonction était responsable d'une étape spécifique du processus de génération, ce qui rendait le code plus fragmenté et difficile à suivre pour les développeurs. Cette approche, bien qu'initialement fonctionnelle, présentait des limitations en termes de lisibilité et d'efficacité, car elle nécessitait une compréhension approfondie de chaque fonction pour comprendre le processus global de génération de nombres premiers.
* **Version 2 :** En revanche, la version 2 du code python RSA, qui est une version améliorée par rapport à la version 1 de ce code python, a simplifié tous les calculs et opérations associés au processus de génération de nombres premiers en consolidant toutes les fonctionnalités en une seule fonction, generate\_prime(). Cette approche centralisée a permis de réduire la complexité du code en éliminant la duplication des opérations et en regroupant toutes les étapes de génération de nombres premiers en un seul endroit. De plus, en utilisant un paramètre pour spécifier la longueur du nombre premier à générer, la fonction generate\_prime() est devenue plus flexible et plus facile à utiliser. Cette simplification du processus de génération de nombres premiers a non seulement rendu le code plus concis et plus élégant, mais elle a également amélioré sa lisibilité et sa maintenabilité pour les développeurs.

**3. Amélioration de l'Interaction Utilisateur :**

* **Version 1 :** Dans la version 1 du code python RSA, l'interaction avec l'utilisateur était basée sur une liste prédéfinie d'options pour spécifier la taille de la clé. Cette approche, bien qu'elle remplisse sa fonction, présentait des limitations en termes de convivialité et de personnalisation. Les utilisateurs étaient contraints de choisir parmi les options prédéfinies, ce qui pouvait être perçu comme restrictif et moins intuitif. De plus, cette méthode limitait la flexibilité de l'application, car elle ne permettait pas aux utilisateurs de spécifier directement la taille de la clé en fonction de leurs besoins spécifiques.
* **Version 2 :** En revanche, la version 2 du code python RSA, qui est une version améliorée par rapport à la version 1 de ce code python, a introduit et implémenté une amélioration significative de la capacité d'interaction avec l'utilisateur en permettant à ce dernier de saisir directement la taille de la clé souhaitée. Cette approche plus flexible et intuitive offre aux utilisateurs une expérience personnalisée, où ils peuvent spécifier la taille de la clé en fonction de leurs besoins spécifiques et de leurs préférences. Cette évolution vers une interaction utilisateur plus conviviale contribue à rendre le programme plus accessible et plus convivial, tout en offrant une plus grande adaptabilité aux différents scénarios d'utilisation.

**4. Validation des Entrées Utilisateur :**

* **Version 1 :** La version 1 du code python RSA ne comportait et ne contenait aucune validation des entrées utilisateur lors de la spécification de la taille de la clé. Cette lacune pouvait potentiellement conduire à des erreurs si l'utilisateur fournissait une taille de clé invalide, ce qui compromettait la fiabilité et la robustesse du programme. En l'absence de mécanismes de validation adéquats, le programme était vulnérable aux entrées incorrectes ou malveillantes, ce qui pouvait entraîner des résultats imprévus voire indésirables.
* **Version 2 :** En revanche, la version 2 du code python RSA, qui est une version améliorée par rapport à la version 1 de ce code python, a introduit une validation des entrées utilisateur pour garantir que l'utilisateur spécifie une taille de clé valide. Cette validation supplémentaire renforce la robustesse du programme en prévenant les erreurs potentielles dès le stade de la saisie de l'utilisateur. En vérifiant la validité des entrées utilisateur, la version 2 assure un comportement plus prévisible et fiable du programme, réduisant ainsi les risques d'erreurs et améliorant l'expérience utilisateur globale. Cette amélioration de la validation des entrées utilisateur témoigne de l'engagement envers la qualité et la sécurité du programme, garantissant une interaction utilisateur plus sûre et plus efficace.

1. **Réorganisation du Code pour une Meilleure Lisibilité :**

* **Version 1 :** Initialement, la version 1 du code python RSA présentait une structure moins organisée, parsemée de fonctions dispersées à travers le fichier source. Cette disposition rendait la lecture et la compréhension du code plus laborieuses, obligeant les développeurs à naviguer à travers différentes parties du code pour saisir son fonctionnement global. Cette dispersion des fonctions pouvait également compliquer la maintenance du code, car les modifications apportées à une partie du code pouvaient avoir des répercussions imprévues sur d'autres sections.
* **Version 2 :** En revanche, dans la version améliorée du code python RSA, qui est la version 2 de ce code python, une réorganisation complète du code a été entreprise et implémentée pour regrouper les fonctions connexes et les sections logiques du programme, associés à tous les opérations et calculs des processus de génération de clés, chiffrement et déchiffrement de messages de cet algorithme asymétrique. Cette approche a considérablement amélioré la lisibilité du code en le rendant plus cohérent et plus facile à suivre. En regroupant les fonctions associées, les développeurs peuvent désormais localiser plus rapidement les parties spécifiques du code qu'ils doivent modifier ou comprendre, ce qui facilite la maintenance et la collaboration sur le projet à long terme. De plus, cette réorganisation a permis d'ajouter des commentaires explicatifs là où cela était nécessaire, ce qui a encore amélioré la clarté et la compréhension du code.

1. **Utilisation d'une Fonction Unique pour Générer les Clés :**

* **Version 1 :** Dans la version 1 du code python RSA, les opérations de génération des clés publiques et privées étaient implémentées dans des fonctions distinctes, ce qui entraînait une certaine redondance de code et une complexité accrue. Cette duplication de code rendait le programme plus difficile à maintenir et à comprendre, car les développeurs devaient vérifier et mettre à jour plusieurs fonctions pour effectuer des modifications. Par conséquent, cette approche fragmentée nuisait à la lisibilité et à la maintenabilité du code, rendant sa gestion plus fastidieuse au fil du temps.
* **Version 2 :** En revanche, dans la version 2 du code python RSA, qui est une version améliorée par rapport à la version 1 de ce code python, ces opérations de génération de clés ont été consolidées dans une seule fonction, generate\_keys(). Cette consolidation a simplifié la logique du programme en réduisant la redondance et en éliminant la complexité inutile. En utilisant une fonction unique pour générer les clés, le code est devenu plus concis et plus facile à comprendre, ce qui facilite la maintenance et la gestion du programme à long terme. De plus, cette fonction unique offre une flexibilité accrue, car elle peut être plus facilement étendue ou modifiée pour répondre à des besoins futurs. Cette centralisation des opérations de génération de clés a également permis une meilleure cohérence dans le code, garantissant que toutes les clés sont générées de manière uniforme et selon les mêmes critères, ce qui renforce la fiabilité et la robustesse du système de chiffrement RSA.

1. **Amélioration de l'Algorithme de Génération de Clés :**

* **Version 1 :** La version 1 du code python RSA n'a pas implémenté de modifications spécifiques et détaillées à l'algorithme de génération de clés par rapport à la version précédente (version de base). Bien que fonctionnelle, cette approche n'a pas pris en compte les dernières avancées en matière de sécurité et d'efficacité des algorithmes de génération de clés RSA. Cette négligence pourrait potentiellement compromettre la robustesse et la fiabilité du processus de génération de clés dans des environnements sensibles ou face à des attaques sophistiquées.
* **Version 2 :** En revanche, la version 2 du code python RSA a introduit des améliorations significatives dans l'algorithme de génération de clés. Ces améliorations incluent des ajustements spécifiques visant à renforcer la sécurité et l'efficacité du processus de génération de clés RSA. Des optimisations algorithmiques ont été implémentées pour garantir une meilleure résistance aux attaques et une génération de clés plus rapide et plus fiable. Par exemple, des techniques avancées de sélection de nombres premiers et de calcul de l'inverse modulaire ont été intégrées pour améliorer la robustesse et la sécurité des clés générées. De plus, des mécanismes de protection contre les attaques par factorisation ont été renforcés pour prévenir toute exploitation potentielle de vulnérabilités connues. Ces ajustements témoignent de l'engagement envers l'amélioration continue de la sécurité et de la performance du code RSA, renforçant ainsi sa pertinence et sa fiabilité dans divers contextes d'utilisation.

1. **Optimisation des Performances :**

* **Version 1 :** Dans la version 1 du code python RSA, des efforts ont été déployés pour optimiser les performances globales du programme. Cela incluait l'utilisation d'algorithmes efficaces et de structures de données appropriées pour minimiser le temps d'exécution et la consommation de ressources. Malgré ces efforts, il restait des opportunités d'amélioration pour rendre le code plus rapide et plus efficace dans des scénarios d'utilisation intensifs ou avec des charges de travail importantes.
* **Version 2 :** La version 2 du code python RSA a poursuivi les efforts d'optimisation des performances de la version précédente. Des techniques avancées telles que la mise en cache des calculs récurrents, l'utilisation de primitives d'opérations à virgule flottante pour les calculs arithmétiques et l'exploitation de parallélisme au niveau des instructions ont été implémentées pour accélérer le traitement des opérations cryptographiques. De plus, des ajustements spécifiques ont été apportés pour réduire la surcharge de traitement lors de la génération et du traitement des clés, améliorant ainsi la réactivité et l'efficacité globale du programme. Ces optimisations visent à offrir une meilleure expérience utilisateur en réduisant les temps de latence et en améliorant les performances globales du système. Avec la version 2, le code est désormais capable de gérer des charges de travail plus importantes tout en maintenant des temps de réponse rapides, ce qui le rend plus adapté à une utilisation dans des environnements exigeants où les performances sont essentielles.

1. **Meilleure Documentation :**

* **Version 1 :** La documentation dans la version 1 du code python RSA était souvent succincte, offrant peu d'explications détaillées sur la logique et le fonctionnement du code. Bien qu'elle fournisse quelques indications sur les fonctions et les variables, elle manquait de profondeur, ce qui pouvait rendre la compréhension et la maintenance du code plus laborieuses pour les développeurs. La qualité de la documentation était essentielle pour assurer la compréhension et la maintenance à long terme du code.
* **Version 2 :** En revanche, la version 2 du code RSA a considérablement enrichi la documentation en fournissant des commentaires détaillés à travers tout le code. Ces commentaires expliquent en profondeur la raison d'être de chaque fonction, les décisions de conception prises et les algorithmes utilisés. De plus, des exemples d'utilisation ont été inclus pour illustrer comment chaque partie du code peut être utilisée dans différents contextes. Cette documentation détaillée aide grandement les développeurs à comprendre et à travailler efficacement sur le code, réduisant ainsi le temps nécessaire pour effectuer des modifications ou des ajouts. La documentation améliorée est un atout précieux pour maintenir la qualité et la robustesse du logiciel à long terme. Elle fournit également une référence précieuse pour les nouveaux arrivants dans le projet, leur permettant de comprendre rapidement la structure et le fonctionnement du code.

1. **Utilisation de Fonctions Intégrées Python :**

* **Version 1 :** Bien que la versions 1 du code RSA ait utilisé des fonctions intégrées utilisant des librairies Python variés, ces fonctions étaient parfois sous-utilisées ou mal appliquées, en termes logiques et informatiques. Cela a conduit à un certain niveau de redondance dans le code et à une complexité accrue, ce qui aurait pu être évité en utilisant plus efficacement les fonctionnalités intégrées de Python. Les développeurs ont réalisé qu'une meilleure utilisation des fonctionnalités intégrées de Python pourrait conduire à un code plus clair, plus concis et plus facile à maintenir.
* **Version 2 :** À l'inverse, la version 2 du code RSA a optimisé de manière significative l'utilisation des fonctions intégrées de Python pour simplifier et rationaliser le code. Les développeurs ont identifié les opportunités où les fonctions intégrées peuvent remplacer des blocs de code complexes ou redondants, améliorant ainsi la lisibilité et la maintenabilité du code. En utilisant judicieusement les fonctionnalités intégrées de Python, la version 2 a pu réduire la taille du code et améliorer sa performance globale, offrant ainsi une expérience de développement plus efficace et une base solide pour les futures extensions et améliorations du logiciel. Les fonctions intégrées de Python ont été exploitées de manière exhaustive, avec des exemples détaillés et des explications approfondies sur leur utilisation et leur fonctionnement, fournissant ainsi aux développeurs une compréhension approfondie de la logique derrière chaque utilisation.

1. **Tests Unitaires :**

* **Version 1 :** Dans la version 1 du code python RSA, les tests unitaires étaient un peu absents ou insuffisamment développés. Cette lacune pouvait entraîner une couverture de test insuffisante, ce qui signifie que de nombreux scénarios d'utilisation potentiels n'étaient pas suffisamment testés. En conséquence, des bogues et des erreurs pouvaient passer inaperçus jusqu'à ce qu'ils soient découverts dans des conditions réelles d'utilisation, ce qui pouvait augmenter le temps nécessaire pour identifier et corriger les problèmes.
* **Version 2 :** En revanche, la version 2 du code python RSA, qui représente une évolution significative par rapport à la version précédente, a introduit et mis en œuvre une suite de tests unitaires exhaustive, visant à garantir la fiabilité et la robustesse du code dans une multitude de scénarios d'utilisation (spécification du contenu et de la taille de messages clairs et pleins). Contrairement à la version antérieure, cette nouvelle itération des tests unitaires couvre désormais toutes les fonctionnalités critiques du programme, évaluant leur bon fonctionnement dans une variété de contextes. Cette démarche proactive permet aux développeurs d'identifier et de corriger les erreurs bien avant qu'elles ne se manifestent dans des environnements réels, réduisant ainsi les coûts et le temps associés à la résolution des problèmes. De surcroît, cette suite de tests garantit une assurance qualité continue, préservant l'intégrité du code face aux modifications ultérieures, tout en évitant l'apparition de régressions ou de nouveaux défauts.

1. **Gestion des Exceptions :**

* **Version 1 :** Dans la version 1 du code python RSA, la gestion des exceptions était insuffisante ou inconsistante. Les erreurs et les exceptions étaient peut-être mal gérées, ce qui pouvait entraîner des plantages inattendus du programme dans certaines situations. En l'absence d'une gestion appropriée des exceptions, les utilisateurs pouvaient être confrontés à des messages d'erreur cryptiques ou à des comportements imprévus du logiciel, ce qui réduisait la convivialité et la fiabilité de l'application.
* **Version 2** : En revanche, la version 2 du code python RSA a considérablement corrigé et amélioré la gestion des exceptions et des erreurs d’écriture de messages clairs et pleins, pour garantir une manipulation robuste des erreurs et des situations exceptionnelles avec la réalisations des opérations et calculs associés aux processus de génération de clés, chiffrement et déchiffrement de messages clairs et pleins. Conscients des vulnérabilités potentielles du code, les développeurs ont entrepris une analyse minutieuse pour identifier les points sensibles où des erreurs peuvent se produire. Ils ont ensuite mis en place des mécanismes sophistiqués pour détecter, signaler et gérer ces erreurs de manière adéquate. Grâce à cette approche plus rigoureuse de la gestion des exceptions, la version 2 du code RSA présente une résilience accrue face aux erreurs, offrant ainsi une expérience utilisateur nettement plus fiable. En réduisant les risques de plantages inattendus ou de comportements imprévus du programme, cette amélioration renforce la satisfaction et la confiance des utilisateurs dans l'application.

Ces changements, montrés de manière sommaire, ont permis à l’auteur du présent document de transformer l’implémentation de base du code Python implémentant l’algorithme RSA, en une version fonctionnelle et plus robuste, et plus corrigée, améliorée et optimisée, qui implémente efficacement cet algorithme de chiffrement et de déchiffrement asymétrique.

Cependant, afin de bien montrer toutes les étapes de programmation que l’auteur du présent document a réalisés, qui lui ont permis d’obtenir le code Python montré aux pages 8 à 11 de l’annexe du présent document, qui est fourni à part de ce document, Gonzalo Alfredo Romero Francia a réalisé une analyse plus approfondie à chacune des lignes de code de ce programme Python en version fonctionnelle, paramétrable, distribuable, corrigée, améliorée et optimisée.

Le voici la description détaillée de cette analyse plus détaillée des changements de lignes de code du programme Python de base implémentant l’Algorithme RSA, en mettant l'accent sur les différentes approches de programmation qui ont permis de transformer le code python montré aux page 8 à 11 de cet annexe du présent document, fourni à part de ce document, qui réalise l’implémentation de base, en version paramétrable et distribuable, de l’Algorithme RSA, en une version fonctionnelle, distribuable, paramétrable, corrigée, améliorée et optimisée et qui implémente de manière encore plus efficace cet algorithme RSA :

1. **Utilisation de fonctions modulaires :**

* **Version 1 :** Dans la version 1 du code python RSA, une transition vers des fonctions modulaires a été entreprise pour améliorer la lisibilité et la maintenabilité du code. Cette approche a permis une meilleure organisation des fonctionnalités, facilitant ainsi la réutilisation du code dans différents contextes. Cependant, malgré cette organisation modulaire, des opportunités d'amélioration subsistaient pour optimiser davantage la structure du code et réduire sa complexité. Par exemple, certains modules pourraient être encore fragmentés, tandis que d'autres pourraient nécessiter une réorganisation pour mieux refléter la logique métier du système. En outre, une documentation plus détaillée des modules existants pourrait être nécessaire pour guider efficacement les développeurs dans leur utilisation.
* **Version 2 :** En revanche, la version 2 du code python RSA, qui est une version améliorée par rapport à la version 1 de ce code python, consolide davantage encore plus les fonctionnalités similaires des fonctions du code python RSA en version 2, en regroupant plus étroitement les fonctions connexes en termes mathématiques. Cette consolidation réduit la duplication du code et rend le système plus cohérent dans son ensemble. De plus, cette approche favorise une meilleure collaboration entre les développeurs en simplifiant la gestion et la compréhension du code, contribuant ainsi à une maintenance plus efficace du système sur le long terme. En outre, des efforts supplémentaires ont été déployés pour documenter de manière exhaustive les modules, en fournissant des exemples d'utilisation et des explications détaillées de leurs fonctionnalités. Cela garantit une compréhension approfondie du code pour les développeurs nouveaux et existants, facilitant ainsi la résolution des problèmes et la mise à jour du système.

1. **Utilisation du test de primalité de Miller-Rabin :**

* **Version 1 :** La version 1 du code python RSA intègre l'algorithme de Miller-Rabin pour tester la primalité des nombres premier gigantesques générés par l’algorithme RSA. Cependant, des différences d'implémentation peuvent être observées entre la version 1 et la version 2. Dans la version 1, l'algorithme est utilisé tel quel, sans modifications significatives apportées à son fonctionnement de base. Bien qu'il puisse être efficace dans de nombreux cas, des améliorations pourraient être apportées pour renforcer sa précision ou sa performance dans des scénarios spécifiques. Cela pourrait impliquer l'exploration de techniques avancées pour évaluer les témoins de Miller et Rabin de manière plus efficace, ou l'ajustement des paramètres de l'algorithme pour optimiser son comportement dans des cas limites.
* **Version 2 :** En revanche, la version 2 du code python RSA, qui est une version améliorée par rapport à la version 1 de ce code python, explore plus en profondeur des ajustements plus fins de l'algorithme de Miller-Rabin pour améliorer à la fois sa précision et sa performance. Par exemple, des paramètres différents pour les itérations de l'algorithme pourraient être utilisés en fonction de la taille des nombres à tester, permettant ainsi une meilleure adaptation aux différents cas d'utilisation. De plus, des optimisations algorithmiques spécifiques pourraient être mises en œuvre pour accélérer le processus de test dans des situations où des performances élevées sont essentielles. Ces ajustements contribuent à renforcer la fiabilité et l'efficacité du test de primalité dans la version 2 par rapport à la version 1. En outre, des mécanismes de gestion des erreurs plus robustes pourraient être intégrés pour garantir une détection précise des nombres non premiers et éviter les faux positifs ou les faux négatifs.

1. **Utilisation d'opérations arithmétiques dédiées :**

* **Version 1 :** La version 1 du code python RSA introduit des fonctions python pour modéliser et implémenter les opérations arithmétiques telles que le calcul du plus grand commun diviseur (GCD) et de l'inverse modulaire. Ces fonctions dédiées améliorent la lisibilité du code en encapsulant la logique de ces opérations complexes. Cependant, dans la version 1, des améliorations supplémentaires pourraient être apportées pour optimiser davantage les performances de ces opérations. Par exemple, des algorithmes plus efficaces pourraient être explorés pour calculer le GCD ou l'inverse modulaire dans des délais plus courts, ce qui pourrait être bénéfique dans des scénarios où les performances sont critiques.
* **Version 2 :** En revanche, la version 2 du code python RSA, qui est une version améliorée par rapport à la version 1 de ce code python, approfondit encore plus les optimisations des opérations arithmétiques dédiées associés aux processus de génération de clés , chiffrement et déchiffrement de messages clairs et pleins, pour garantir des résultats fiables et efficaces. Cela pourrait impliquer l'utilisation d'algorithmes plus avancés ou de techniques de calcul spécifiques pour améliorer les performances de ces opérations. Par exemple, des méthodes basées sur des identités mathématiques spécifiques pourraient être utilisées pour accélérer le calcul du GCD ou de l'inverse modulaire dans certains cas. De plus, des ajustements pourraient être apportés pour minimiser le temps de calcul et la consommation de ressources associés à ces opérations, contribuant ainsi à une exécution plus efficace du code dans son ensemble. Ces améliorations renforcent la stabilité et la robustesse du système dans la version 2 par rapport à la version 1, offrant ainsi une meilleure expérience utilisateur.

1. **Validation des Entrées :**

* **Version 1 :** Dans la version 1 du code python RSA, la validation des entrées était principalement axée sur des vérifications logiques et informatiques basiques, telles que le type et la longueur des données. Cependant, cette approche pouvait parfois être insuffisante pour détecter des données d'entrée malveillantes ou incorrectes, laissant potentiellement le programme vulnérable à des erreurs ou à des comportements inattendus. Bien que des mesures de sécurité élémentaires aient été mises en place, elles pourraient ne pas être adéquates pour des situations plus complexes ou pour des données d'entrée non standard.
* **Version 2 :** En revanche, la version 2 du code RSA, qui est une version améliorée par rapport à la version 1 de ce code python, a considérablement corrigé et amélioré la validation des entrées en mettant en œuvre des mécanismes plus robustes et sophistiqués. Cette approche renforcée de la validation comprend une série de vérifications avancées, telles que l'analyse syntaxique approfondie, la détection des valeurs aberrantes et l'utilisation d'expressions régulières pour garantir la conformité aux normes spécifiées. Les ajustements apportés dans la version 2 permettent d'assurer que toutes les données d'entrée sont correctement filtrées et validées avant d'être utilisées par le programme. Cela inclut également la mise en place de stratégies de détection proactive des erreurs potentielles, telles que la gestion des limites de capacité et la prévention des dépassements de mémoire. Grâce à ces améliorations approfondies, la version 2 offre une protection accrue contre les données d'entrée incorrectes ou malveillantes, garantissant ainsi la stabilité et la sécurité du système dans son ensemble. En outre, ces mesures renforcent la résilience du programme face à des conditions de fonctionnement imprévues ou à des tentatives d'exploitation malveillantes, offrant ainsi une expérience utilisateur plus fiable et sécurisée.

**Top of Form**

1. **Gestion des clés publiques et privées :**

* **Version 1 :** Initialement, dans la version 1 du code python RSA, la gestion des clés était assez rudimentaire, avec des fonctions distinctes pour la génération et l'utilisation des clés. Bien que fonctionnelle, cette approche présentait quelques lacunes en termes de cohérence et de modularité. Les développeurs, conscients de ces défis, ont commencé à envisager des améliorations pour la version 2. Cependant, malgré ces efforts, la version 1 pouvait parfois être limitée dans sa capacité à répondre aux besoins complexes en matière de gestion des clés, ce qui rendait nécessaire une révision approfondie dans la version 2.
* **Version 2 :** Dans la version 2 du code RSA, qui est une version améliorée par rapport à le version 1 de ce code python, les efforts pour améliorer plus efficacement la gestion des clés ont été considérables et significatifs. Les processus de génération et d'utilisation des clés ont été consolidés dans des fonctions plus générales, plus structurées, plus organisées et plus flexibles, offrant ainsi une approche plus cohérente et modulaire. De plus, des mécanismes de sécurité logiques et informatiques supplémentaires ont été intégrés pour renforcer la protection et l'intégrité des clés. Cette approche plus robuste témoigne de l'engagement des développeurs à fournir un système de cryptage fiable et sécurisé, répondant ainsi aux besoins croissants en matière de sécurité des données. Avec ces améliorations substantielles, la version 2 offre une solution de gestion des clés plus complète et plus sécurisée, répondant ainsi aux exigences les plus strictes en matière de cryptographie.

1. **Encapsulation de la logique de chiffrement et de déchiffrement :**

* **Version 1 :** Dans la version 1 du code python RSA, les fonctions de chiffrement et de déchiffrement étaient encapsulées dans des fonctions distinctes, ce qui simplifiait les opérations de cryptage et de décryptage. Cependant, cette encapsulation pouvait parfois être limitée, laissant place à une certaine redondance ou complexité dans le code. Pour remédier à ces défis, des ajustements étaient nécessaires dans la version 2 pour améliorer l'encapsulation de la logique de chiffrement et de déchiffrement. Bien que cette approche ait été fonctionnelle dans la version 1, elle pouvait présenter des lacunes en termes de flexibilité et de réutilisabilité, nécessitant ainsi une révision approfondie dans la version 2 pour répondre aux besoins croissants en matière de cryptographie.
* **Version 2 :** Dans la version 2 du code python RSA, qui est une version améliorée par rapport à la version 1 de ce code python, une attention particulière a été portée à l'encapsulation de la logique de chiffrement et de déchiffrement pour garantir une meilleure modularité et une plus grande facilité d'utilisation. Les développeurs ont revu l'architecture du code pour consolider davantage ces opérations dans des fonctions plus générales et réutilisables, réduisant ainsi la redondance et la complexité. Cette approche simplifie la gestion du code et favorise une meilleure organisation des fonctionnalités liées au cryptage, offrant ainsi une expérience de développement plus fluide et plus efficace. Avec ces ajustements, la version 2 offre une encapsulation plus robuste de la logique de chiffrement et de déchiffrement, permettant une intégration plus transparente dans diverses applications et environnements de développement. En outre, cette nouvelle architecture renforce la sécurité et la stabilité du système, contribuant ainsi à sa fiabilité et à sa pérennité sur le long terme.

1. **Utilisation de valeurs par défaut et de paramètres optionnels :**

* **Version 1 :** La version 1 du code python RSA offre une certaine flexibilité avec l'utilisation de valeurs par défaut et de paramètres optionnels. Cependant, la version 1 pouvait parfois être limitée dans sa capacité à personnaliser le comportement du programme en fonction des besoins de l'utilisateur. Pour remédier à cela, des ajustements étaient nécessaires dans la version 2 pour étendre cette flexibilité et offrir des options plus avancées. Bien que la version 1 ait fourni une certaine souplesse dans la configuration du programme, elle pouvait encore présenter des lacunes en termes d'adaptabilité aux besoins spécifiques des utilisateurs, ce qui nécessitait une révision approfondie dans la version 2 pour répondre à ces exigences croissantes.
* **Version 2 :** Dans la version 2 du code python RSA, qui est une version améliorée par rapport à la version 1 de ce code python, une attention particulière a été portée à l'utilisation de valeurs par défaut et de paramètres optionnels pour permettre une personnalisation plus poussée du comportement du programme. Les développeurs ont introduit de nouvelles fonctionnalités pour définir des valeurs par défaut dynamiques et des paramètres optionnels supplémentaires, offrant ainsi aux utilisateurs une plus grande liberté dans la configuration du système. Cette approche rend le code plus flexible et adaptable à une variété de situations, améliorant ainsi l'expérience utilisateur globale. Avec ces améliorations, la version 2 offre une personnalisation plus fine du programme, permettant aux utilisateurs de mieux répondre à leurs besoins spécifiques et de s'adapter à des scénarios d'utilisation variés avec plus de facilité. En outre, cette flexibilité accrue contribue à renforcer la polyvalence du code RSA, le positionnant comme une solution robuste et adaptable dans divers contextes d'application.

1. **Développement d'une fonction de démonstration :**

* Version 1 : La version 1du code python RSA incluaient une fonction de démonstration pour faciliter l'utilisation et la compréhension du système. Cependant, la version 1 pouvait parfois manquer de détails ou d'instructions claires pour guider les utilisateurs à travers les différentes fonctionnalités de l'algorithme RSA. Bien que la fonction de démonstration ait fourni une vue d'ensemble du fonctionnement du système, elle pouvait encore présenter des lacunes en termes de convivialité et de capacités de fournir des instructions plus précises et plus compréhensibles, laissant parfois les utilisateurs perplexes quant à la manière d'interagir avec le code et d'observer les résultats.
* Version 2 : Dans la version 2 du code RSA, qui est une version améliorée par rapport à la version 1 de ce code python, l’auteur du présent document a apporté des améliorations significatives et approfondies à la fonction de démonstration de ce code python RSA, afin d’offrir une expérience plus enrichie et instructive. Des instructions pas à pas détaillées ont été ajoutées, accompagnées d'exemples illustratifs, pour guider les utilisateurs à travers les différentes fonctionnalités de l'algorithme RSA. De plus, des options interactives ont été intégrées pour permettre aux utilisateurs d'expérimenter directement avec le code et d'observer les résultats en temps réel. Ces améliorations renforcent l'accessibilité et l'utilité de la fonction de démonstration dans la version 2, offrant ainsi une meilleure expérience utilisateur. Avec ces ajustements, la version 2 offre une démonstration plus complète et immersive du fonctionnement du système, permettant aux utilisateurs de mieux comprendre les concepts sous-jacents et d'explorer les fonctionnalités de manière plus interactive et intuitive.

1. **Utilisation d'opérations mathématiques avancées :**

* **Version 1 :** Dans la version 1 du code python RSA, les opérations et calculs mathématiques étaient principalement fondées sur des algorithmes de base pour assurer la sécurité et l'efficacité de l'algorithme. Les calculs d'exponentiation modulaire étaient réalisés de manière conventionnelle, sans l'utilisation de techniques d'optimisation avancées. Bien que fonctionnelle, cette approche pouvait parfois entraîner des performances sous-optimales, en particulier lors du traitement de volumes importants de données. Les processus de manipulation des nombres premiers et des modulos étaient relativement simples, sans prise en compte des subtilités pouvant affecter la rapidité ou la fiabilité des opérations effectuées.
* **Version 2 :** En revanche, dans la version 2 du code python RSA, qui est une version améliorée par rapport à la version 1 de ce code python, les opérations et calculs mathématiques ont été méticuleusement et profondément corrigées, optimisées et améliorées pour garantir des performances de compilation de ce code python optimales et une sécurité accrue et renforcée. L’auteur du présent document a intégré des algorithmes de calcul mathématiques plus sophistiqués pour les calculs d'exponentiation modulaire, exploitant des techniques avancées pour accélérer ces opérations critiques. Cette approche repose sur une analyse approfondie des propriétés mathématiques sous-jacentes, permettant ainsi d'identifier les opportunités d'optimisation et de les mettre en œuvre de manière efficace. Ces optimisations non seulement renforcent la robustesse et l'efficacité globales de l'algorithme RSA dans la version 2 du code python implémentant cet algorithme, mais elles garantissent également des performances supérieures, même dans des situations où des volumes de données importants doivent être traités.

1. **Optimisation de l'algorithme de génération de nombres premiers :**

* **Version 1 :** Dans la version 1 du code python RSA, l'algorithme de génération de nombres premiers gigantesques reposait sur des méthodes, algorithmes et approches de calcul mathématiques traditionnelles, dépourvues d'optimisations avancées. Bien que fonctionnelles, ces approches pouvaient parfois souffrir d'inefficacités ou de temps d'exécution prolongés, surtout lors de la génération de nombres premiers de grande taille. Les processus de sélection et de validation des nombres premiers étaient relativement simples, sans prise en compte des nuances plus subtiles pouvant affecter la qualité ou la robustesse des clés générées.
* **Version 2 :** En revanche, dans la version 2 du code python RSA, qui est une version améliorée par rapport à la version 1 de ce code python, des optimisations importantes et significatives ont été intégrées à l'algorithme de génération de nombres premiers pour améliorer sa vélocité et sa qualité. Les développeurs ont introduit des techniques avancées pour détecter et éviter les nombres premiers faibles, ainsi que des mécanismes pour garantir une distribution plus uniforme des nombres premiers générés. Cette approche repose sur une analyse plus approfondie des propriétés mathématiques des nombres premiers, permettant d'identifier et d'éliminer les faiblesses potentielles du processus de génération. Ces améliorations non seulement renforcent la fiabilité et la sécurité de l'algorithme de génération de clés RSA dans la version 2, mais elles garantissent également une performance accrue, même dans des contextes où des nombres premiers de grande taille sont nécessaires.

1. **Améliorations de la documentation et des commentaires :**

* **Version 1 :** La documentation et les commentaires dans la version 1 du code python RSA étaient assez rudimentaires, offrant des explications générales sur le fonctionnement du système mais manquant parfois de détails spécifiques. Les exemples étaient limités et les références croisées peu nombreuses, ce qui pouvait rendre la compréhension et la maintenance du code plus difficiles. Bien que fonctionnelle, cette approche pouvait laisser certains utilisateurs perplexes quant à certains aspects du code et de son fonctionnement interne.
* **Version 2 :** En revanche, dans la version 2 du code RSA, qui est une version améliorée par rapport à la version 1 de ce code python, une attention particulière plus profonde a été portée à l'amélioration de la documentation et des commentaires. Des exemples détaillés, des explications approfondies et des références croisées ont été ajoutés pour aider les utilisateurs à mieux comprendre le code et ses fonctionnalités. Les commentaires ont été enrichis pour fournir des informations précises sur le fonctionnement interne du système, facilitant ainsi la maintenance et le développement futur. De plus, des sections spécifiques ont été ajoutées pour aborder les scénarios d'utilisation courants et les bonnes pratiques de développement, offrant ainsi aux utilisateurs une ressource complète pour exploiter pleinement les fonctionnalités du code RSA. Ces améliorations visent à rendre la documentation non seulement informative mais aussi pédagogique, aidant ainsi les utilisateurs à maîtriser efficacement le code et à résoudre rapidement les éventuels problèmes rencontrés.

1. **Gestion des exceptions et tests unitaires :**

* **Version 1 :** Dans la version 1 du code python RSA, la gestion des exceptions était assez rudimentaire, se contentant de capturer les erreurs les plus courantes. Bien que cette approche soit fonctionnelle, elle pouvait parfois manquer de précision dans la gestion des cas d'erreur spécifiques, rendant ainsi le processus de débogage et de correction des erreurs plus laborieux. Les développeurs se sont appuyés sur des méthodes standard pour gérer les exceptions, mais cela pouvait entraîner des lacunes dans la manipulation des erreurs plus complexes ou moins prévues. Ainsi, certaines situations exceptionnelles pourraient ne pas être correctement prises en charge, ce qui pourrait compromettre la stabilité et la fiabilité du système dans son ensemble.
* **Version 2 :** Dans la version 2 du code RSA, qui est une version améliorée par rapport à la version 1 de ce code python, des améliorations profondes et significatives ont été apportées à la gestion des exceptions et à l'introduction de tests unitaires pour garantir la fiabilité et la stabilité du système de la version 2 de ce code python RSA. L’auteur du présent document a adopté une approche encore plus proactive, méthodique, structurée et organisée en matière de gestion des erreurs de spécification des messages clairs et pleins fournis par l’utilisateur, en mettant en place des mécanismes plus sophistiqués pour capturer et gérer ces erreurs de manière précise. Cette approche permet non seulement d'identifier et de résoudre les erreurs déjà décrits ci-dessus plus rapidement, mais aussi de garantir une meilleure réactivité et proactivité face aux situations exceptionnelles de spécification du contenu et de la taille des messages clairs et pleins. De plus, l'introduction de tests unitaires offre une méthode systématique pour valider le bon fonctionnement de chaque composant du code, ce qui contribue à améliorer sa qualité et sa robustesse. Ces ajustements renforcent la confiance dans la fiabilité du système, offrant ainsi une expérience utilisateur plus fluide et sécurisée dans la version 2 par rapport à la version précédente.

En combinant ces différentes approches de programmation, Gonzalo Alfredo Romero Francia a considérablement amélioré la version 1du code Python RSA, déjà paramétrable et distribuable, en la rendant une version de code python distribuable et paramétrable, corrigée, améliorée et optimisée, tout en améliorant encore sa lisibilité, sa robustesse et sa facilité d'utilisation.

Le voici une comparaison entre le code python, implémentant l’algorithme RSA, en version 1modifiée, corrigée, améliorée et optimisée par Gonzalo Alfredo Romero Francia (premier code python), qui est déjà en version de base paramétrable et distribuable, corrigée, améliorée et optimisés, et le code python implémentant de manière encore plus efficace cet algorithme asymétrique, et ayant des capacités de distribution et de paramétrisation de paramètres (deuxième code python), ce code python en version 2, a été aussi corrigé, amélioré et optimisé, pour satisfaire aux besoins et aux demandes du projet de fin d’études nommé «  Implémentation et comparaison des algorithmes de chiffrage asymétrique » :

**Changements de lignes de code :**

1. **Importations et initialisations** :

* **Version 1 :** Dans la version 1 du code Python RSA, des importations de librairies python spécifiques telles que import random étaient utilisées, accompagnées d'initialisations de variables globales. Ces importations étaient souvent nombreuses et parfois redondantes, ce qui pouvait rendre le code moins lisible et plus sujet aux erreurs. De plus, les initialisations de variables globales ajoutaient de la complexité au code et rendaient le suivi des variables plus difficile. Bien que fonctionnelle, cette approche pouvait rendre la maintenance et la compréhension du code plus laborieuses.
* **Version 2 :** En revanche, dans la version 2 du code Python RSA, qui est une version améliorée par rapport à la version 1 de ce code python, les importations inutiles de librairies python ont été retirées et les initialisations de variables globales ont été supprimées pour améliorer la lisibilité et la propreté du code. Les développeurs ont adopté une approche plus modulaire, en organisant les importations de manière logique et en limitant leur nombre aux seules bibliothèques nécessaires. De plus, les variables globales ont été remplacées par des variables locales là où cela était possible, réduisant ainsi la portée des variables et rendant le code plus prévisible et plus facile à comprendre. Ces ajustements contribuent à rendre la version 2 plus agréable à travailler et à maintenir que la version 1. Les efforts pour rationaliser les importations et les variables ont permis de rendre le code plus flexible et évolutif, facilitant ainsi son adaptation à de nouveaux besoins et exigences du projet.

1. **Fonctions de primalité :**

* **Version 1 :** Dans la version 1 du code python RSA, la méthode utilisée pour trouver des nombres premiers impliquait l'utilisation du crible d'Eratosthène, une technique classique pour la génération de nombres premiers. Cette méthode était accompagnée d'une fonction de test de primalité basée sur le calcul du PGCD (Plus Grand Commun Diviseur). Bien que cette approche soit largement acceptée et fonctionnelle, elle pouvait parfois présenter des limites en termes de vitesse et d'efficacité, surtout lors du traitement de nombres premiers de grande taille. De plus, la méthode du crible d'Eratosthène pouvait nécessiter des ajustements pour gérer efficacement des volumes de données plus importants, ce qui ajoutait une complexité supplémentaire au code.
* **Version 2 :** En revanche, dans la version 2 du code python RSA, qui est une version améliorée par rapport à la version 1 de ce code python, des améliorations plus significatives ont été apportées aux fonctions de primalité (qui déterminent la primalité des nombres premiers générés par l’algorithme RSA) pour garantir des performances optimales et une fiabilité accrue. L’auteur du présent document a introduit des fonctions spécifiques telles que is\_prime(), generate\_prime\_candidate(), et generate\_prime(), utilisant l'algorithme de Miller-Rabin, reconnu pour sa robustesse et son efficacité dans la génération de nombres premiers et le test de leur primalité. Cette approche offre des avantages significatifs par rapport à la méthode du crible d'Eratosthène, notamment en termes de rapidité et d'évolutivité. De plus, l'utilisation de l'algorithme de Miller-Rabin permet de garantir une meilleure qualité des nombres premiers générés, renforçant ainsi la sécurité et la fiabilité de l'algorithme RSA dans la version 2 du code.

1. **Optimisation de la génération de clés :**

* **Version 1 :** Dans la version 1 du code python RSA, le processus de génération de clés était relativement simple, impliquant des opérations de génération de nombres premiers suivies de calculs basiques pour obtenir les clés publiques et privées. Bien que fonctionnelle, cette approche pouvait parfois manquer d'optimisation et de robustesse, en particulier lors de la manipulation de nombres premiers de grande taille ou dans des conditions de charge élevée.
* **Version 2 :** En revanche, dans la version 2 du code python RSA, qui est une version améliorée par rapport à la version 1 de ce code python, des améliorations plus significatives ont été apportées au processus de génération de clés pour garantir des performances optimales et une sécurité renforcée. L’auteur du présent document a mis en place des algorithmes encore plus sophistiqués pour la génération de nombres premiers et l'obtention des clés publiques et privées, intégrant des techniques avancées telles que le criblage de nombres premiers et l'utilisation de méthodes de calcul plus efficaces. Ces ajustements permettent d'optimiser le processus de génération de clés, réduisant ainsi le temps nécessaire pour obtenir des clés sécurisées tout en garantissant leur fiabilité et leur unicité. En outre, des mesures supplémentaires ont été prises pour renforcer la protection des clés générées contre les attaques potentielles, assurant ainsi une sécurité accrue dans la version 2 du code.

1. **Chiffrement et déchiffrement :**

* **Version 1 :** Dans la version 1 du code Python RSA, les processus de chiffrement et de déchiffrement de messages clairs et pleins était géré à l'aide de boucles de calcul. Bien que cette approche soit fonctionnelle, elle pouvait parfois être sujette à des inefficacités, en particulier lors du traitement de grands volumes de données. Les boucles introduisaient une surcharge supplémentaire, ce qui pouvait entraîner des temps de traitement plus longs et une consommation de ressources accrue. De plus, la gestion manuelle du chiffrement et du déchiffrement pouvait rendre le code moins modulaire et plus difficile à maintenir.
* **Version 2 :** En revanche, dans la version 2 du code Python RSA, qui est une version améliorée par rapport à la version 1 de ce code python, des améliorations significatives ont été apportées aux processus de chiffrement et de déchiffrement de messages clairs et pleins, pour garantir des performances de chiffrement et de déchiffrement de messages optimales et une sécurité plus renforcée pour ces deux processus. L’auteur du présent document a introduit des fonctions python distinctes et bien définies, telles que encrypt() et decrypt(), dédiées à ces opérations spécifiques. Ces fonctions utilisent la fonction pow() pour un calcul plus efficace et optimisé, éliminant ainsi la nécessité d'utiliser des boucles de calcul. Cette approche modulaire permet une meilleure organisation du code et une plus grande lisibilité, facilitant ainsi la maintenance et le débogage. De plus, elle contribue à améliorer les performances globales du système en réduisant la charge de calcul nécessaire pour le chiffrement et le déchiffrement des données.

1. **Encodage et décodage des messages :**

* **Version 1 :** Dans la version 1 du code Python RSA, l'encodage et le décodage des messages étaient effectués en utilisant les valeurs ASCII des caractères. Cette approche, bien qu'efficace dans de nombreux cas, pouvait présenter des limites en termes de clarté et de modularité du code. En utilisant directement les valeurs ASCII, le processus d'encodage et de décodage pouvait être sujet à des erreurs ou à des incohérences, surtout lors du traitement de caractères spéciaux ou de langues autres que l'anglais. De plus, cette méthode ne permettait pas une séparation claire des responsabilités entre le chiffrement/déchiffrement et l'encodage/décodage, ce qui pouvait rendre le code plus difficile à comprendre et à maintenir.
* **Version 2 :** En revanche, dans la version 2 du code Python RSA, des améliorations significatives ont été apportées à l'encodage et au décodage des messages en introduisant des fonctions spécifiques (encode\_message(), decode\_message()). Ces fonctions permettent une séparation claire des responsabilités en isolant le processus d'encodage/décodage dans des blocs de code distincts. En utilisant ces fonctions dédiées, le code devient plus lisible, modulaire et facile à maintenir. De plus, cette approche offre une flexibilité accrue, permettant par exemple de changer facilement le jeu de caractères utilisé pour l'encodage sans affecter le reste du code. Ces ajustements contribuent à améliorer la qualité et la robustesse de l'algorithme RSA dans la version 2 du code Python. En outre, les fonctions encode\_message() et decode\_message() ont été conçues pour être facilement extensibles, ce qui signifie qu'elles peuvent être adaptées pour prendre en charge des encodages/décodages spécifiques à des besoins particuliers. Cette modularité accrue permet aux développeurs d'ajuster et de personnaliser le comportement de l'algorithme de chiffrement/déchiffrement en fonction des exigences spécifiques de leur application, offrant ainsi une plus grande souplesse et une meilleure adaptabilité.

1. **Exécution principale :**

* **Version 1 :** Dans la version 1 du code Python RSA, l'exécution principale était contenue dans un bloc conditionnel if name == 'main':, avec l’ajout d’une fonction rsa\_demo(). Cette approche permettait de fournir une démonstration pratique du fonctionnement de l'algorithme RSA, offrant ainsi aux utilisateurs un aperçu clair des capacités et des performances du code. Cependant, bien que fonctionnelle, cette implémentation pouvait parfois manquer de clarté et de modularité, en particulier si d'autres fonctionnalités devaient être ajoutées à l'exécution principale. De plus, la présence d'une seule fonction démo pouvait limiter la flexibilité du code pour la mise en œuvre de scénarios personnalisés ou avancés.
* **Version 2 :** En revanche, dans la version 2 du code Python RSA, une attention particulière a été portée à l'organisation et à la clarté de l'exécution principale. Tout en conservant le bloc conditionnel if name == 'main':, la version 2 a ajouté une fonction rsa\_demo() distincte pour démontrer le fonctionnement de l'algorithme. Cette approche modulaire permet une meilleure séparation des responsabilités et offre une plus grande flexibilité pour l'ajout de fonctionnalités supplémentaires à l'exécution principale. De plus, en isolant la démonstration dans une fonction dédiée, le code devient plus lisible et plus facile à comprendre, ce qui améliore l'expérience globale de développement et d'utilisation. Ces ajustements contribuent à rendre la version 2 du code Python RSA plus conviviale et plus adaptable aux besoins spécifiques des utilisateurs.

**Approches de programmation :**

1. **Modularité et Séparation des Responsabilités :**

* **Version 1 :** Dans la version 1 du code Python RSA, bien que fonctionnelle, l'approche de programmation était souvent moins modulaire, avec des tâches spécifiques regroupées dans des parties plus larges du code. Cette organisation pouvait parfois rendre le code moins lisible et plus difficile à maintenir, en particulier lorsque des modifications ou des ajouts de fonctionnalités étaient nécessaires. La séparation des responsabilités n'était pas toujours clairement définie, ce qui pouvait entraîner une complexité accrue et des difficultés lors du débogage ou de l'extension du code.
* **Version 2 :** En revanche, dans la version 2 du code Python RSA, une approche plus modulaire a été adoptée pour améliorer la lisibilité et la maintenabilité du code. Des fonctions distinctes ont été définies pour des tâches spécifiques, permettant ainsi une séparation claire des responsabilités. Par exemple, les opérations de chiffrement et de déchiffrement sont encapsulées dans des fonctions distinctes (encrypt(), decrypt()), ce qui rend le code plus facile à comprendre et à gérer. De plus, cette approche modulaire favorise la réutilisation du code et facilite l'ajout de nouvelles fonctionnalités sans perturber le fonctionnement existant. Ces ajustements contribuent à rendre la version 2 du code Python RSA plus robuste et plus évolutive, offrant ainsi une expérience de développement améliorée. L'adoption d'une architecture modulaire permet également une meilleure gestion des exceptions et des erreurs, avec des blocs de code spécifiques dédiés à la gestion des cas d'erreur, améliorant ainsi la fiabilité et la stabilité globales du système.

1. **Utilisation de Fonctions et de Structures de Données Stochastiques :**

* **Version 1 :** Dans la version 1 du code Python RSA, l'utilisation de fonctions et de structures de données stochastiques était limitée. Les générateurs de nombres aléatoires de base, tels que random.randint(), étaient parfois utilisés pour des tâches spécifiques, mais leur utilisation n'était pas généralisée. De plus, les structures de données utilisées pour stocker les messages encodés pouvaient être basiques, telles que des tableaux ou des chaînes de caractères, sans optimisations particulières pour la gestion efficace des données.
* **Version 2 :** En revanche, dans la version 2 du code Python RSA, qui est une version améliorée par rapport à la version 1 de ce code python, représente une amélioration plus significative par rapport à cette version 1 de ce code python, car des progrès importants ont été réalisés dans l'utilisation de fonctions et de structures de données stochastiques. L’auteur du présent document a fait un usage plus étendu des fonctions Python intégrées dans des librairies python, telles que random.randint() pour la génération de nombres aléatoires, garantissant ainsi une distribution plus uniforme et prévisible des valeurs aléatoires. De plus, des structures de données plus appropriées, comme les listes, ont été utilisées pour stocker les messages encodés, offrant une meilleure efficacité et une gestion plus souple des données. Ces améliorations contribuent à renforcer la fiabilité et la robustesse de l'algorithme RSA dans la version 2 du code Python, en garantissant une manipulation plus précise et efficace des données stochastiques.

**3. Utilisation d'Algorithmes et de Techniques Optimisés :**

* **Version 1 :** Dans la version 1 du code Python RSA, les techniques utilisées pour tester la primalité des nombres premiers étaient principalement basées sur la méthode de l’algorithme nommé crible d'Eratosthène. Bien que cette méthode soit largement acceptée et fonctionnelle, elle peut devenir inefficace lors du traitement de nombres premiers de grande taille. De plus, cette approche peut nécessiter des ajustements pour garantir une génération efficace de nombres premiers, ajoutant ainsi de la complexité au code.
* **Version 2 :** En revanche, dans la version 2 du code Python RSA, une approche encore plus avancée a été adoptée, en utilisant l'algorithme de calcul nommé algorithme de Miller-Rabin pour tester la primalité des nombres premiers générés par le code python implémentant l’algorithme RSA. Connu pour sa rapidité et sa fiabilité, cet algorithme offre des performances supérieures à la méthode de crible d'Eratosthène, surtout pour des nombres premiers de grande taille. L'utilisation de l'algorithme de Miller-Rabin renforce ainsi la sécurité et l'efficacité de l'algorithme RSA dans la version 2 du code Python, assurant une génération et une utilisation plus efficaces des nombres premiers. De plus, des techniques d'optimisation spécifiques ont été mises en œuvre pour accélérer davantage le processus de test de primalité, réduisant ainsi le temps d'exécution global de l'algorithme RSA. Ces améliorations garantissent une meilleure performance et une plus grande fiabilité de l'algorithme dans des scénarios d'utilisation réels, renforçant ainsi sa pertinence et son utilité.

**4. Validation et Gestion des Erreurs :**

* **Version 1 :** Dans la version 1 du code Python RSA, la validation des entrées utilisateur et la gestion des erreurs étaient présentes mais pouvaient être limitées. Les vérifications se concentraient généralement sur les erreurs les plus courantes, telles que les saisies incorrectes ou les données manquantes, mais elles pouvaient ne pas prendre en compte toutes les éventualités possibles. Cette approche, bien qu'utile pour traiter les cas les plus fréquents, laissait potentiellement le programme vulnérable aux comportements inattendus ou aux entrées malveillantes. De plus, les mécanismes de gestion des erreurs étaient souvent réactifs plutôt que proactifs, ce qui signifie qu'ils étaient conçus pour réagir aux erreurs une fois qu'elles s'étaient produites plutôt que de les anticiper et de les prévenir à l'avance.
* **Version 2 :** En revanche, dans la version 2 du code Python RSA, des améliorations significatives ont été apportées à la validation et à la gestion des erreurs de frappe des caractères d’un message clair et plein spécifié par l’utilisateur, en ce qui a trait le contenu et la taille de ce message clair et plein. De plus, des vérifications supplémentaires ont été ajoutées pour valider les entrées utilisateur, qui spécifie un message clair et plein en termes de contenu et de taille de ce message, garantissant ainsi un comportement correct du programme même dans des scénarios moins courants. De plus, des mécanismes de gestion des erreurs plus robustes ont été mis en place pour capturer et traiter les erreurs de manière plus précise, offrant ainsi une expérience utilisateur plus fiable et plus prévisible. Ces ajustements renforcent la qualité et la fiabilité du code dans la version 2 du code Python RSA, garantissant un fonctionnement plus fluide et plus sûr du programme. En outre, des techniques avancées de gestion des erreurs ont été implémentées, telles que la journalisation détaillée des erreurs et la prise en charge de scénarios de récupération d'erreur sophistiqués, permettant ainsi aux utilisateurs de comprendre et de résoudre rapidement les problèmes éventuels.

1. **Clarté du Code et Documentation :**

* **Version 1 :** Dans la version initiale du code RSA, la clarté du code et la qualité de la documentation étaient des aspects à améliorer. Les commentaires étaient sporadiques et souvent limités à des explications de haut niveau, ne détaillant pas les subtilités de l'implémentation. De plus, les noms de variables et de fonctions pouvaient parfois manquer de descriptivité, ce qui rendait la compréhension du code plus difficile pour les nouveaux arrivants au projet. Cette approche pouvait entraîner des difficultés lors de la maintenance du code et de la résolution des problèmes, car les développeurs devaient parfois passer du temps supplémentaire à déchiffrer le fonctionnement du code existant.
* **Version 2 :** En revanche, la version 2 du code python RSA, qui est une version améliorée par rapport à cette version 1 de ce code python RSA, a mis et établi un point de haut niveau d’optimisation à améliorer la clarté du code et la qualité de la documentation. Les commentaires ont été révisés et étendus pour fournir des explications détaillées sur le fonctionnement interne de chaque fonction et algorithme. Les développeurs ont pris le temps de choisir des noms de variables et de fonctions significatifs, reflétant leur rôle et leur utilisation dans le contexte du code. De plus, la documentation externe a été enrichie avec des guides d'utilisation complets, des exemples d'utilisation et des références croisées pour aider les utilisateurs à naviguer plus facilement dans le code. Cette approche garantit que le code est plus lisible, compréhensible et maintenable, ce qui facilite le développement continu et la collaboration au sein de l'équipe.

1. **Performance et Optimisation :**

* **Version 1 :** Dans la version 1 du code python RSA, l'optimisation des performances offertes par cette version 1 du code python RSA n'était pas une priorité majeure pour ce code python RSA. Les algorithmes utilisés dans les fonctions modélisant tous les processus associés à l’algorithme RSA, qui sont les processus de génération de clés publiques et privées, chiffrement et déchiffrement de messages clairs et pleins, étaient généralement des choix de base, offrant une fonctionnalité correcte mais ne cherchant pas nécessairement à maximiser l'efficacité ou la vitesse d'exécution. Cette approche pouvait entraîner des performances sous-optimales dans des cas d'utilisation exigeants, tels que le traitement de grandes quantités de données ou le chiffrement/déchiffrement intensif.
* **Version 2 :** En revanche, la version 2 du code python RSA, qui est une version améliorée par rapport à la version 1 de ce code python, a mis un accent particulier et très détaillé et approfondi sur l'optimisation des performances offertes par cette version 2 du code python RSA . L’auteur du présent document a complètement réévalué chaque aspect de l'algorithme RSA pour identifier les opportunités de correction, d’amélioration, d'optimisation et de rationalisation de toutes les fonctions de son code python RSA modélisant tous les processus associés à cet algorithme, qui sont la génération de clés publiques et privées, chiffrement et déchiffrement de messages clairs et pleins. Des techniques avancées ont été utilisées pour accélérer les opérations critiques, telles que l'exploitation de parallélisme, l'utilisation de structures de données optimisées et l'application de stratégies de mise en cache. De plus, des tests de performance rigoureux ont été effectués pour évaluer les améliorations et garantir que les changements apportés n'avaient pas d'effets secondaires indésirables. Cette approche a permis d'obtenir des gains significatifs en termes d'efficacité globale de l'algorithme, offrant ainsi une meilleure expérience utilisateur et une plus grande capacité à traiter des charges de travail intensives**.**

En conclusion, la deuxième version de code Python implémentant l’algorithme RSA présente des améliorations plus que significatives en termes de modularité, de clarté du code et de performances, par rapport à la première version de code Python implémentant cet algorithme, grâce à une meilleure utilisation des fonctionnalités Python et à une approche encore plus efficace des problèmes algorithmiques sous-jacents, par rapport à ce que fournit le code python de base implémentant cet algorithme de chiffrement et de déchiffrement asymétrique.

Finalement, et afin de montrer clairement comment Gonzalo Alfredo Romero Francia a réussi à modifier toutes les lignes de code du programme Python implémentant l’algorithme RSA, montré aux pages 3 à 8 de l’annexe du présent document, qui est fourni à part de ce document, afin de réussir à obtenir le code Python montré aux pages 8 à 11 de ce même annexe du présent document déjà décrit ci-dessus, il a décidé de fournir les informations suivantes :

Examinons plus en détail les changements de lignes de code ainsi que les approches de programmation qui ont été modifiées ou ajoutées pour passer du premier code au deuxième code Python, qui implémente l’algorithme de chiffrement t et de déchiffrement asymétrique RSA.

**Changements de lignes de code :**

* + 1. **Importations et Initialisations :**
* **Version 1 :** Dans la version initiale du code Python RSA, les importations spécifiques telles que import random étaient courantes, accompagnées d'initialisations de variables globales. Bien que cette approche ait permis de démarrer rapidement le développement, elle a également introduit de la complexité et de la confusion dans le code. Les importations inutiles ont parfois encombré l'espace de noms, tandis que les variables globales pouvaient rendre le suivi de l'état du programme difficile. Cette pratique, bien que fonctionnelle, pouvait compromettre la lisibilité et la maintenabilité du code à mesure que le projet évoluait.
* **Version 2 :** En revanche, la version 2 du code python RSA, qui est une version améliorée par rapport à la version 1 de ce code python, a pris des mesures plus significatives pour rationaliser les importations et les initialisations, afin d'optimiser davantage la lisibilité et la maintenabilité du code. Les importations inutiles ont été éliminées de manière proactive, réduisant ainsi non seulement le désordre, mais aussi le risque de conflits potentiels entre les modules. Cette démarche a permis de créer un environnement de développement plus propre et plus cohérent, facilitant la navigation et la compréhension du code pour les développeurs. De plus, les variables globales, souvent sources de confusion et de dépendances indésirables, ont été remplacées par des variables locales lorsque cela était possible. Cette substitution a favorisé une meilleure encapsulation des fonctionnalités et a permis de réduire la portée des variables, améliorant ainsi la prédictibilité et la stabilité du code. En adoptant cette approche plus modulaire et plus propre, la version 2 du code Python RSA devient non seulement plus agréable à travailler, mais aussi plus facile à maintenir à long terme, offrant ainsi une base solide pour les futures évolutions et extensions du projet.

1. **Fonctions de Primalité :**

* **Version 1 :** Dans sa première itération, le code RSA utilisait une méthode de crible d'Eratosthène pour trouver des nombres premiers, avec une fonction de test de primalité basée sur le calcul du PGCD (Plus Grand Commun Diviseur). Bien que cette approche soit courante, elle pouvait parfois manquer de précision et de rapidité, en particulier lors de la génération de nombres premiers de grande taille. De plus, le crible d'Eratosthène pouvait nécessiter des ajustements pour gérer efficacement des volumes de données plus importants, ce qui ajoutait une complexité logique et informatique supplémentaire et pas nécessaire au code python RSA en version 1, pour répondre efficacement aux demandes générales du projet de fin d’études.
* **Version 2 :** En revanche, la version 2 du code python RSA, qui est une version améliorée par rapport à la version 1 de ce code python, a adopté une approche plus sophistiquée, plus complexe et plus efficace pour modéliser la génération de nombres premiers associés à l’algorithme RSA ainsi que le test de leur primalité, marquant ainsi une évolution encore plus significative par rapport à la première version du code python RSA(version 1). Des fonctions spécifiques telles que is\_prime(), generate\_prime\_candidate(), et generate\_prime() ont été introduites, mettant en œuvre l'algorithme de Miller-Rabin, reconnu pour sa robustesse et son efficacité dans la détermination de la primalité des nombres. Cette transition vers un algorithme de test de primalité plus efficace a permis d'atteindre des performances et une précision considérablement améliorées dans la génération des nombres premiers, renforçant ainsi la fiabilité globale de l'algorithme RSA dans sa deuxième itération. Ces améliorations offrent une base solide pour le chiffrement asymétrique, garantissant une meilleure sécurité et une meilleure protection des données sensibles dans les applications utilisant cet algorithme.

1. **Génération de Clés :**

* **Version 1 :** Dans la version initiale du code RSA, la génération de clés était souvent intégrée à une fonction principale telle que generate\_keys(), ce qui rendait le code moins modulaire et moins organisé. Cette approche pouvait entraîner un enchevêtrement des responsabilités et rendre le code difficile à comprendre et à maintenir à mesure qu'il se développait. De plus, l'absence de séparation claire des fonctionnalités rendait difficile l'identification et la correction des erreurs potentielles. Bien que cette version ait rempli sa fonction de base, elle présentait des défauts de conception qui pouvaient affecter sa pérennité et sa capacité à évoluer avec les exigences du projet.
* **Version 2 :** En revanche, la version améliorée du code RSA a introduit une approche plus modulaire et plus sophistiquée pour la génération de clés. Une fonction distincte generate\_keys() a été définie pour cette tâche spécifique, favorisant ainsi une meilleure organisation du code et une séparation claire des responsabilités. Cette modularité accrue rend le code plus lisible, plus maintenable et plus évolutif, facilitant ainsi sa maintenance à long terme et son extension pour répondre aux besoins changeants du projet. En déplaçant la génération de clés vers sa propre fonction dédiée, la version 2 du code RSA offre une structure plus cohérente et une meilleure abstraction des fonctionnalités, améliorant ainsi la qualité globale du code et sa convivialité pour les développeurs. De plus, cette approche favorise une meilleure extensibilité du code, permettant aux développeurs d'ajouter de nouvelles fonctionnalités liées à la génération de clés avec une plus grande facilité et une réduction du risque d'introduire des erreurs dans le code existant.

1. **Chiffrement et déchiffrement :**

* **Version 1 :** Dans sa première itération, le code RSA utilisait des boucles pour effectuer à la fois le chiffrement et le déchiffrement des messages. Cette approche, bien que fonctionnelle, pouvait parfois entraîner une inefficacité dans l'exécution du code, notamment pour les messages de grande taille. En effet, le recours à des boucles pouvait ralentir le processus de chiffrement et de déchiffrement, surtout lorsque le nombre d'itérations nécessaire était élevé. De plus, cette méthode pouvait rendre le code plus difficile à lire et à maintenir, en particulier pour les développeurs moins expérimentés, en raison de sa complexité et de sa propension aux erreurs.
* **Version 2 :** En revanche, la version améliorée du code RSA a adopté une approche plus structurée et efficace pour le chiffrement et le déchiffrement des messages. Plutôt que d'utiliser des boucles, des fonctions distinctes (encrypt() et decrypt()) ont été introduites pour ces opérations spécifiques. Ces fonctions exploitent la fonction pow() pour effectuer un calcul plus efficace, ce qui réduit considérablement le temps nécessaire pour chiffrer et déchiffrer les messages. Cette approche offre des performances globales améliorées, en utilisant des opérations arithmétiques plus efficaces pour manipuler les nombres, ce qui se traduit par des temps d'exécution plus rapides et une utilisation optimisée des ressources système. De plus, en utilisant des fonctions distinctes pour le chiffrement et le déchiffrement, le code devient plus modulaire et plus facile à comprendre, ce qui facilite sa maintenance et son évolutivité à long terme. En résumé, la version 2 du code RSA apporte des améliorations significatives en termes d'efficacité et de lisibilité, renforçant ainsi la robustesse et la praticité de l'algorithme de chiffrement asymétrique.

1. **Encodage et décodage des messages :**

* **Version 1 :** la version 1 du code python RSA utilisait les valeurs ASCII standards des caractères alpha-numériques pour réaliser les processus d'encodage et de décodage des messages clairs et pleins. Cette approche, bien que fonctionnelle, présentait quelques limitations en termes de modularité et de flexibilité. En effet, le code avait tendance à mêler les opérations d'encodage et de décodage avec d'autres fonctionnalités, ce qui rendait parfois difficile la compréhension et la maintenance du système. De plus, cette approche ne permettait pas une séparation claire des responsabilités, ce qui pouvait compliquer l'ajout de nouvelles fonctionnalités ou la correction d'éventuels problèmes.
* **Version 2 :** En revanche, la version 2 du code Python RSA, significativement améliorée par rapport à sa précédente itération, a introduit des fonctions spécifiques, minutieusement spécifiées, définies et structurées (encode\_message() et decode\_message()) pour réaliser les processus d'encodage et de décodage des messages clairs et pleins. Cette démarche de modularisation renforcée et rigoureusement structurée permet une séparation nette des responsabilités entre les calculs mathématiques et logiques, conférant ainsi au code une lisibilité accrue et une facilité de maintenance améliorée. De plus, cette approche offre une flexibilité étendue, facilitant l'intégration de nouvelles fonctionnalités liées à l'encodage et au décodage des messages. En somme, la version 2 du code RSA présente une solution plus élégante et plus modulaire pour ces processus, ce qui se traduit par une amélioration significative de la qualité globale du système et renforce sa praticité et sa fiabilité. Ce nouvel agencement garantit également une meilleure évolutivité du système, permettant une intégration fluide de futures améliorations et fonctionnalités.

1. **Exécution principale :**

* **Version 1 :** Dans la version du code python RSA, l'exécution principale de ce code python RSA était contenue, définie et programmée dans un bloc de code python principal (la classe de ce bloc de code est nommé main) conditionnel if name == 'main':. Bien que cette approche soit courante en informatique et en TI, elle pouvait parfois rendre le code moins lisible et un peu plus difficile à comprendre, en particulier pour des nouveaux développeurs qui deviennent des utilisateurs de ce code python RSA en version 1. De plus, cette méthode ne permettait pas de démontrer facilement le fonctionnement de l'algorithme, ce qui pouvait limiter sa compréhension par les utilisateurs finaux.
* **Version 2 :** En revanche, la version 2 du code python RSA, qui est une version améliorée par rapport à la version 1 de ce code python, a définie et ajouté une fonction rsa\_demo() pour démontrer le fonctionnement de l'algorithme de manière plus détaillée et interactive. Cette fonctionnalité additionnelle rend le code plus convivial et plus facile à comprendre, en permettant aux utilisateurs de visualiser concrètement le processus de chiffrement et de déchiffrement des messages à travers des exemples pratiques. De plus, la fonction rsa\_demo() fournit des informations supplémentaires sur les étapes clés de l'algorithme, offrant ainsi une meilleure compréhension de son fonctionnement global. Cette approche enrichie améliore significativement l'accessibilité du code, facilitant ainsi son utilisation par un large éventail d'utilisateurs finaux, même ceux qui ne sont pas familiers avec les concepts sous-jacents de la cryptographie asymétrique. En résumé, l'ajout de la fonction rsa\_demo() dans la version 2 du code RSA renforce considérablement sa convivialité et sa facilité d'utilisation, offrant ainsi une meilleure expérience aux utilisateurs finaux et contribuant à la diffusion plus large de l'algorithme dans divers domaines d'application.

**Approches de programmation :**

1. **Modularité et Séparation des Responsabilités :**

* **Version 1 :** Dans la version 1 du code python RSA, la modularité des fonctions de cette version 1 du code python RSA était limitée, car ce code python RSA en version 1 contient des fonctionnalités souvent regroupées dans des sections plus larges et moins distinctes. Les différentes responsabilités étaient souvent entremêlées, ce qui rendait la compréhension et la maintenance du code plus difficiles pour les développeurs. Cette approche moins modulaire pouvait également rendre la réutilisation des fonctionnalités plus complexe, car elles étaient fortement couplées les unes aux autres. En conséquence, la flexibilité du code était limitée, ce qui pouvait entraver son évolutivité et sa capacité à répondre efficacement aux besoins changeants du projet.
* **Version 2 :** En revanche, la version 2 du code python RSA, qui est une version améliorée par rapport à la version 1 de ce code Python RSA, adopte une approche encore plus modulaire et une meilleure et plus approfondie séparation des responsabilités dans les fonctions définies dans la version 2 du code python RSA. Les fonctionnalités sont soigneusement décomposées en modules distincts, chacun ayant une responsabilité spécifique et clairement définie. Cette modularisation accrue rend le code plus flexible et évolutif, permettant une adaptation plus aisée aux exigences changeantes du projet. De plus, cette approche favorise la réutilisation des fonctionnalités, car elles sont conçues de manière à être indépendantes les unes des autres. Cela facilite la maintenance du code et améliore sa lisibilité, tout en réduisant les risques d'erreurs et en favorisant une meilleure collaboration entre les membres de l'équipe de développement. En résumé, la version 2 du code Python RSA offre une architecture plus robuste et mieux organisée, ce qui contribue à renforcer sa qualité globale et sa pérennité dans le temps.

1. **Utilisation de Fonctions et de Structures de Données Stochastiques :**

* **Version 1 :** Dans la version 1 du code python RSA, la génération de nombres aléatoires dans le code Python RSA pouvait être réalisée à l'aide de méthodes plus conventionnelles ou de bibliothèques tierces, sans suivre une approche standardisée. Cette approche pouvait entraîner une variabilité dans la génération des nombres aléatoires et rendre le processus moins prévisible. De même, les structures de données utilisées pour stocker les messages encodés pouvaient ne pas être optimales, ce qui pouvait affecter l'efficacité et la lisibilité du code.
* **Version 2 :** En revanche, la version 2 du code python RSA, qui est une version améliorée par rapport à la version 1 de ce code Python RSA, une approche encore plus sophistiquée est adoptée pour la génération de nombres aléatoires. En utilisant des fonctions Python intégrées telles que random.randint(), le processus de génération des nombres aléatoires est standardisé, garantissant ainsi une distribution plus uniforme et prévisible. Cette uniformité renforce la fiabilité du système, en réduisant les variations potentielles dans les résultats de chiffrement et de déchiffrement. De plus, la version 2 privilégie l'utilisation de structures de données appropriées, telles que les listes, pour stocker les messages encodés. Cette approche améliore considérablement l'organisation du code en fournissant une structure plus claire et plus intuitive, ce qui facilite la compréhension et la maintenance du code pour les développeurs. Ces améliorations combinées contribuent à rendre le code plus efficace et plus facile à comprendre, ce qui est essentiel pour garantir la qualité et la robustesse du système dans son ensemble. En résumé, la version 2 du code Python RSA établit une norme plus élevée en termes de génération de nombres aléatoires et de gestion des données, offrant ainsi une base solide pour un code plus fiable et évolutif.

1. **Utilisation d'Algorithmes et de Techniques Optimisés :**

* Version 1 : Dans la version 1 du code python RSA, les méthodes de test de primalité utilisaient des techniques moins avancées telles que le crible d'Eratosthène. Bien que ces approches soient acceptables dans de nombreux cas, elles pouvaient parfois entraîner des performances sous-optimales lors de la génération de nombres premiers. Cela pouvait potentiellement compromettre la sécurité du système en introduisant des nombres premiers faibles. Malgré son fonctionnement adéquat dans la plupart des situations, la version 1 du code RSA pouvait bénéficier d'améliorations pour renforcer la robustesse et la sécurité de l'algorithme.
* Version 2 : La version 2 du code python RSA, une itération améliorée et raffinée de son prédécesseur, a apporté des améliorations significatives dans l'utilisation d'algorithmes et de techniques optimisées pour tester la primalité des nombres. Contrairement à la version antérieure, qui s'appuyait sur des méthodes traditionnelles comme le crible d'Eratosthène, la version 2 a opté pour l'algorithme de Miller-Rabin, largement reconnu pour sa précision et son efficacité dans le domaine de la cryptographie. Cette transition vers un algorithme plus sophistiqué a eu un impact considérable sur les performances et la sécurité de l'algorithme RSA. En utilisant des techniques plus avancées et rigoureuses, la version 2 garantit une génération de nombres premiers plus fiable, réduisant ainsi les risques potentiels liés à la cryptographie asymétrique. L'adoption de l'algorithme de Miller-Rabin renforce la robustesse du système et accroît sa résilience face aux attaques malveillantes, consolidant ainsi sa sécurité globale. En somme, la version 2 représente une évolution majeure par rapport à son prédécesseur, offrant des performances améliorées et une sécurité renforcée, ce qui en fait une mise à niveau essentielle pour l'algorithme RSA dans un environnement toujours plus exigeant en matière de sécurité.

1. **Validation et Gestion des Erreurs :**

* **Version 1 :** Dans la version 1 du code python RSA, la validation des entrées fournies par l’utilisateur et la gestion des erreurs faits par celui-ci, à l’heure de spécifier des messages clairs et pleins, étaient relativement basiques. Les contrôles de validation étaient limités, ce qui signifie que les utilisateurs pouvaient potentiellement saisir des données incorrectes ou malveillantes, entraînant des comportements inattendus ou des vulnérabilités de sécurité. Cette approche moins rigoureuse de la validation et de la gestion des erreurs pouvait compromettre la stabilité et la sécurité du système, en laissant place à des scénarios imprévus et à des risques potentiels.
* **Version 2 :** En revanche, la version améliorée du code Python RSA, qui représente toute une avancée significative par rapport à sa précédente itération, a considérablement plus renforcé la validation des entrées fournies par utilisateur ainsi que la gestion des erreurs commises par celui-ci, à l’heure de spécifier des messages clairs et pleins. Des dispositifs de validation plus rigoureux ont été implémentés pour examiner méticuleusement chaque entrée utilisateur, assurant ainsi l'intégrité et la sécurité des données saisies. De plus, la version 2 intègre une approche proactive de la gestion des erreurs, dotée de mécanismes de détection et de correction plus élaborés. Ceci permet non seulement d'anticiper et de rectifier les erreurs potentielles avant qu'elles ne compromettent le bon fonctionnement du programme, mais également de fournir des messages d'erreur plus informatifs et des solutions de contournement pour les cas rencontrés. En résumé, la version 2 offre une démarche plus solide et sécurisée en matière de validation et de gestion des erreurs, garantissant ainsi la fiabilité, la stabilité et l'expérience utilisateur améliorée du système dans sa globalité. Cette approche proactive et minutieuse de la gestion des erreurs témoigne de l'engagement envers l'excellence et la qualité du développement logiciel, renforçant ainsi la confiance des utilisateurs et la robustesse du système.

1. **Clarté du Code et Documentation :**

* **Version 1 :** La version 1 du code Python RSA souffrait fréquemment d'un manque de documentation et de commentaires adéquats, ce qui pouvait rendre le code difficile à comprendre pour les développeurs. Le manque de documentation claire et de commentaires explicatifs rendait souvent difficile la compréhension des fonctionnalités spécifiques du code, en particulier pour les membres de l'équipe qui n'avaient pas été impliqués dans son développement initial. Cette lacune en matière de documentation pouvait entraver la maintenance du code à long terme et compliquer la résolution des problèmes rencontrés.
* **Version 2 :** En revanche, la version 2 du code python RSA, qui représente une évolution significative par rapport à sa prédécesseur, met un accent considérable sur l'élaboration d'une documentation encore plus exhaustive et sur des commentaires encore plus détaillés, contribuant ainsi à une clarté et une compréhensibilité accrues du code. Les développeurs ont investi un temps précieux pour fournir des explications approfondies sur le fonctionnement précis de chaque composant du code, ainsi que des commentaires explicatifs pour éclaircir les intentions derrière chaque décision de conception. De plus, les choix de dénomination des variables et des fonctions ont été méticuleusement sélectionnés pour refléter leur usage et leur contexte, rendant ainsi le code plus intuitif et facile à appréhender. Cette démarche renforcée en matière de documentation et de commentaires favorise une collaboration encore plus étroite entre les membres de l'équipe de développement, permettant une compréhension plus profonde du code et facilitant ainsi la résolution des problèmes éventuels. Cette amélioration notable de la documentation et des commentaires garantit une maintenance plus fluide et une évolutivité accrue du code sur le long terme, offrant ainsi une base solide pour le développement continu du projet.

1. **Performance et Optimisation :**

* **Version 1 :** La version 1 du code Python RSA mettait moins l'accent sur l'optimisation des performances de toutes les fonctions réalisant des opérations et des calculs mathématiques associés aux processus de génération de clés publiques et privées, chiffrement et déchiffrement de messages clairs et pleins, ce qui pouvait entraîner des temps de traitement plus longs et une utilisation inefficace des ressources système. Les algorithmes utilisés n'étaient pas toujours les plus efficaces, ce qui pouvait limiter les performances globales du système, en particulier lors de la génération de clés RSA et du chiffrement/déchiffrement des données. **Version 2 :** En revanche, la version 2 du code python RSA, qui représente une itération notable par rapport à son prédécesseur, accentue encore davantage son engagement envers l'optimisation des performances de toutes les fonctions impliquées dans les opérations mathématiques liées à la génération de clés publiques et privées, ainsi que sur l'utilisation d'algorithmes encore plus efficaces sur le plan logique et informatique. Par exemple, l'adoption de l'algorithme de Miller-Rabin pour tester la primalité des nombres premiers offre une amélioration significative de l'efficacité lors de la génération des clés RSA, en réduisant considérablement le temps nécessaire pour cette tâche critique. De plus, des techniques d'optimisation avancées, telles que l'utilisation de structures de données adaptées et l'optimisation des opérations arithmétiques, contribuent à réduire les temps de traitement et à minimiser l'utilisation des ressources système. Ces améliorations globales garantissent des performances améliorées et une utilisation plus efficace des ressources, ce qui est essentiel pour les applications exigeant des opérations cryptographiques rapides et sécurisées. En résumé, la version 2 du code RSA se positionne comme une mise à jour significative, offrant des performances optimisées et une sécurité renforcée pour répondre aux besoins croissants en matière de sécurité informatique.

En résumé, les changements de lignes de code et les approches de programmation effectués dans la version 2 du code Python implémentant l’algorithme RSA, ont permis d'améliorer la performance, la lisibilité et la maintenabilité du code, tout en ajoutant des fonctionnalités supplémentaires telles que la validation des entrées utilisateur et l'utilisation d'algorithmes plus efficaces pour les opérations mathématiques critiques.

Le code Python implémentant l’algorithme RSA, montré aux pages 8 à 11 de l’annexe du présent document, qui est fourni à part de ce document, est le code Python implémentant l’Algorithme RSA, qui sera utilisé pour base pour obtenir le code Python, en version finale ayant tous les blocs de code python implémentant les 9 critères d’évaluation de ce code python en version finale (ces critères sont montrés à la page 7 du présent document), et qui va satisfaire aux demandes de l’énoncé du projet de fin d’études « Implémentation et comparaison des algorithmes de chiffrage asymétrique », le développement de tous les points en lien avec la réalisation de cette tâche sera développé en détail dans le présent travail au noté #3 d’INF 1430.

Il est à noter que le code python implémentant l’algorithme RSA montré aux pages 8 à 11 de l’annexe du présent document, fourni à part de ce document, **servira comme code python de base pour obtenir le code python en version finale, auquel l’auteur du présent document va insérer des blocs de code python implémentant les critères d’évaluation montrés à la page 8 du présent document.**

**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.