

projet de semestre

AdaRSA

Daniel Lifschitz et Nicolas Seriot,
classe EI2B, EIVD*, Yverdon.

13 juillet 2003

Résumé

Nous présentons ici un programme de cryptographie. Ce programme permet de générer et de conserver des clés RSA de 128 à 2048 bits. La génération des clés se fait de manière didactique; l'utilisateur est informé des différentes étapes. Le programme permet aussi de chiffrer et déchiffrer des messages au format texte. Enfin, il propose différentes fonctions ayant trait à la théorie des nombres, comme des tests de primalité probabilistes, la factorisation de grands nombres ou la génération de grands nombres premiers aléatoires.

Une grande partie du travail réalisé a consisté en l'implémentation d'un type numérique permettant de travailler sur de très grands entiers, bien supérieurs au plus grand des entiers codés sur 32 bits. Le programme effectuant des calculs intensifs, il a fallu implémenter les fonctions arithmétiques sur ce type de manière efficace.

Le programme fonctionne bien et son développement s'est révélé passionnant. Nous espérons avoir exposé clairement les concepts mathématiques mis en œuvre et les détails de leur implémentation.

*<http://www.eivd.ch/>

Merci à Jean-François Hêche pour l'aide qu'il nous a apportée.

Table des matières

I	Documentation de développement	9
1	Introduction	9
1.1	Introduction à la cryptographie	9
1.2	La cryptographie asymétrique	9
2	L’algorithme RSA	10
2.1	Création des clés	11
2.2	Chiffrement et déchiffrement	12
2.3	Exemple	12
2.3.1	Création des clés	13
2.3.2	Chiffrement	13
2.3.3	Déchiffrement	14
2.4	Preuve	14
2.5	La sécurité de RSA	15
3	La gestion des grands nombres entiers	17
4	Les grands nombres entiers aléatoires	18
5	Tests de primalité	20
5.1	Le crible d’Ératosthène	20
5.2	Les tests probabilistes	21
5.2.1	Le test de Fermat	21
5.2.2	Le test de Miller-Rabin	22
5.3	Conjuguer les tests efficacement	22
5.4	Les tests déterministes	23
6	Factorisation	23
6.1	L’algorithme ρ de Pollard	24
6.1.1	Exemple	24
6.1.2	Application	25
7	Déroulement du projet	25
8	État du projet	25
9	Conclusions	26
9.1	Évolutions futures	26
9.2	Signatures des auteurs	27
II	Documentation utilisateur	29

10 Informations générales concernant le logiciel	29
10.1 But et généralités	29
10.2 Configuration minimale recommandée	30
10.3 Lieu d'existence	30
10.4 Licence	30
11 Installation	30
11.1 Désinstallation	30
12 Mode d'emploi	30
12.1 Démarrage du logiciel	30
12.2 Fenêtre principale	31
12.3 Chiffrement RSA	31
12.3.1 Le cadre <i>Message</i>	32
12.3.2 Le cadre <i>Clé publique</i>	32
12.3.3 Le cadre <i>Clé privée</i>	32
12.4 Génération de clés RSA	33
12.4.1 Le cadre <i>Valeurs intermédiaires</i>	34
12.4.2 Le cadre <i>Clé publique</i>	34
12.4.3 Le cadre <i>Clé privée</i>	35
12.5 Nombres premiers	35
12.5.1 Le cadre <i>Nombre considéré</i>	35
12.5.2 Le cadre <i>Tests de primalité</i>	35
12.5.3 Le cadre <i>Factorisation</i>	37
12.6 Quitter <i>AdaRSA</i>	37
12.7 Messages d'erreurs	37
13 Exemples d'utilisation de <i>AdaRSA</i>	39
13.1 Chiffrement et déchiffrement d'un message	39
13.1.1 Ouverture d'un fichier message	39
13.1.2 Chiffrer le message	39
13.1.3 Déchiffrer le message	39
13.2 Génération d'un couple de clés RSA	39
13.3 Tests de primalité et factorisation	40
14 Mise en garde	40
15 Problèmes connus	40
16 Coordonnées des auteurs	41
III Documentation technique	43

17 Généralités	43
17.1 Langage de programmation	43
17.2 Bibliothèques externes	43
17.3 Compilation	43
17.4 Programme d'installation	43
18 Découpe du programme	43
19 Description des fichiers	44
19.1 Programme principal	44
19.1.1 Interface graphique	44
19.1.2 Réponses aux sollicitations de l'utilisateur	46
19.1.3 Procédures du programme principal	46
19.2 Paquetage <code>conversion_textes_nombres</code>	46
19.2.1 Type <code>T_Séquence</code>	46
19.2.2 Exceptions	46
19.2.3 Procédure <code>Ajouter</code>	47
19.2.4 Procédure <code>Textes_En_Nombres</code>	47
19.2.5 Procédure <code>Nombres_En_Textes</code>	47
19.2.6 Procédure <code>Obtenir_Nombres_G</code>	48
19.2.7 Procédure <code>Obtenir_Textes_G</code>	48
19.3 Paquetage <code>grands_entiers_G</code>	48
19.3.1 Partie formelle générique	48
19.3.2 Types déclarés	49
19.3.3 Constantes	50
19.3.4 Exceptions	50
19.3.5 Opérateurs de comparaison	51
19.3.6 Fonction unaire « + »	51
19.3.7 Fonction unaire « - »	51
19.3.8 Fonction unaire « abs »	51
19.3.9 Fonction binaire « + »	52
19.3.10 Fonction binaire « - »	52
19.3.11 Fonction binaire « * »	53
19.3.12 Procédure <code>Division</code>	53
19.3.13 Fonction « / »	54
19.3.14 Fonction « mod »	55
19.3.15 Fonction « rem »	56
19.3.16 Fonction « ** »	56
19.3.17 Fonction <code>Puissance_Modulo</code>	57
19.3.18 Fonction <code>Racine_Carrée</code>	57
19.3.19 Fonction <code>Log</code>	58
19.3.20 Fonction <code>PGCD</code>	58
19.3.21 Fonction <code>PPCM</code>	58
19.3.22 Fonction <code>Max</code>	58
19.3.23 Fonction <code>Min</code>	58
19.3.24 Fonction <code>Image</code>	59

19.3.25	Fonction Valeur	59
19.4	Paquetage <code>grands_entiers_G-aleatoires_G</code>	59
19.4.1	Exceptions	59
19.4.2	Procédure Initialise_Aléatoire	59
19.4.3	Fonction Aléatoire	60
19.5	Paquetage <code>jewl_windows_extension</code>	60
19.5.1	Fonction Multiline	60
19.6	Paquetage <code>nombres_premiers</code>	61
19.6.1	Exceptions	61
19.6.2	Constantes	61
19.6.3	Fonction Crible	61
19.6.4	Fonction Est_Premier_Crible (sur le type Positive)	62
19.6.5	Fonction Est_Premier_Crible (sur le type T_Long_Entier)	62
19.6.6	Procédure Afficher_Crible	62
19.6.7	Fonction Premiers_Entre_Eux	63
19.6.8	Fonction Est_Premier_Fermat	63
19.6.9	Fonction Est_Premier_Miller_Rabin	63
19.6.10	Fonction Est_Premier	64
19.6.11	Fonction Premier_Aléatoire	64
19.6.12	Fonction Factorisation	64
19.7	Paquetage <code>queue_g</code>	65
19.7.1	Partie formelle générique	65
19.7.2	Exceptions	65
19.7.3	Procédure Déposer	65
19.7.4	Procédure Supprimer	66
19.7.5	Procédure Modifier	66
19.7.6	Fonction Tête	66
19.7.7	Procédure Vider	66
19.7.8	Procédure Copier	66
19.7.9	Fonction Longueur	67
19.7.10	Fonction Est_Vide	67
19.7.11	Procédure Parcourir	67
19.8	Paquetage <code>rsa</code>	67
19.8.1	Exceptions	67
19.8.2	Type T_Clés	68
19.8.3	Type T_Clé_Publique	68
19.8.4	Type T_Clé_Privée	68
19.8.5	Type T_Couple_Clés	68
19.8.6	Type T_Souche	68
19.8.7	Fonction Générer_Souche	69
19.8.8	Fonction Générer_Clés	69
19.8.9	Fonction Chiffrer	69
19.8.10	Fonction Déchiffrer	69

IV	Annexes	71
A	Cahier des charges	73
B	Journal du projet	75
B.1	Début avril	75
B.2	Fin avril et début mai	75
B.3	Fin mai	76
B.4	Début juin	76
B.5	Mi-juin	77
B.6	Fin juin et début juillet	78
C	Formulaire et tables	79
C.1	Théorème fondamental de l'arithmétique	79
C.2	Théorème des nombres premiers	79
C.3	La fonction indicatrice de Euler	79
C.4	Le théorème de Bezout	79
C.5	Le théorème de Euler	79
C.6	Le petit théorème de Fermat	80
C.7	L'algorithme d'Euclide	80
C.8	L'algorithme d'Euclide étendu	80
C.9	Les clés RSA	81
D	Jeux de tests	83
D.1	Cadre <i>Chiffrement RSA</i>	83
D.2	Cadre <i>Nombres premiers</i>	83
E	Contenu du CD-ROM	85
F	Listages complets	87

Liste des algorithmes

1	Génération d'un couple de clés RSA	12
2	Chiffrement RSA	12
3	Déchiffrement RSA	12
4	Crible d'Ératosthène	21
5	Test de Fermat	22
6	Test de Miller-Rabin	23
7	Algorithme Rho de Pollard	24
8	Algorithme d'addition	52
9	Algorithme de soustraction	53
10	Algorithme de multiplication	54
11	Algorithme de division 1	54
12	Algorithme de division 2	55
13	Algorithme de la Puissance modulo	57
14	Algorithme d'Euclide	80
15	Algorithme d'Euclide étendu	81

Première partie

Documentation de développement

1 Introduction

1.1 Introduction à la cryptographie

La cryptographie, c'est la science du secret ; crypter un message consiste à lui appliquer une série de transformations (inversibles) afin de le rendre incompréhensible à toute personne qui n'en est pas le destinataire.

La cryptographie est une discipline ancienne, traditionnellement étudiée par quelques élites travaillant pour des chefs d'états ou des gouvernements. Utilisée depuis l'antiquité à des fins diplomatiques et militaires, elle a souvent joué un rôle méconnu mais important dans l'histoire du monde. Les développements de l'imprimerie, des relations diplomatiques puis des machines et des ordinateurs ont contribué au développement de la cryptographie à travers les siècles.

Dans les années 1970, les publications de chercheurs comme Diffie et Hellmann ont levé le voile sur cette discipline. La compétition économique et le besoin accru de protéger des données des regards indiscrets ont fait de la cryptographie une science publique.

Aujourd'hui, la cryptographie a envahit la vie civile. Elle permet d'assurer la sécurité des cartes à puces et du commerce électronique, de produire des signatures numériques. Elle permet aussi aux dissidents de régimes répressifs de communiquer confidentiellement.

La cryptographie telle qu'on la connaît de nos jours repose sur un principe essentiel établi par Auguste Kerckhoffs dès 1883 dans son article *La cryptographie militaire* : la sécurité d'un système de chiffrement ne doit pas être fondée sur le secret de la procédure qu'il suit, mais uniquement sur un paramètre utilisé lors de sa mise en œuvre, la clé. En effet, tous les algorithmes « secrets » sont analysés et finissent par être publiés, comme c'est le cas pour le CSS qui protège certains DVD, ou le système A5 utilisé dans le protocole GSM. La publicité de l'algorithme assure en outre que les chercheurs pourront contribuer à en déceler les failles et à le rendre plus sûr.

1.2 La cryptographie asymétrique

On distingue généralement deux types de systèmes cryptographiques : symétriques et asymétriques, dits aussi respectivement à clé secrète et à clé publique.

Dans les systèmes symétriques (ou à clé secrète), tels que l'algorithme DES, le codage et le décodage s'effectuent selon le même principe, avec la même clé. Pour garantir que leur communication reste confidentielle, deux interlocuteurs doivent donc avoir pu échanger préalablement la clé de chiffrement par une voie sécurisée.

À l'inverse, dans un système asymétrique (ou à clé publique), chaque individu possède deux clés distinctes. La première clé est publique, elle peut être connue de tous. Cette clé permet de chiffrer des messages que seul le propriétaire de la clé pourra déchiffrer à l'aide d'une clé secrète qu'il est le seul à détenir. L'algorithme RSA utilisé par *AdaRSA* fait partie de cette catégorie d'algorithmes.

Le concept de la cryptographie à clé publique a été présenté au public pour la première fois¹ en 1976 par Whitfield Diffie et Martin Hellman dans l'article *New directions in cryptography*. L'idée générale était d'utiliser une fonction de chiffrement à sens unique, c'est-à-dire une fonction qui soit facile à appliquer mais pratiquement impossible à inverser pour qui ne connaît pas une information que le destinataire est seul à détenir. Le monde physique connaît de telles fonctions. Par exemple, si le mélange de peinture bleue et de peinture jaune est facile à réaliser, leur séparation est ensuite très difficile.

Les algorithmes asymétriques sont environ mille fois plus lents que les algorithmes symétriques et nécessitent des clés dix fois plus grandes. Cependant, le fait que la transmission d'une clé puisse se faire publiquement constitue un énorme avantage. Par exemple, si le propriétaire d'un couple de clés suspecte que sa clé privée est compromise, il peut en changer sans devoir communiquer de secret à ses correspondants. Il lui suffit de publier sa nouvelle clé publique. Dans la pratique (transactions bancaires, commerce électronique), pour des raisons de performance, la cryptographie à clé publique est souvent utilisée en combinaison avec la cryptographie à clé secrète. On obtient alors un système hybride alliant performance et souplesse dans la gestion des clés.

La cryptographie asymétrique peut être utilisée pour chiffrer des messages mais aussi pour authentifier l'auteur d'un document, c'est-à-dire qu'elle peut servir à la mise en œuvre de systèmes de signature numérique. Nous n'abordons pas cet aspect là dans notre travail.

2 L'algorithme RSA

De tous les algorithmes à clé publique, RSA est de loin le plus simple à comprendre et à implémenter. C'est aussi le plus répandu. Son nom vient des initiales de ses inventeurs, Ron Rivest, Adi Shamir et Leonard Adleman. Depuis sa présentation en 1978, il a traversé des années d'analyse sans que les cryptanalystes ne lui trouvent d'attaque efficace. Il est aujourd'hui largement utilisé, que ce soit dans les navigateurs web Netscape et Explorer ou dans certaines cartes bancaires comme les cartes Visa.

Nous présentons ici les algorithmes de génération du couple de clés publiques et privées, de chiffrement et de déchiffrement, un exemple ainsi que la preuve que le système fonctionne.

¹En fait, les services de renseignement britanniques avaient eux aussi découvert ce concept dès le début des années 1970, mais ne purent jamais le mettre en œuvre car personne ne disposait alors d'une puissance de calcul suffisante[2].

2.1 Création des clés

Les clés sont constituées chacune d'un nombre N produit de deux grands nombres premiers aléatoires P et Q . On appelle ce nombre N le module RSA. On désigne aussi par ϕ la fonction indicatrice de Euler (voir C.3) pour le nombre N . Puisque P et Q sont premiers, ϕ est égal à $(P - 1)(Q - 1)$. Chaque clé comporte en plus un autre chiffre ; E premier avec $\phi = (P - 1)(Q - 1)$ pour la clé publique et D tel que $ED \pmod{\phi} = 1$ pour la clé privée.

Clé publique
N produit de P et Q , deux grands nombres premiers aléatoires
E nombre premier avec $\phi = (P - 1)(Q - 1)$

Clé privée
N produit de P et Q , deux grands nombres premiers aléatoires
D tel que $ED \pmod{\phi} = 1$

Détermination de N N est facile à calculer dès lors que l'on peut générer deux grands nombres premiers aléatoires. Nous verrons par la suite que l'obtention de ces deux nombres n'est pas un problème élémentaire (voir 5).

Détermination de E E est également facile à déterminer. On cherche un nombre qui soit premier avec $\phi = (P - 1)(Q - 1)$. On peut donc choisir des nombres au hasard et tester s'ils vérifient la condition. On peut aussi utiliser un nombre premier quelconque, puisque n'importe quel nombre premier est par définition premier avec ϕ , c'est-à-dire que E et ϕ n'ont pas de facteur commun.

Dans la pratique, on utilise généralement les nombres 3, 17 ou 65537, dans le but d'accélérer les calculs informatiques. En effet, $65537 = 2^{16} + 1$, c'est-à-dire que sa représentation binaire² n'a que deux uns et nécessite peu de multiplications lors de son exponentiation. C'est la raison pour laquelle l'exposant public E de *AdaRSA* est 65537.

La fonction de génération d'un E aléatoire est implémentée dans le code source, elle est simplement désactivée par des commentaires.

Détermination de D D est l'inverse multiplicatif de E modulo ϕ . On le calcule en appliquant l'algorithme d'Euclide étendu (voir C.8).

Notons que D et E sont premiers deux à deux. Les deux nombres premiers P et Q ne servent plus à rien. Ils doivent être oubliés ou détruits ; les révéler reviendrait à divulguer la clé privée.

²10000000000000001

Algorithme 1 Génération d'un couple de clés RSA

Sorties: Un couple de clés RSA.

Générer deux très grands nombres premiers P et Q .

Calculer $N = PQ$ et $\phi = (P - 1)(Q - 1)$.

Choisir un entier aléatoire E entre 1 et ϕ , tel que $\text{pgcd}(E, \phi) = 1$.

Utiliser l'algorithme d'Euclide étendu (voir C.8) pour calculer l'entier D compris entre 1 et ϕ tel que $ED \bmod \phi = 1$.

La clé publique est le couple $\{E, N\}$.

La clé privée est le couple $\{D, N\}$.

2.2 Chiffrement et déchiffrement

Pour chiffrer un message M , il faut d'abord le diviser en blocs de chiffres plus petits que N (avec des données binaires, choisir la plus grande puissance de 2 plus petite que N). Si P et Q ont tous les deux 100 chiffres, alors N aura un peu moins de 200 chiffres et chaque bloc de chiffres devra avoir un peu moins de 200 chiffres.

Chiffrement Pour obtenir un message chiffré C à partir d'un message clair M et d'une clé publique $\{E, N\}$ on calcule :

$$C = M^E \bmod N$$

Algorithme 2 Chiffrement RSA

Entrées: Un message clair M , une clé publique $\{E, N\}$.

Sorties: Un message chiffré C .

$$C = M^E \bmod N$$

Déchiffrement Pour restituer un message clair M à partir d'un message chiffré C et d'une clé privée $\{D, N\}$ on calcule :

$$M = C^D \bmod N$$

Algorithme 3 Déchiffrement RSA

Entrées: Un message chiffré C , une clé publique $\{D, N\}$.

Sorties: Le message clair M .

$$M = C^D \bmod N$$

2.3 Exemple

Voici un exemple détaillé de ce qui se passe quand Bernard veut envoyer un message chiffré à Alice³.

³La littérature a coutume d'appeler le destinataire et l'expéditeur du message respectivement Alice et Bernard. Nous suivons ici cette convention.

2.3.1 Création des clés

Pour se fabriquer un couple de clés, Alice commence par choisir deux nombres premiers aléatoires. Comme il s'agit ici d'un exemple, elle choisit deux petits nombres premiers.

$$\begin{aligned}P &= 47 \\Q &= 71\end{aligned}$$

Elle peut alors calculer N et ϕ .

$$\begin{aligned}N &= P \cdot Q \\&= 3337\end{aligned}$$

$$\begin{aligned}\phi &= (P - 1)(Q - 1) \\&= 46 \cdot 70 \\&= 3220\end{aligned}$$

Alice choisit aléatoirement l'exposant public E premier avec ϕ .

$$E = 79$$

Reste à calculer D , l'inverse de $E \pmod{\phi}$. Pour cela, Alice peut utiliser l'algorithme étendu de Euclide (voir C.8 page 80).

$$\begin{aligned}ED \pmod{\phi} &= 1 \\D &= E^{-1} \pmod{\phi} \\&= 79^{-1} \pmod{3220} \\&= 1019\end{aligned}$$

Alice peut maintenant publier sa clé publique constituée de N et E . Elle efface ou détruit P et Q et conserve précieusement D , qui est la clé secrète permettant de déchiffrer les messages qui lui seront envoyés.

2.3.2 Chiffrement

Bernard utilise la clé publique de Alice pour chiffrer un message à son attention. Dans notre exemple, le message de Bernard est 6882326879666683. Si le message est un texte, Bernard le transforme en nombre à l'aide d'une convention publique de codage, comme le code ASCII par exemple. Bernard commence par découper son message en blocs de chiffres plus petits que N .

$$\begin{aligned}
 M_1 &= 688 \\
 M_2 &= 232 \\
 M_3 &= 687 \\
 M_4 &= 966 \\
 M_5 &= 668 \\
 M_6 &= 003
 \end{aligned}$$

Le premier bloc est chiffré avec la clé publique de Alice :

$$\begin{aligned}
 C_1 &= M_1^E \pmod{N} \\
 &= 688^{79} \pmod{3337} \\
 &= 1570
 \end{aligned}$$

Bernard répète cette opération sur les autres blocs et obtient le message chiffré :

$$C = 1570 \ 2756 \ 2091 \ 2276 \ 2423 \ 158$$

Il peut maintenant envoyer ce message à Alice par un canal qui n'a pas besoin d'être sûr, comme la messagerie électronique par exemple.

2.3.3 Déchiffrement

Pour déchiffrer le message, Alice utilise sa clé privée. Elle reconstitue le message clair contenu dans le premier bloc :

$$\begin{aligned}
 M_1 &= C_1^D \pmod{N} \\
 &= 1570^{1019} \pmod{3337} \\
 &= 688
 \end{aligned}$$

En répétant l'opération sur les autres blocs, elle retrouve le message clair :

$$688 \ 232 \ 687 \ 966 \ 668 \ 003$$

2.4 Preuve

La preuve formelle que l'algorithme RSA fonctionne s'appuie sur le théorème de Euler. On trouvera ce théorème en annexe (C.5).

D est l'inverse de $E \pmod{\phi(n)}$, donc $ED \equiv 1 \pmod{\phi(n)}$, et donc il existe un k dans \mathbb{Z} tel que $k\phi(n) = ED - 1$, donc $ED = k\phi(n) + 1$.

$$\begin{aligned}
 C^D &= M^{ED} \pmod{N} \\
 &= M^{1+k\phi(n)} \pmod{N} \quad k \in \mathbb{Z} \\
 &= M \cdot (M^{\phi(n)})^k \pmod{N} \\
 &= M \cdot 1 \pmod{N} \quad \text{par le th. de Euler} \\
 &= M \pmod{N} \\
 &= M
 \end{aligned}$$

Puisque M et N sont premiers entre eux, en appliquant le théorème de Euler, on obtient que $M^{\phi(n)} \equiv 1 \pmod{n}$. D'autre part, puisque M est par hypothèse strictement inférieur à N , le résidu de M modulo N est M lui-même. On retrouve donc le message clair.

2.5 La sécurité de RSA

La sécurité de RSA repose sur la difficulté de la factorisation des grands nombres. Recomposer un message clair à partir du message chiffré est par conjecture équivalent à factoriser le produit des deux nombres premiers. Cela n'est pas prouvé et on peut imaginer qu'un jour quelqu'un découvre une manière plus simple de le faire. Cependant, le fait que les cryptanalystes les plus brillants recherchent une telle méthode depuis 1976 et ne l'aient pas encore trouvée permet d'accorder une grande confiance à l'algorithme RSA.

La société *RSA Security*[6] organise un concours qui consiste à factoriser des nombres produits de grands nombres premiers⁴, dans le but de stimuler la recherche. Pour frapper l'imagination, voici deux défis proposés par ce concours.

Une somme de 10'000 \$ est promise à qui factorisera ce nombre de 576 bits (174 chiffres) :

```

18819881292060796383869723946165043980716356337941
73827007633564229888597152346654853190606065047430
45317388011303396716199692321205734031879550656996
221305168759307650257059

```

Pour ce nombre de 2048 bits (617 chiffres), la récompense s'élève à 200'000 \$:

```

25195908475657893494027183240048398571429282126204
03202777713783604366202070759555626401852588078440

```

⁴<http://www.rsasecurity.com/rsalabs/challenges/factoring/>

```

69182906412495150821892985591491761845028084891200
72844992687392807287776735971418347270261896375014
97182469116507761337985909570009733045974880842840
17974291006424586918171951187461215151726546322822
16869987549182422433637259085141865462043576798423
38718477444792073993423658482382428119816381501067
48104516603773060562016196762561338441436038339044
14952634432190114657544454178424020924616515723350
77870774981712577246796292638635637328991215483143
81678998850404453640235273819513786365643912120103
97122822120720357

```

Le 22 août 1999, un groupe de chercheurs est parvenu à factoriser un nombre de 512 bits (155 chiffres) en faisant travailler près de 285 ordinateurs, dont un super-calculateur, pendant près de cinq mois. Voici ce nombre :

```

10941738641570527421809707322040357612003732945449
20599091384213147634998428893478471799725789126733
24976257528997818337970765372440271467435315933543
33897
= 10263959282974110577205419657399167590071656780803
8066803341933521790711307779
x 10660348838016845482092722036001287867920795857598
9291522270608237193062808643

```

Si l'on suppose que la puissance des machines suit la loi empirique de Moore (la puissance de calcul double tous les 18 mois), on estime une date limite de résistance des clés. Ainsi, les clés de 1024 bits (309 chiffres) seront cassables en 2030 et les clés de 2048 bits (617 chiffres) en 2080, à moins qu'une percée théorique ou technique n'y parvienne avant (voir annexe C.9).

En fait, les problèmes de sécurité potentiels résident moins dans la factorisation du module RSA que dans l'implémentation. En effet, il faut que P et Q soient réellement premiers et réellement aléatoires, ce dernier point étant extrêmement compliqué en informatique (voir 4 page 18).

En ce qui concerne la primalité de P et Q , on remarquera tout de suite si ces deux nombres ne sont pas premiers car le chiffrement et le déchiffrement ne pourront s'effectuer correctement.

Il faut encore veiller à ce que P et Q ne soient pas trop proches l'un de l'autre, leur découverte pourrait être facilitée par la recherche d'un diviseur de N depuis sa racine vers zéro.

Le fait que l'exposant E soit toujours le même, comme c'est le cas dans *AdaRSA*, n'affaiblit pas la sécurité. E est de toute façon public et N reste toujours aussi difficile à factoriser. Cependant, il ne devrait pas être égal à 3 comme c'est parfois le cas. En effet, si $E = 3$, il y a de grandes chances pour que M^3 soit inférieur à N . Dans ce cas, le chiffré est $C = M^3$, sans réduction modulaire. L'inversion de la fonction consiste alors à calculer une racine cubique, un calcul aisé pour tout le monde !

On doit aussi garder à l'esprit que la fonction de chiffrage est déterministe. Ainsi, si l'on sait que le chiffré correspond à un message clair qui ne peut être que « oui » ou « non », il suffit pour connaître le contenu du message de chiffrer soi-même les deux possibilités et de comparer les résultats avec le chiffré transmis, qui est soit celui de « oui », soit celui de « non ».

Dans la pratique, on pallie à ces problèmes en insérant des octets particuliers, dont des valeurs aléatoire, devant le message M , afin d'obtenir un entier de la taille de N et un chiffrement probabiliste. *AdaRSA* n'utilise pas cette méthode ; il n'est donc sûr que si l'attaquant potentiel ne peut présumer du contenu exact du message.

Enfin, il ne faut pas oublier que les attaques peuvent être physiques, telles que l'analyse du temps de calcul ou de la consommation électrique, pouvant permettre de retrouver l'exposant privé $D[3]$.

3 La gestion des grands nombres entiers

Notre compilateur Ada n'offrant pas la possibilité de travailler sur des entiers de plusieurs centaines de chiffres, il nous a fallu définir notre propre structure et implémenter les opérations mathématiques standards permettant d'y remédier.

Pour l'esprit humain, la représentation d'un nombre entier est une suite de chiffres de 0 à 9 ordonnés de telle sorte que les chiffres placés à gauche ont une plus grande valeur. Dans notre système décimal, chaque chiffre a une valeur dix fois supérieure à son voisin situé juste à sa droite, le dernier ayant sa propre valeur.

C'est également de cette façon que nous avons implémenté notre type de données définissant les grands entiers à la différence près que la base employée n'est pas celle utilisée dans la notation décimale, mais une puissance de 2'147'483'348 (2^{31}). L'argument majeur pour justifier ce choix est dû au fait que le type prédéfini Integer permet de mémoriser des entiers entre -2^{31} et $2^{31} - 1$ ce qui permet de réduire au maximum l'espace mémoire inutilisé. Concrètement, les nombres sont sauvegardés dans un tableau contraint par un paramètre générique de notre paquetage `Grands_Entiers_G`.

Au début de notre projet, afin de créer un paquetage le plus compact et le plus simple possible, nous avons prévu de mettre uniquement à disposition un type permettant de représenter des entiers naturels positifs. Cette option a dû être entièrement revue

lorsqu'il s'est avéré très compliqué d'effectuer certains calculs permettant de générer des clés RSA sans nombres négatifs. Nous avons donc adjoint à notre type de données un *flag* indiquant si le nombre stocké est positif ou non. De ce fait, de nouvelles fonctions ont fait leur apparition et toutes les fonctions ont été adaptées afin de tenir compte du signe.

Le dernier élément faisant encore partie de notre type de données est un champ indiquant l'indice du tableau contenant le poids le plus fort du nombre actuellement stocké. Bien que pas absolument nécessaire, cet indicateur permet d'accélérer les calculs en évitant de balayer tout le tableau afin de rechercher l'emplacement où débute le nombre, les autres étant à zéro.

4 Les grands nombres entiers aléatoires

La première question que l'on est en droit de se poser est : qu'est-ce qu'un nombre aléatoire ? Par exemple, est-ce que 25 est un nombre aléatoire ? En général, on parle de séquence de nombres aléatoires suivant une certaine distribution, et cela signifie que chaque nombre produit a été obtenu par chance et que celui-ci n'a strictement rien à voir avec les autres nombres de la séquence. De plus, la probabilité qu'un nombre donné soit tiré au sort doit être identique pour chacun des nombres se trouvant dans l'intervalle désiré.

Il existe de nombreuses méthodes permettant de tirer des nombres aléatoirement. Que l'on songe tout simplement au lancer du dé ou au brassage de cartes. Mais dès l'arrivée des ordinateurs, des scientifiques ont cherché des méthodes permettant d'obtenir des nombres aléatoires de manière uniquement logicielle. C'est John von Neumann qui fut le premier à proposer une méthode faisant appel uniquement à des opérations mathématiques. Son idée était d'élever au carré un nombre aléatoire précédemment généré et d'en extraire les chiffres du milieu.

Par exemple, pour la génération d'un nombre à 10 chiffres avec comme valeur précédente 5772156649, on l'élève au carré, ce qui donne 33317792380594909201, donc le nouveau nombre aléatoire est 7923805949.

Mais comment donc une séquence ainsi générée peut-elle être aléatoire alors que chaque nombre est entièrement déterminé par le précédent ? La réponse est que cette séquence n'est pas aléatoire, mais apparaît comme telle. C'est pourquoi ces générateurs sont souvent appelés générateurs pseudo-aléatoires.

Dans la pratique, l'algorithme de John von Neumann souffre d'un grave défaut. En effet, il existe un risque important que la séquence dégénère en un cycle très court. Prenons par exemple un nombre à 4 chiffres. Si à un moment de la séquence le nombre 6100 venait à être tiré, la suite serait : 2100, 4100, 8100, 6100, 2100, ..., raison pour laquelle on lui préfère aujourd'hui d'autres méthodes, telles que les générateurs à congruence linéaire.

Le générateur à congruence linéaire, inventé en 1948 par D. H. Lehmer, est un des algorithmes les plus utilisés aujourd'hui pour produire des nombres aléatoires. Pour comprendre son principe, définissons 4 nombres entiers :

$$\begin{array}{lll} m, & \text{le modulo;} & 0 < m. \\ a, & \text{le multiplicateur;} & 0 \leq a < m. \\ c, & \text{l'incrément;} & 0 \leq c < m. \\ X_0 & \text{le germe;} & 0 \leq X_0 < m. \end{array}$$

La séquence de nombres aléatoires (X_n) peut maintenant être obtenue par la formule suivante :

$$X_{n+1} = (aX_n + c) \pmod{m}, \quad n \geq 0.$$

Par exemple, la séquence obtenue avec $m = 10$ et $X_0 = a = c = 7$ est :

$$7, 6, 9, 0, 7, 6, 9, 0, \dots$$

Comme on peut le constater dans cet exemple, le fait de prendre n'importe quelle valeur de m , a , c et X_0 ne rend pas une séquence forcément aléatoire. Afin d'obtenir une séquence contenant une période maximale, il est nécessaire de choisir judicieusement les valeurs de départ. Donald E. Knuth[1] a proposé les critères suivants :

- c et m doivent être premiers entre eux ;
- $a - 1$ doit être un multiple de p , pour tout p nombre premier diviseur de m ;
- $a - 1$ doit être un multiple de 4 si m est un multiple de 4 ;
- si m est une puissance de 2, le bit de poids faible des nombres produits vaut alternativement 0 et 1.

Même en respectant les critères donnés par Donald E. Knuth, il n'est pas certain que la suite générée ressemble à une suite aléatoire. Prenons l'exemple suivant :

$$X_{n+1} = (31415821 \cdot X_n + 1) \pmod{100000000}$$

Les critères de Donald E. Knuth ont été respectés :

- 1 et 100'000'000 sont bien premiers entre eux. Par conséquent, on est certain qu'il n'y aura pas de problème de blocage ou de série de nombres tous pairs ou impairs. Tous les nombres entre 0 et 99'999'999 vont sortir et chacun une fois tous les 100'000'000 de tirages ;
- $31415821 - 1 = 31415820$ est bien un multiple de 2 et de 5 (les deux seuls nombres premiers qui divisent 100'000'000) ;
- C'est aussi un multiple de 4 (car 100'000'000 est lui-même un multiple de 4).

Alors, regardons ce que ça donne. Par exemple pour $X_0 = 0$, les nombres produits sont :

1, 31415822, 40519863, 62952524, 25482205, 90965306, 70506227, 6817368,
12779129, 29199910, 45776111, 9252132, 22780373, 20481234, 81203115, ...

Vous ne remarquez rien ? Non ? Regardez attentivement le dernier chiffre de chaque nombre. . .

Comme on peut le constater, écrire un algorithme permettant de générer des nombres aléatoires est un travail délicat. Il est difficile d'être certain qu'un générateur donné est bon sans investir d'énormes efforts dans des tests statistiques variés. Nos connaissances actuelles dans ce domaine des mathématiques n'étant de loin pas suffisantes, il nous a paru plus judicieux d'utiliser le générateur aléatoire fourni avec le compilateur ADA. Concrètement, chaque élément du tableau de notre type `T_Long_Entier` se voit attribuer une valeur de la séquence du paquetage `Ada.Numerics.Discrete_Random`.

5 Tests de primalité

Comme nous l'avons vu (2.5, 2.4), le principe et la sécurité de RSA nécessitent de disposer de grands nombres premiers aléatoires. Nous venons d'étudier (4) la problématique des grands nombres aléatoires. Nous cherchons maintenant à obtenir de tels nombres qui soient premiers. Pour rappel, un nombre premier est un nombre plus grand que 1 qui n'admet pas d'autres diviseurs que 1 et lui-même. Il y a une infinité de nombres premiers. Les nombres qui ne sont pas premiers sont dits composés.

Malheureusement, on ne dispose d'aucune formule pour générer des nombres premiers aléatoires. La seule stratégie connue est donc de générer aléatoirement des chiffres et de tester s'ils sont premiers. Mais comment déterminer si un nombre n est premier ?

La première méthode qui vient à l'esprit consiste à essayer de diviser n par tous les chiffres inférieurs \sqrt{n} . Mais plus n est grand, plus cette façon de faire est inefficace ; elle demande un nombre de calculs tel qu'elle devient vite inapplicable.

5.1 Le crible d'Ératosthène

Une autre méthode consiste à tirer parti du fait que tout nombre composé est un produit de plusieurs nombres premiers (théorème fondamental de l'arithmétique, C.1). Dès lors, chercher si n est premier revient à chercher s'il existe un diviseur de n qui soit premier et inférieur à la racine de n . Pour obtenir les nombres premiers plus petits que la racine de n , on peut utiliser la méthode du crible d'Ératosthène (240 av. J.-C.).

On représente d'abord chaque nombre de 2 à n . Ensuite, on considère chacun des nombres en partant de 2 et on supprime ses multiples de la liste. À la fin, il ne reste que les nombres qui n'ont pas de diviseur dans la liste, c'est-à-dire les nombres premiers plus petits que n .

Algorithme 4 Crible d'Ératosthène

Entrées: Un nombre $n > 1$.

Sorties: Les nombres premiers $< n$.

Représenter tous les nombres de 2 à n .

pour i de 2 à n **faire**

Supprimer les multiples de i de la liste.

fin pour

{Les nombres restants sont les nombres premiers $< n$.}

La méthode du crible est parmi les plus efficaces quand n est petit, mais n'est pas adaptée à la détermination de la primalité d'un nombre comme $2^{127} - 1$.

5.2 Les tests probabilistes

Dans la cryptographie à clé publique, les nombres dont on cherche à déterminer la primalité sont très grands, de l'ordre de 10^{100} . On a vu (2.5 page 15) que la recherche de facteurs était un problème extrêmement difficile et très long. Dans la pratique, plutôt que de chercher à factoriser le nombre, on lui fait passer une série de tests. Si n échoue un de ces tests, c'est qu'il est composé. Si n réussit tous les tests, il y a des chances qu'il soit premier. Il existe ainsi plusieurs tests probabilistes. En répétant ces tests un certain nombre de fois, la probabilité qu'un nombre ayant passé tous ces tests soit premier augmente.

D'après le théorème des nombres premiers (voir C.2 page 79), pour un n donné, il y a environ $\frac{n}{\ln n}$ nombres premiers plus petits que n . Ainsi, si l'on choisit un nombre de 500 chiffres au hasard, on a environ une chance sur $\ln 10^{500}$ de tomber sur un nombre premier, c'est-à-dire environ une chance sur 1150. Pour un nombre de 20 chiffres, il faut effectuer en moyenne 46 tests avant de trouver un nombre premier. Cela reste très raisonnable.

5.2.1 Le test de Fermat

Les tests probabilistes de non-primalité s'appuient sur le petit théorème de Fermat (voir C.6). On choisit au hasard un entier a tel que $2 \leq a \leq n - 1$ et on calcule $a^{n-1} \bmod n$. Si on obtient autre chose que 1, c'est que n est composé.

Ce test est assez efficace en pratique. Il a toutefois un défaut. Il existe des nombres composés que le test ne peut espérer repérer. Ces nombres ont la propriété que pour tout $a \in \{2, \dots, n - 1\}$, on a $a^{n-1} \bmod n = 1$, sauf si $\text{pgcd}(a, n) \neq 1$. On les appelle les nombres de Carmichael. Ils sont très rares : il n'y en a que 255 en dessous de 100'000'000,

Algorithme 5 Test de Fermat

Entrées: Un nombre $n > 1$.**Sorties:** Indique si n est composé, n'indique pas s'il est premier.Choisir un entier a au hasard entre 2 et $n - 1$.**si** $\text{pgcd}(a, m) \neq 1$ **alors**STOP, n est composé.**fin****si** $a^{m-1} \bmod m \neq 1$ **alors**STOP, n est composé.**fin**

et 2'163 en dessous de 25'000'000'000. Les nombres de Carmichael inférieurs à 100'000 sont 561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841, 29341, 41041, 46657, 52633, 62745, 63973, et 75361.

La probabilité que le test n'identifie pas un nombre composé serait d'au moins $\frac{1}{2}$ s'il n'y avait pas les nombres de Carmichael. Du fait de leur existence, la probabilité de réussite du test de Fermat ne peut être bornée. Il existe d'autres tests dont la probabilité est de $\frac{1}{2}$, tels que le test de Solovay-Strassen et celui de Lehmann.

La complexité du test de Fermat est en $O(R \cdot n^3)$, pour R répétitions.

5.2.2 Le test de Miller-Rabin

Comme le test de Fermat, le test de Miller-Rabin est un test probabiliste. Il a été présenté en 1976 par Michael Rabin, qui s'est appuyé sur les travaux de Gary Miller.

L'idée est la suivante. Soit m un nombre composé impair. On peut écrire $m - 1 = 2^s t$ avec t impair. Soit encore $b \in \{1, 2, \dots, m - 1\}$. Alors soit $b^t \bmod m = 1$, ou il existe $r \in \{0, \dots, s - 1\}$ tel que $b^{2^r t} \bmod m = -1$.

La probabilité que le test de Miller-Rabin détecte un nombre composé est de $\frac{1}{4}$. Le test de Miller-Rabin de *AdaRSA* est exécuté 10 fois successivement. Ainsi, la probabilité qu'un nombre ayant passé ces 10 tests soit premier est de $1 - (\frac{1}{4})^{10}$, soit 99.9999%.

5.3 Conjuguer les tests efficacement

Disposant de toutes ces méthodes, la manière la plus efficace pour déterminer si un nombre est premier est de lui chercher un diviseur dans le crible d'Ératosthène (5.1 page 20) contenant les nombres premiers inférieurs à 600 (c'est-à-dire les diviseurs des nombres inférieurs à $600^2 = 360000$). Si on ne trouve pas de diviseur, alors on applique un certain nombre de fois le test de Miller-Rabin (10 fois dans *AdaRSA*). Si le nombre n'est pas premier, on en choisit un autre. S'il a passé tous les tests, il est certainement premier. Dans *AdaRSA*, cette probabilité est de 99.9999%.

Algorithme 6 Test de Miller-Rabin

Entrées: Un nombre $m > 1$.**Sorties:** Indique si m est composé, n'indique pas s'il est premier.Déterminer s et t tels que $m - 1 = 2^s t$, t impair.Premier \leftarrow TRUE.**pour** $i = 1$ à k **faire**Choisir un nombre entier a aléatoire, $1 < a < m$.**si** $a^t \equiv \pm 1$ **alors**Continuer ($k \leftarrow k + 1$).**finsi****pour** $r = 1$ à $s - a$ **faire****si** $a^{2^r t} \bmod m = -1$ **alors**Continuer ($k \leftarrow k + 1$).**finsi****fin pour**Premier \leftarrow FALSE, STOP**fin pour**Retourner Premier.

5.4 Les tests déterministes

Il existe aussi des tests de primalité déterministes, mais moins efficaces ou plus difficiles à mettre en œuvre. Parmi les plus courants, citons celui réalisé par L. Adelman, C. Pomerance et R. Rumely en 1981 [8]. Ce test a depuis connu plusieurs déclinaisons et améliorations.

6 Factorisation

Factoriser un nombre consiste à trouver ses facteurs premiers.

$$\begin{aligned} 10 &= 2 \cdot 5 \\ 3731 &= 7 \cdot 13 \cdot 41 \\ 2^{113} - 1 &= 3391 \cdot 23279 \cdot 65993 \cdot 1868569 \cdot 1066818132868207 \end{aligned}$$

La factorisation est un des plus vieux problèmes de la théorie des nombres. C'est une opération simple, mais longue.

Les algorithmes les plus efficaces sont des algorithmes particuliers, pour lesquels on connaît certaines informations sur le nombre à factoriser.

Les meilleurs algorithmes généraux de factorisation sont dans l'ordre :

- **Number field sieve (NFS)** adapté aux nombres de plus de 130 chiffres ; il a été utilisé pour casser le RSA-155 (voir 2.5).

- **Crible quadratique** le meilleur algorithme pour factoriser des nombres de moins de 130 chiffres ; pour des nombres de 10 à 20 chiffres, l'algorithme de Pollard est plus rapide.
- **Méthode des courbes éллиptiques** cette méthode a permis de factoriser un nombre de 43 chiffres.
- **Algorithme ρ de Pollard [9] [1]** c'est l'algorithme implémenté dans *AdaRSA*. Il est particulièrement efficace jusqu'à 20 chiffres.
- **Algorithme des fractions continues** cet algorithme n'est plus utilisé.
- **Divisions successives** c'est la plus vieille méthode connue ; elle consiste à chercher des facteurs dans le crible d'Ératosthène.

6.1 L'algorithme ρ de Pollard

L'idée développée dans cet algorithme est la suivante : on choisit une fonction polynomiale simple et irréductible, telle que $f(x) = x^2 + 1$. On génère une séquence de nombres depuis un x_0 donné, telle que $x_i = f(x_{i-1}) \pmod n$. Le but est de découvrir des paires de nombres x_i et x_j telles que le diviseur d divise $x_i - x_j$.

Algorithme 7 Algorithme Rho de Pollard

Entrées: Un nombre n .

Sorties: Un facteur de n .

Choisir une valeur initiale x_0 entre 1 et $n - 1$, typiquement la valeur 2.

Choisir une valeur c , définir une fonction polynomiale $f(x) = (x^2 + c) \pmod n$, typiquement, $c = 1$.

tantque on a pas trouvé de facteur **faire**

Calculer $x_i = f(x_{i-1})$.

Calculer $x_{2i} = f(f(x_0))$.

$\text{pgcd}(|x_i - x_{2i}|, n)$ est un facteur de n .

si le facteur est plus grand que 1 et plus petit que n **alors**

on a trouvé un facteur !

finsi

si le facteur est 0 **alors**

L'algorithme a échoué. On peut essayer de recommencer avec une nouvelle valeur de c . Si on retrouve plusieurs fois 0, n est probablement premier.

sinon

Augmenter i de 1.

finsi

fin tantque

Retourner le facteur.

Cet algorithme nécessite très peu de mémoire. Sa complexité est de $O(\sqrt[4]{n})$.

6.1.1 Exemple

Voici ce qui se passe lors de la factorisation de 6944629145383337877043.

À la 1607ème itération, l'algorithme ρ trouve le facteur 845951. Le reste du nombre à factoriser est 8209256972783693.

À la 6994ème itération, l'algorithme ρ trouve le facteur 7601089. Le reste du nombre est 1080010637. Ce nombre est premier.

On connaît maintenant tous les facteurs du nombre :

$$6944629145383337877043 = 845951 \times 7601089 \times 1080010637$$

6.1.2 Application

Dans la pratique, nous avons constaté qu'il est plus rapide de commencer à chercher les petits facteurs du nombre dans un crible d'Ératosthène contenant les nombres premiers inférieurs à 600. C'est seulement après que nous appliquons l'algorithme de Pollard. Nous avons implémenté l'algorithme de manière à ce qu'il s'arrête après un million d'itérations.

7 Déroulement du projet

Le projet s'est déroulé entre les mois d'avril et de juillet 2003. Le reste de nos études à l'EIVD ne nous a malheureusement pas permis d'accorder tout le temps que nous aurions souhaité à *AdaRSA*. Aussi, nous avons écrit l'essentiel du logiciel pendant nos vacances de Pâques et de printemps.

Nous avons vite rencontré le problème de la gestion des sources. Pour y pallier, nous avons mis en place un serveur web et un accès ftp sur lequel nous avons déposé des versions numérotées de nos fichiers. Ce système, s'il a bien fonctionné, a montré ses limites et à l'avenir nous envisageons d'utiliser un système de gestion des sources tel que CVS⁵.

La collaboration a de part et d'autre été tout à fait efficace et le fait d'être deux a permis de maintenir une grande motivation. Nous nous en remercions mutuellement.

8 État du projet

Le projet est terminé. Nous avons implémenté toutes les fonctionnalités qui figurent dans le cahier des charges, à l'exception du générateur aléatoire et de la mesure de son entropie (voir 4 page 18).

⁵<http://www.cvshome.org/>

Le développement de ce projet a nécessité l'implémentation de fonctionnalités liées aux nombres premiers. Ces fonctionnalités (tests de primalité, génération de nombres premiers aléatoires) devaient initialement rester internes au programme. Au cours du développement, il nous est apparu que ces fonctions, une fois présentées et documentées de manière adéquate, pouvaient être d'un grand intérêt pour l'utilisateur. En effet, la sécurité du chiffrement repose sur la qualité des nombres premiers et la difficulté de factoriser le produit de deux de ces nombres. Nous avons donc ajouté des éléments d'interface supplémentaires qui présentent ces fonctionnalités à l'utilisateur.

De même, toujours dans une optique didactique, nous avons implémenté une fonctionnalité qui n'était pas prévue au départ ; la factorisation. Il existe bien sûr des méthodes plus efficaces que la nôtre (voir page 23). Cependant, sa présence permet à l'utilisateur de réaliser à quel point l'effort de calcul nécessaire pour casser les clés RSA est important.

9 Conclusions

Le développement de ce programme s'est révélé passionnant. Entre les quelques articles du magazine *Pour la science* qui ont contribué au choix de ce projet et la version 1.0 de *AdaRSA*, nous avons pu mettre en pratique, approfondir et préciser nos méthodes de programmation en Ada. Nous avons aussi et surtout pu faire l'expérience du développement d'un petit projet à deux. Un tel développement a quelque chose d'exaltant en ce sens qu'il aboutit à un « vrai logiciel ». Il nécessite également une grande rigueur dans la planification, le codage et la gestion des sources qui n'a rien de commun avec les petits programmes que nous écrivons régulièrement au cours de l'année.

Nous savions que la cryptographie était un domaine important ; son étude nous a permis de découvrir à quel point nous avons raison ! De la théorie des nombres à l'algorithmique, nous nous sommes plongés dans des sujets complexes dont nous avons acquis des notions de base et une expérience qui nous seront utiles pour le reste de nos études. L'élaboration d'une documentation importante et structurée telle que celle-ci est également une expérience à part entière. Dans ce cadre là, nous avons pu approfondir notre connaissances du langage \LaTeX .

9.1 Évolutions futures

Standardisation *AdaRSA*, malgré le fait qu'il fonctionne normalement, reste un exercice pédagogique. S'il veut un jour quitter le stade de prototype, il lui faudra être compatible avec les autres logiciels du marché. En particulier, *AdaRSA* devrait être capable de communiquer avec d'autres logiciels par le biais de formats de fichiers standardisés. Nous avons déjà identifié quelques RFC⁶ documentant ces standards : il s'agit des RFC 2313 et 2437 et 3447. La standardisation est un sujet important qui mériterait une étude rigoureuse.

⁶<http://www.ietf.org/rfc.html>

Performances L'autre domaine à améliorer prioritairement est la performance des algorithmes. Nous pensons avoir réalisé de bonnes implémentations mais sommes bien conscients que nous pourrions gagner beaucoup de temps à l'exécution. Nous avons utilisé des outils de *profiling* pour déterminer les fonctions à améliorer en priorité. Sans surprise, les procédures dont l'amélioration aurait l'incidence la plus grande sur les performances du programme sont la division et l'opérateur modulo, principalement en raison de leur omniprésence dans les calculs.

Sécurité Il faudrait aussi envisager l'insertion de bits aléatoires dans les messages chiffrés, pour parer au fait qu'un attaquant pouvant présumer du contenu du message peut déduire le message en chiffrant lui-même les contenus possibles.

Portabilité De par le fait qu'il ait été développé exclusivement à l'aide de logiciels libres et de par sa structure modulaire, *AdaRSA* peut être facilement porté sur une autre plateforme que Microsoft Windows ; seul le programme principal devra être réécrit.

En conclusion, nous sommes heureux d'avoir pu relever ce défi, tout en restant conscients de l'immensité du travail restant à accomplir pour faire de *AdaRSA* un logiciel exploitable en production.

9.2 Signatures des auteurs

Daniel Lifschitz

Nicolas Seriot

Références

- [1] Donald Ervin Knuth. *The Art of Computer Programming, 2nd edition*. Addison-Wesley, 1981.
- [2] Frédéric Rémi. La cryptographie à clé publique. *Pour la science*. pp. 44–51, juillet-août 2002.
- [3] Louis Goubin. Stratégies d'attaques. *Pour la science*. pp. 58–61, juillet-août 2002.
- [4] David Poincheval. Prouver la sécurité. *Pour la science*. pp. 52–53, juillet-août 2002.
- [5] Bruce Schneier. *Applied Cryptography, Second Edition : Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., 1996. ISBN : 0471128457.
- [6] RSA Security. <http://www.rsasecurity.com/>

- [7] Jean-François Hêche. Cours de mathématiques discrètes, EIVD, 2002–2003.
- [8] Léonard M. Adelman, Carl Pomerance, and Robert S. Rumely. On distinguishing prime numbers from composite numbers. *Annals of Mathematics*. pp. 173–206, January 1987.
- [9] John M. Pollard. A Monte Carlo Method for Factorization. *BIT*. v. 15, pp. 331–334, 1975.

Deuxième partie

Documentation utilisateur

10 Informations générales concernant le logiciel

10.1 But et généralités

Ce logiciel a pour but de se familiariser avec les principes de la cryptographie asymétrique, plus communément appelée cryptographie à clé publique.

La cryptographie à clé publique utilise un système de chiffrement où la clé utilisée pour le chiffrement diffère de celle utilisée pour le déchiffrement : pour chiffrer un message, on utilise la clé publique du destinataire (qui, comme l'indique son nom, peut être connue de tous et n'a donc pas de caractère secret) ; pour déchiffrer le message qu'il a reçu, le destinataire utilise une autre clé, sa clé privée, qu'il est seul à connaître. Parmi les nombreux systèmes asymétriques qui sont proposés, RSA est l'un des plus répandus.

La sécurité du système réside dans le fait qu'il existe des fonctions mathématiques relativement facile à calculer, mais dont l'inverse ne l'est justement pas. Prenons le nombre 13717421 et demandons-nous quels sont les deux nombres premiers⁷ dont ce nombre est le produit. La réponse est 3607 et 3803. En effet, une simple multiplication, même mentale, permet de le vérifier. Comme on peut le constater, si multiplier ces deux entiers est une opération des plus élémentaires, effectuer l'inverse demande de nombreux essais avant de trouver la solution. L'opération qui consiste à retrouver les facteurs premiers d'un nombre s'appelle la factorisation.

Le dernier record de factorisation d'un entier a été établi durant l'été 1999 par une équipe internationale de mathématiciens qui a factorisé un entier de 155 chiffres (512 bits). Cette factorisation a été obtenue après cinq mois de calculs effectués sur 285 ordinateurs et un super-calculateur. En supposant que la puissance des machines suive la loi empirique de Moore (la puissance de calcul double tous les 18 mois), on estime qu'une clé de 1024 bits (309 chiffres) sera cassée en 2030, une clé de 2048 bits (617 chiffres) en 2080. À moins, bien sûr, qu'une percée théorique ou technique insoupçonnée ne fasse s'écrouler ces belles suppositions.

Une clé RSA, qu'elle soit privée ou publique, est composée de nombres dont la taille peut être de plusieurs centaines de chiffres. Plus la taille de la clé est grande, plus le cryptage est sûr. En contrepartie, les opérations de chiffrage et de déchiffrage prennent beaucoup plus de temps. Les tailles les plus utilisées aujourd'hui se situent entre 512 et 1024 bits. À titre d'exemple, les cartes bancaires à puce renferment des clés de 768 bits et l'échange de clés privées sur internet pour des transactions bancaires se fait en général à l'aide de clés publiques de 1024 bits. Le logiciel *AdaRSA* permet de générer des clés de 128 à 2048 bits, ce qui correspond à un nombre d'environ 617 chiffres.

⁷nombres divisible **uniquement** par 1 et par eux-même (2, 3, 5, 7, 11, ...)

Les nombres premiers sont essentiels dans la génération de clés asymétriques. *AdaRSA* permet de générer de tels nombres et offre plusieurs outils permettant d'effectuer des tests de primalité. En outre, une fonction de factorisation permet de décomposer un nombre donné en facteurs premiers.

10.2 Configuration minimale recommandée

- processeur Intel Pentium II à 500Mhz ;
- Microsoft Windows 2000 ou supérieur ;
- écran de résolution minimale 1024*768 ;
- un navigateur web pour la lecture des fichiers d'aide ;
- Adobe Acrobat Reader⁸ pour la lecture du guide de l'utilisateur.

10.3 Lieu d'existence

Le programme d'installation est disponible sur le CD fourni avec le document.

10.4 Licence

AdaRSA est libre de droits. Son code source est disponible sur simple demande (section 16 page 41).

11 Installation

Vous devez être en possession des droits « administrateurs » ou « utilisateurs avec pouvoir » pour installer *AdaRSA*⁹. Si vous n'êtes pas sûr des droits qui vous sont accordés, veuillez consulter votre administrateur système.

Pour lancer l'installation, effectuez un double-clic sur le fichier **Setup.exe**. Un assistant vous guidera pas à pas.

11.1 Désinstallation

Il y a deux façons de désinstaller *AdaRSA* :

- Aller dans le menu *Démarrer / Programmes / AdaRSA* et lancer le programme **Uninstall**.
- Aller dans le menu *Démarrer / Paramètres / Panneau de configuration / Ajout / Suppression de programmes*, sélectionner la ligne *AdaRSA* et cliquer sur *Supprimer*.

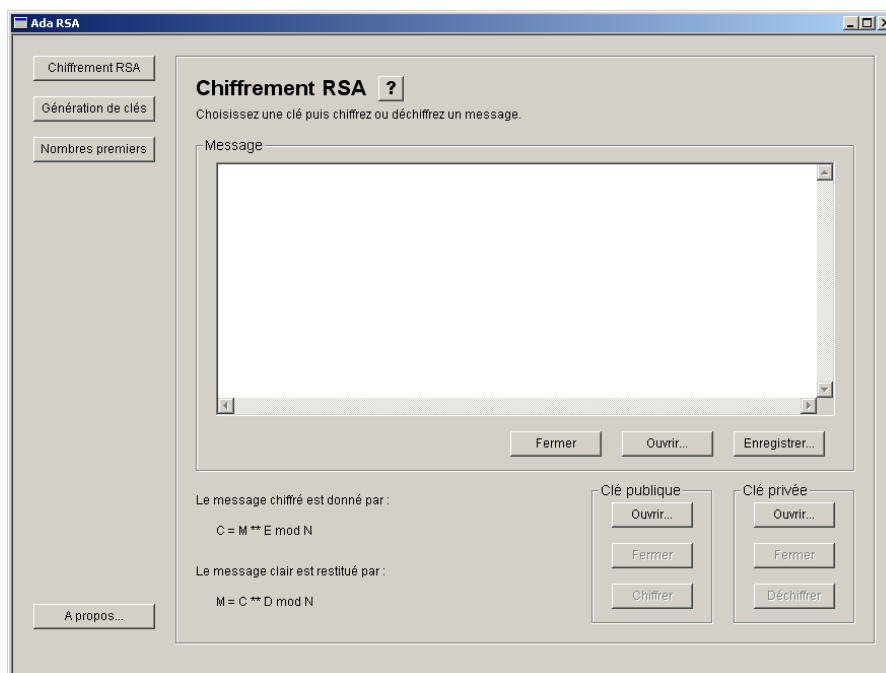
12 Mode d'emploi

12.1 Démarrage du logiciel

Pour lancer le programme, allez dans *Démarrer / Programmes / AdaRSA* puis cliquez sur **AdaRSA**.

⁸<http://www.adobe.fr/products/acrobat/>

⁹Recherchez « groupe d'utilisateurs » dans l'aide de Windows pour plus de précisions.

FIG. 1 – Fenêtre *AdaRSA* Chiffrement RSA

12.2 Fenêtre principale

La fenêtre principale de *AdaRSA* (figure 1) est découpée en deux parties. À gauche vous trouvez les boutons permettant au choix de :

- Chiffrer et déchiffrer des messages textes à l’aide de clés RSA (bouton *Chiffrement RSA*).
- Générer des couples de clés RSA (bouton *Génération de clés*).
- Tester la primalité de nombres entiers (bouton *Nombres premiers*).
- Identifier la version du logiciel (bouton *A propos*).

Le contenu de la partie droite de la fenêtre s’adapte au choix effectué. N’hésitez pas à consulter l’aide ! Celle-ci est accessible via le bouton ? qui se trouve à droite du titre de chacune des fenêtres.

12.3 Chiffrement RSA

La fenêtre *Chiffrement RSA* (figure 1) est réservée aux opérations de chiffrement et de déchiffrement de messages textes. Avant de pouvoir effectuer une de ces opérations, vous devez être en possession d’un fichier contenant soit une clé publique RSA, soit une clé privée RSA. Si vous n’en êtes pas encore pourvu, veuillez vous référer à la section 12.4 en page 33.

12.3.1 Le cadre *Message*

Le cadre *Message* est destiné à contenir le texte à chiffrer ou à déchiffrer. Vous pouvez soit saisir directement le texte dans le champ prévu à cet effet, soit choisir un fichier sur le disque en utilisant le bouton *Ouvrir...*

Le bouton *Ouvrir...* situé juste en dessous du champ *Message* vous permet de choisir un fichier sauvegardé sur le disque dur de votre ordinateur ou sur un support externe.

Le bouton *Enregistrer...* situé à droite du bouton *Ouvrir...* vous permet de sauvegarder le contenu du champ *Message* dans un fichier texte.

Le bouton *Fermer* permet de vider de son contenu le champ *Message*.

12.3.2 Le cadre *Clé publique*

Le cadre *Clé publique* est destiné aux opérations de chiffrement de messages.

Le bouton *Ouvrir...* permet de choisir la clé publique. Les clés publiques de *AdaRSA* comportent l'extension *.pub*. Une boîte de dialogue vous guidera lors l'ouverture du fichier. Les boutons *Fermer* et *Chiffrer* du cadre *Clé publique* deviennent disponibles après le chargement de la clé.

Le bouton *Fermer* permet de supprimer la clé publique de la mémoire. Ce bouton a pour effet de rendre inaccessibles les boutons *Fermer* et *Chiffrer* du cadre *Clé publique*.

Ce bouton est grisé, et donc inaccessible, si aucune clé publique n'est chargée en mémoire.

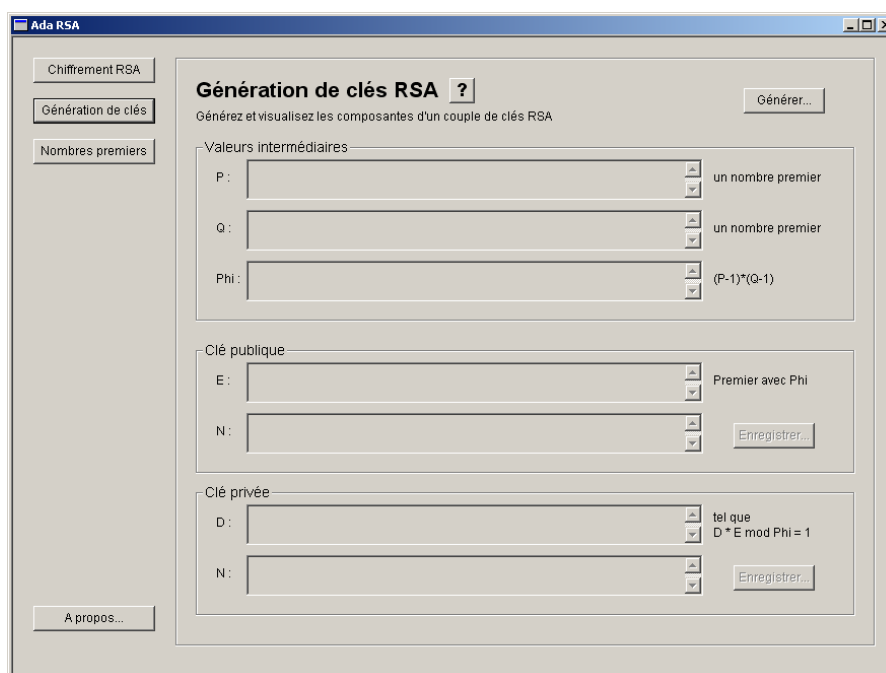
Le bouton *Chiffrer* lance l'opération de chiffrement du texte situé dans le champ *Message*. Après un court instant, le texte chiffré s'affiche en lieu et place du message d'origine.

Ce bouton est grisé, et donc inaccessible, si aucune clé publique n'est chargée en mémoire.

12.3.3 Le cadre *Clé privée*

Le cadre *Clé privée* est destiné aux opérations de déchiffrement de messages.

Le bouton *Ouvrir...* permet de choisir la clé privée. Les clés privées de *AdaRSA* comportent l'extension *.pri*. Une boîte de dialogue vous guidera lors l'ouverture du fichier. Les boutons *Fermer* et *Chiffrer* du cadre *Clé privée* deviennent disponibles après le chargement de la clé.

FIG. 2 – Fenêtre *AdaRSA* Génération de clés RSA

Le bouton *Fermer* permet de supprimer la clé privée de la mémoire. Ce bouton a pour effet de rendre inaccessible les boutons *Fermer* et *Chiffrer* du cadre *Clé privée*.

Ce bouton est grisé, et donc inaccessible, si aucune clé privée n'est chargée en mémoire.

Le bouton *Déchiffrer* lance l'opération de déchiffrement du texte situé dans le champ *Message*. Après un moment, qui peut durer quelque minutes si la clé utilisée est grande et si le message est long, le texte clair s'affiche en lieu et place du message chiffré.

Ce bouton est grisé, et donc inaccessible, si aucune clé privée n'est chargée en mémoire.

12.4 Génération de clés RSA

La fenêtre *Génération de clés RSA* (figure 2) est réservée aux opérations permettant la fabrication de clés RSA. De plus, les étapes intermédiaires menant à la créations des clés sont indiquées à l'écran, ceci dans un but pédagogique.

Le bouton *Générer...* ouvre une nouvelle boîte de dialogue (figure 3) permettant de choisir la taille de la clé à générer. Les choix possibles sont :

- 128 bits (environ 39 chiffres)
- 256 bits (environ 78 chiffres)
- 512 bits (environ 155 chiffres)
- 1024 bits (environ 309 chiffres)

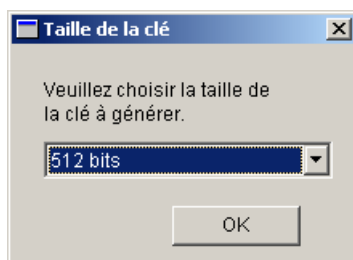


FIG. 3 – Taille de la clé

- 2048 bits (environ 617 chiffres)

Une fois votre choix effectué, cliquez sur le bouton *OK*. Après un instant qui peut durer quelques minutes si vous avez choisi des clés de grande taille, *AdaRSA* affiche les nouvelles clés publiques et privées ainsi que les valeurs intermédiaires utilisées lors des calculs. Ces valeurs permettent de comprendre comment fonctionne la génération des clés.

12.4.1 Le cadre *Valeurs intermédiaires*

Cette partie contient trois valeurs initiales. Ces valeurs sont à la base du calcul des clés RSA.

Les valeurs P et Q sont deux nombres premiers aléatoires de sorte que $P \cdot Q$ soit de la taille choisie pour les clés RSA.

La valeur ϕ , notée *Phi* dans la fenêtre de *AdaRSA*, est le produit mathématique de $(P - 1) \cdot (Q - 1)$.

Ces valeurs, au même titre que la clé privée, ne doivent jamais être divulguées. Elles permettent de reconstituer la clé privée et de déchiffrer les messages.

12.4.2 Le cadre *Clé publique*

Il s'agit de la clé que vous pouvez librement distribuer. C'est elle qui permet de chiffrer les messages qui vous seront adressés. Seul le détenteur de la clé privée associée peut déchiffrer les messages ainsi chiffrés.

La valeur E représente l'exposant public. E doit être premier avec Φ . On pourrait déterminer E aléatoirement, mais on gagne du temps en choisissant un nombre premier arbitrairement. Ici, on utilise le nombre 65537, par convention et parce que cette valeur, du fait de sa représentation binaire qui ne comprend que deux 1, accélère les calculs.

La valeur N représente le module RSA. Elle est le résultat du produit mathématique de $P \cdot Q$.

Le bouton *Enregistrer...* situé à droite de la valeur N permet de sauvegarder la clé publique dans un fichier. Ce bouton est grisé si aucune clé n'est affichée dans le cadre.

12.4.3 Le cadre *Clé privée*

La clé privée, aussi appelée clé secrète, ne doit jamais être divulguée au risque de rendre tous les messages chiffrés à l'aide de la clé publique associée immédiatement déchiffrables.

La valeur D représente l'exposant privé. Sa valeur doit être telle que

$$E \cdot D \pmod{\Phi} = 1$$

La valeur N représente le module RSA. Elle est identique à la valeur N de la clé publique.

Le bouton *Enregistrer...* situé à droite de la valeur N permet de sauvegarder la clé privée dans un fichier. Ce bouton est grisé si aucune clé n'est affichée dans le cadre.

12.5 Nombres premiers

La fenêtre *Nombres premiers* (figure 4) permet de faire passer une batterie de tests de primalité à des grands nombres entiers. Pour rappel, un nombre premier est un nombre divisible uniquement par 1 et par lui-même. Par convention, 1 n'est pas considéré comme premier et les nombres négatifs non plus.

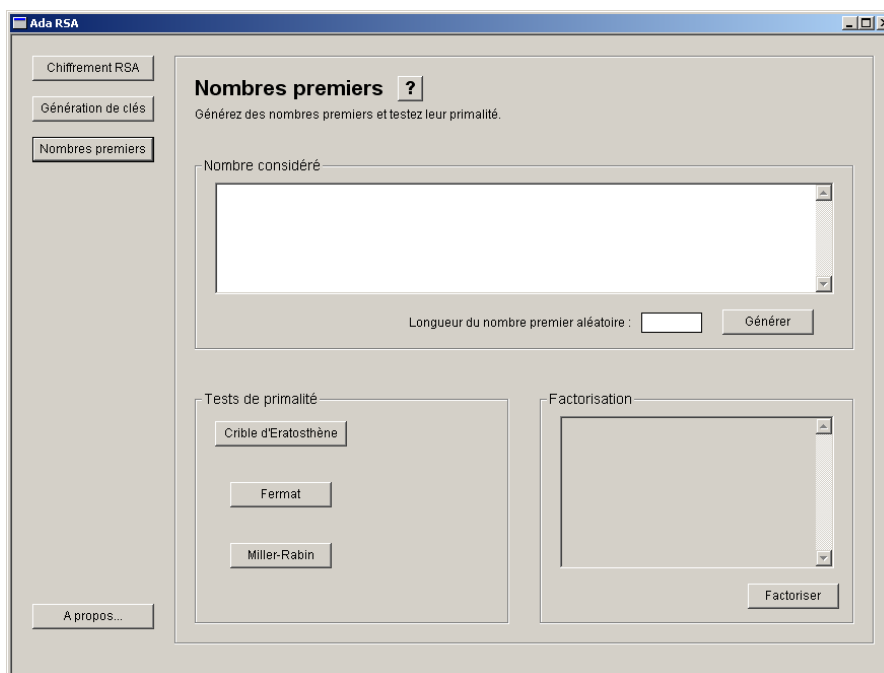
12.5.1 Le cadre *Nombre considéré*

Il s'agit ici du nombre qui sera utilisé pour les tests. Vous pouvez saisir un nombre dans le champ prévu à cette effet. Pour une meilleure lisibilité, il est possible de saisir le nombre en séparant les chiffres par des espaces ou des caractères de soulignement (par exemple 1_000_000).

Vous pouvez aussi générer un nombre déjà premier en indiquant sa taille en nombre de chiffres dans le champ *Longueur du nombre premier aléatoire* puis en cliquant sur le bouton *Générer*.

12.5.2 Le cadre *Tests de primalité*

Le bouton *Crible d'Ératosthène* permet de rechercher un diviseur parmi les nombres premiers inférieurs à 600. Si on trouve un diviseur, on est certain que le nombre n'est pas premier. Dans le cas contraire, il faut distinguer le cas où le nombre à tester est inférieur ou supérieur à 600^2 , soit 360'000. Dans le premier cas, on est certain que le nombre testé est premier. Dans l'autre, il existe une probabilité que le nombre soit premier, mais elle est inversement proportionnelle à la taille du nombre testé.

FIG. 4 – Fenêtre *AdaRSA* Nombres premiers

Le bouton *Fermat* lance un test qui utilise le petit théorème de Fermat. On choisit au hasard un entier a tel que $2 \leq a \leq n - 1$ et on calcule $a^{n-1} \bmod n$. Si on obtient autre chose que 1, c'est que n est composé.

La probabilité que le test n'identifie pas un nombre composé est proche de $\frac{1}{2}$. En répétant le test R fois avec à chaque fois un a différent, la probabilité qu'un nombre composé ne soit pas identifié est d'environ $(\frac{1}{2})^R$. Dans *AdaRSA*, le test de Fermat est répété 20 fois. Un nombre qui passe les 20 tests sans être déclaré composée a donc près de $1 - (\frac{1}{2})^{20}$ chances d'être premier, soit 99.9999%.

Cette probabilité ne peut toutefois pas être bornée, du fait de l'existence de nombres particuliers dont le test de Fermat ne peut déterminer le caractère composé : on les appelle les nombres de Carmichael. Ces nombres ont la propriété que pour tout $a \in \{2, \dots, n-1\}$, on a $a^{n-1} \bmod n = 1$, sauf si $\text{pgcd}(a, n) \neq 1$. Les nombres de Carmichael sont très rares : il n'y en a que 255 en dessous de 100'000'000, et 2'163 en dessous de 25'000'000'000. Les nombres de Carmichael inférieurs à 100'000 sont : 561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841, 29341, 41041, 46657, 52633, 62745, 63973 et 75361.

Le bouton *Miller-Rabin* permet de lancer un autre test probabiliste très efficace. Il a été présenté par Michael Rabin, qui s'est appuyé sur les travaux de Gary Miller. Il a de nombreux avantages : il n'existe pas de nombres dont il ne puisse détecter le caractère composé comme c'est le cas pour des nombres de Carmichael pour le test de Fermat. Il

est aussi plus rapide et plus précis.

L'idée est la suivante. Soit m un nombre composé impair. On peut écrire $m - 1 = 2^s t$ avec t impair. Soit encore $b \in \{1, 2, \dots, m - 1\}$. Alors soit $b^t \pmod m = 1$, ou il existe $r \in \{0, \dots, s - 1\}$ tel que $b^{2^r t} \pmod m = -1$.

La probabilité que le test de Miller-Rabin détecte un nombre composé est de $\frac{1}{4}$. Dans *AdaRSA*, le test de Miller-Rabin est exécuté 10 fois successivement. Ainsi, la probabilité qu'un nombre ayant passé les 10 tests soit premier est de $1 - \left(\frac{1}{4}\right)^{10}$, soit 99.9999%.

12.5.3 Le cadre *Factorisation*

Le bouton **Factoriser** lance un algorithme permettant de trouver tous les nombres premiers composant le *Nombre considéré*.

Si la factorisation est possible dans un laps de temps raisonnable, le résultat s'affiche dans le champ multiligne situé dans le cadre *Factorisation*. Dans le cas contraire, un message s'affiche et indique que la factorisation ne peut se faire dans un temps raisonnable, c'est-à-dire de l'ordre d'une dizaine de minutes.

12.6 Quitter *AdaRSA*

Pour quitter l'application, cliquez sur la croix située en haut à droite de la fenêtre.

12.7 Messages d'erreurs

Impossible de lire le fichier choisi. Celui-ci est peut-être en cours d'utilisation ou corrompu. Ce message est associé aux trois boutons *Ouvrir...* de la fenêtre *Chiffrement RSA*. Il indique que le fichier choisi ne peut pas être lu correctement. Pour corriger cette erreur, contrôlez que le fichier n'est pas déjà en cours d'utilisation dans une autre application ou qu'il n'est pas en lecture seule. Enfin, utilisez un logiciel de détection et réparation des erreurs disque¹⁰.

Impossible d'écrire dans le fichier. Celui-ci est peut-être en cours d'utilisation, en lecture seule ou corrompu. Ce message est associé aux boutons *Enregistrer...* des fenêtres *Chiffrement RSA* et *Génération de clés RSA*. Il indique que la sauvegarde du fichier désiré n'est pas possible. Pour corriger cette erreur, contrôlez que le fichier n'est pas déjà en cours d'utilisation dans une autre application ou qu'il n'est pas en lecture seule. Enfin, utilisez un logiciel de détection et réparation des erreurs disque¹⁰.

La clé de chiffrement est corrompue. Ce message est associé au bouton *Chiffrer* de la fenêtre *Chiffrement RSA*. Il indique que la clé publique que vous avez précédemment chargée est corrompue. Pour remédier à ce problème, il est nécessaire d'utiliser une autre clé.

¹⁰Recherchez « réparer des erreurs disque » dans l'aide de Windows pour plus de précisions.

Ce texte n'est pas un message chiffré. Ce message est associé au bouton *Déchiffrer* de la fenêtre *Chiffrement RSA*. Il indique que le message que vous tentez de déchiffrer n'est pas un message chiffré, ou alors que ce dernier est corrompu. Un message chiffré est facilement identifiable au fait qu'il ne comporte que des chiffres.

Le fichier choisi ne contient pas une clé valide. Ce message est associé aux boutons *Ouvrir...* de la fenêtre *Chiffrement RSA*. Il indique que la clé contenue dans le fichier choisi n'est pas valide ou a été altérée. Pour remédier à ce problème, il est nécessaire d'utiliser une autre clé.

Veillez saisir un entier entre 1 et 616. Ce message d'erreur est associé au bouton *Générer* de la fenêtre *Nombres premiers*. Il indique que la valeur que vous avez saisie dans le champ *Longueur du nombre premier aléatoire* n'est pas comprise entre ces deux bornes.

Le nombre à considérer n'est pas valable. Ce message est associé aux boutons *Crible d'Erastosthène*, *Fermat*, *Miller-Rabin* et *Factoriser* de la fenêtre *Nombres premiers*. Il indique que le nombre saisi dans le champ *Nombre considéré* n'est pas valable. Ce champ n'admet que les chiffres de 0 à 9, les espaces et les caractères de soulignement (`_`).

Le nombre considéré ne doit pas avoir plus de 616 chiffres. Ce message est associé aux boutons *Crible d'Erastosthène*, *Fermat*, *Miller-Rabin* et *Factoriser* de la fenêtre *Nombres premiers*. Il indique que le nombre saisi dans le champ *Nombre considéré* est trop grand. Ne saisissez pas de nombre de plus de 616 chiffres.

La factorisation est possible uniquement avec des valeurs positives. Veuillez corriger la saisie SVP. Ce message est associé au bouton *Factoriser* de la fenêtre *Nombres premiers*. Il est tout simplement impossible de factoriser des nombres négatifs ainsi que le nombre zéro.

Il est impossible de factoriser ce nombre dans un temps raisonnable. L'opération a été interrompue. Ce message est associé au bouton *Factoriser* de la fenêtre *Nombres premiers*. Il indique que le temps nécessaire à la factorisation du nombre demandé prendrait beaucoup trop de temps. Pour rappel, la difficulté de casser une clé RSA est précisément due au fait qu'il est très long et difficile de factoriser un grand entier comportant deux grands facteurs premiers.

Le fichier 'Aide_Nom_Fichier.html' (ou un de ses composants) est introuvable. Vérifiez que le chemin et le nom de fichier sont corrects, et que toutes les bibliothèques requises sont disponibles. Ce message indique que le fichier d'aide demandé ne peut pas être ouvert. Vérifiez que celui-ci se trouve bien dans le répertoire d'installation de *AdaRSA*. Si tel est le cas, vérifiez les paramètres de votre navigateur internet afin que celui-ci ouvre automatiquement les fichiers portant l'extension `.html`.

13 Exemples d'utilisation de *AdaRSA*

13.1 Chiffrement et déchiffrement d'un message

Commencez par sélectionner la fenêtre **Chiffrement RSA** en cliquant sur le bouton du même nom situé à gauche de la fenêtre. Vous devriez obtenir la fenêtre de la figure 1 en page 31.

13.1.1 Ouverture d'un fichier message

Cliquez maintenant sur le bouton *Ouvrir...* situé sous le cadre *Message*. Une boîte de dialogue permettant de sélectionner un fichier s'ouvre.

Effectuez un double-clic sur le répertoire **Exemples** qui se trouve dans le répertoire d'installation de *AdaRSA* (C:\Program Files\AdaRSA si vous avez choisi les paramètres d'installation par défaut). Une liste de fichiers s'affiche dans la fenêtre. Sélectionnez le fichier *La cigale et la fourmi.txt* puis cliquez sur *Ouvrir*. Le poème de Jean de la Fontaine s'affiche dans la partie *Message*.

13.1.2 Chiffrer le message

Dans le cadre *Clé publique*, cliquez sur le bouton *Ouvrir...* Ceci ouvre à nouveau une boîte de dialogue de sélection de fichiers. Choisissez le fichier *Clé_128.pub* puis *Ouvrir*. Le fichier se charge en mémoire, la boîte de dialogue se ferme et le bouton *Chiffrer* devient disponible ; l'opération de chiffrement peut maintenant commencer. Cliquez sur le bouton *Chiffrer* ; après un court instant, le poème est remplacé par des chiffres ; le message est chiffré.

13.1.3 Déchiffrer le message

Dans le cadre *Clé privée*, cliquez sur le bouton *Ouvrir...* Ceci ouvre à nouveau une boîte de dialogue de sélection de fichiers. Choisissez le fichier *Clé_128.pri* puis *Ouvrir*. Le fichier se charge en mémoire, la boîte de dialogue se ferme et le bouton *Déchiffrer* devient disponible ; l'opération de déchiffrement peut maintenant commencer. Cliquez sur le bouton *Déchiffrer* ; après un instant, les chiffres laissent à nouveau place au poème ; le message est déchiffré.

13.2 Génération d'un couple de clés RSA

Sélectionnez d'abord la fenêtre **Génération de clés** en cliquant sur le bouton du même nom situé à gauche de la fenêtre. Vous devriez obtenir la fenêtre de la figure 2 en page 33.

Cliquez sur le bouton *Générer...* situé en haut à droite de la fenêtre. Dans la nouvelle fenêtre qui s'ouvre, choisissez une taille de 512 bits, puis *OK*. Après quelques secondes, tous les champs de la fenêtre se remplissent de chiffres : vous venez de générer un couple de clés RSA !

Pour sauvegarder les clés générées, cliquez sur le bouton *Enregistrer...* situé dans le cadre *Clé publique* de la fenêtre. Une boîte de dialogue vous permettant de choisir l'emplacement de sauvegarde et de saisir un nom de fichier s'ouvre. Une fois les champs remplis, cliquez sur *Enregistrer*.

Répétez la même opération pour la clé privée.

13.3 Tests de primalité et factorisation

Sélectionnez d'abord la fenêtre **Nombres premiers** en cliquant sur le bouton du même nom situé à gauche de la fenêtre. Vous devriez obtenir la fenêtre de la figure 4 en page 36.

Saisissez le nombre 52633 dans le champ *Nombre considéré*, puis cliquez successivement sur les boutons *Crible d'Ératosthène*, *Fermat* et *Miller-Rabin*. Le résultat des tests s'affiche en regard de chacun des boutons. Comme on peut le constater, le nombre saisi a la particularité de ne pas passer correctement le test de Fermat. Il s'agit d'un nombre de Carmichael. Pour le vérifier, cliquez sur le bouton *Factoriser*. Les facteurs premiers de 52633 s'affichent dans le champ texte *factorisation*.

Saisissez maintenant la valeur 20 dans le champ *Longueur du nombre premier aléatoire*, puis cliquez sur le bouton *Générer*. Un nombre de 20 chiffres s'affiche juste au dessus. Répétez les tests de primalité et lancez la factorisation ; tout indique que ce nombre est bien premier.

14 Mise en garde

La fonction de chiffrement utilisée pour la cryptographie à clé publique est une fonction déterministe. Cela veut dire que si l'on sait que le message chiffré correspond à un message clair qui ne peut être que « oui » ou « non » par exemple, il suffit de tenter de chiffrer soi-même ces deux possibilités et de comparer le résultat avec le chiffré transmis, qui est soit celui de « oui », soit celui de « non ».

Dans la pratique, on pallie à cette faiblesse en insérant des caractères aléatoires devant le message à chiffrer afin d'obtenir à tous les coups un message d'apparence différente. *AdaRSA* n'utilise pas cette méthode ; il n'est donc sûr que si l'attaquant potentiel ne peut présumer du contenu exact du message.

15 Problèmes connus

Les seuls problèmes connus actuellement concernent le champ *Message* de la fenêtre *Chiffrement RSA* :

- la taille maximale du texte pouvant prendre place dans ce champ est limitée à 30 Ko ;
- il n'est pas tenu compte des lignes ne comportant aucun caractère autre que le retour à la ligne. Pour conserver un saut de ligne, ajoutez une espace ;

- la fermeture de la fenêtre graphique durant un calcul n’interrompt pas l’application, celle-ci termine son exécution en tâche de fond. Les résultats éventuels ne sont pas affichés.

16 Coordonnées des auteurs

Vous pouvez contacter les développeurs par courrier électronique aux adresses suivantes :

Nicolas Seriot nicolas@seriot.ch

Daniel Lifschitz daniel@daniel.li

Troisième partie

Documentation technique

17 Généralités

17.1 Langage de programmation

AdaRSA a été entièrement écrit en ADA 95 en utilisant uniquement les notions apprises en première année à l'EIVD, à savoir la programmation séquentielle.

17.2 Librairies externes

Mise à part les libraires standards livrées avec le compilateur GNAT, nous avons utilisé les services de la librairie JEWL. Celle-ci permet la création et la manipulation de fenêtres graphiques sous Windows 95 et supérieur et ceci de manière relativement simple. JEWL peut être téléchargé à l'adresse <http://www.it.bton.ac.uk/staff/je/jewl/>.

17.3 Compilation

Nous avons utilisé la version 3.15p du compilateur Ada GNAT. À noter que l'utilisation de la librairie JEWL nécessite l'installation préalable de GnatWin¹¹ qui fournit les librairies d'interfaces avec Windows.

En supposant que la librairie Jewl soit installée en `c:\jewl` la ligne de commande pour compiler *AdaRSA* est :

```
gnatmake adarsa -Ic:\jew\source -larges -mwindows -s
```

Le paramètre `-mwindows` empêche l'ouverture d'une console Windows lors de l'exécution du programme et le paramètre `-s` supprime les symboles additionnels utilisés pour le déverminage. Ceci a pour effet de rendre le fichier exécutable plus compact.

17.4 Programme d'installation

Le programme d'installation de *AdaRSA* a été créé à l'aide du logiciel gratuit NSIS de Nullsoft Inc.¹² Celui-ci permet à l'aide d'un script de générer un exécutable d'installation pour Windows.

Le script `AdaRSA.nsi` utilisé pour la génération du programme `setup.exe` de *AdaRSA* est inclus dans le CD de développement.

18 Découpe du programme

AdaRSA est composé d'un programme principal et de plusieurs paquetages, chacun ayant son propre rôle dans l'élaboration de l'application. Nous présentons ici une courte description de chacun d'entre eux.

¹¹GnatWin peut être téléchargé à l'adresse <http://libre.act-europe.fr/GNAT/main.html>

¹²<http://nsis.sourceforge.net/>

Grands_Entiers_G Ce paquetage générique fournit un type `T_Long_Entier` permettant de représenter et de manipuler de très grands nombres entiers.

Grands_Entiers_G.Aleatoire_G Il s'agit d'une unité enfant du paquetage précité. Celui-ci permet de générer des nombres aléatoires pour le type `T_Long_Entier`.

Grands_Entiers Il s'agit d'une unité de bibliothèque qui instancie le paquetage générique **Grands_Entiers_G** avec une valeur fixant la contrainte du type `T_Long_Entier` à 2048 bits.

Nombres_Premiers Ce paquetage fournit des opérations permettant aussi bien de générer de nombres premiers aléatoires ainsi que des tests permettant de le vérifier. En outre, il met à disposition une procédure de factorisation.

RSA Ce paquetage met à disposition tous les outils nécessaires à la création de clés RSA et au chiffrement et déchiffrement.

Conversion_Textes_Nombres Ce paquetage fournit les outils nécessaires à la conversion de chaînes de caractères en nombres et vice-versa.

Queue_G Ce paquetage générique met à disposition des fonctions permettant de stocker des informations dans une queue dynamique.

AdaRSA Il s'agit du programme principal de notre application. Celui-ci construit l'interface graphique et réagit aux actions exécutées par l'utilisateur.

19 Description des fichiers

19.1 Programme principal

Le programme principal remplit les trois fonctions principales suivantes :

1. définition et construction de l'interface graphique ;
2. réponses aux sollicitations de l'utilisateur ;
3. appels aux fonctions et procédures des paquetages et affichage des résultats.

19.1.1 Interface graphique

Chaque objet de la fenêtre graphique (cadres, boutons, étiquettes, champs de saisie. . .) est au préalable construit dans la partie déclarative du programme. Par la suite, dans la partie corps du programme, ces objets sont rendus visibles, invisibles ou grisés en fonction des actions entreprises par l'utilisateur et des résultats des fonctions et procédures appelées.

Dans une interface Windows standard, le pointeur de la souris est modifié en sablier pour indiquer à l'utilisateur que le système est occupé. `Jewl` n'offrant pas cette fonctionnalité, nous avons mis en place un mécanisme permettant de griser tous les boutons

durant les phases de calculs, puis de les réactiver en fonction de leur état précédent et du résultat des calculs.

Pour réaliser cette opération, nous avons déclaré les types suivants :

```
type T_Evenement is (Aucun,
                       Clé_Pub_Oui,
                       Clé_Pub_Non,
                       Clé_Pri_Oui,
                       Clé_Pri_Non,
                       Génération);
```

```
type T_Etat_Bouton is
  record
    Bouton      : T_Ptr_Bouton;
    Actif       : Boolean;
    Activer     : T_Evenement;
    Désactiver  : T_Evenement;
  end record;
```

```
type T_Etat_Boutons is array (Positive range <>) of T_Etat_Bouton;
```

Le type T_Evenement énumère les événements susceptibles de modifier l'état des boutons :

- Aucun : indique que l'événement en cours n'implique aucun changement dans l'aspect des boutons;
- Clé_Pub_Oui : cet événement a lieu lorsqu'une clé publique est chargée en mémoire;
- Clé_Pub_Non : cet événement à lieu lorsque la clé publique n'est plus en mémoire;
- Clé_Pri_Oui : idem à Clé_Pub_Oui mais concerne la clé privée;
- Clé_Pri_Non : idem à Clé_Pub_Non mais concerne la clé privée;
- Génération : cet événement a lieu lors de la génération d'un couple de clés RSA.

Le type T_Etat_Bouton permet d'associer à un bouton les événements susceptibles de l'activer ou de le désactiver ainsi que son état actuel.

Quant au type T_Etat_Boutons il permet de déclarer un tableau pouvant contenir l'état de tous les boutons de l'application.

Dans la partie déclarative, une variable Etat_Boutons est déclarée et contient l'état initial des boutons. Cet objet est ensuite transmis au fur et à mesure des actions aux trois procédures suivantes :

```
procedure Activer_Boutons (Liste : in T_Etat_Boutons);

procedure Désactiver_Boutons (Liste : in T_Etat_Boutons);

procedure Changer_Boutons (Liste      : in out T_Etat_Boutons;
                          Evénement : in      T_Evenement);
```

19.1.2 Réponses aux sollicitations de l'utilisateur

Le corps du programme principal est constitué d'une grande boucle `loop` dans laquelle une instruction `case` permet d'effectuer les branchements nécessaires en fonction du bouton sélectionné par l'utilisateur.

19.1.3 Procédures du programme principal

Chacune des branches de l'instruction `case` fait appel à une des procédures se trouvant dans le programme principal. Le rôle de celles-ci est en général le suivant :

- lecture des valeurs saisies par l'utilisateur ;
- appel à une fonction ou procédure publique d'un des paquetages réalisés ;
- affichage des résultats ;
- traitement des exceptions.

Le reste du code ne présente aucune difficulté majeure ; raison pour laquelle il n'est pas détaillé ici. Toutefois, il est possible de se reporter au code source qui contient de nombreux commentaires.

19.2 Paquetage `conversion_textes_nombres`

Ce paquetage permet de convertir des chaînes de caractères de type `String` en nombres de type `T_Long_Entier` et vice-versa.

19.2.1 Type `T_Séquence`

```

type T_Séquence is limited private;

private
  type T_Séquence is
    record
      Chaîne : Queue_Chaîne.T_Queue;
      Nombre : Queue_Nombre.T_Queue;
    end record;

```

Description Le type `T_Séquence` définit un objet permettant de stocker dans des queues dynamiques des objets `String` et `T_Long_Entier`.

19.2.2 Exceptions

`Mémoire_Insuffisante`

```
Mémoire_Insuffisante : exception;
```

Description Cette exception est levée si la mémoire disponible est insuffisante pour ajouter des éléments dans la séquence ou pour effectuer la conversion.

19.2.3 Procédure Ajouter

procédure Ajouter (Séquence : **in out** T_Séquence;
Chaîne : **in** String);

procédure Ajouter (Séquence : **in out** T_Séquence;
Nombre : **in** T_Long_Entier);

Description Ces procédures permettent respectivement d'ajouter un objet de type `String` et de type `T_Long_Entier` à une séquence en prévision d'une conversion.

Paramètres

- **Séquence** la liste contenant la séquence des objets à convertir ;
- **Chaîne** la chaîne de caractères à ajouter à la séquence ;
- **Nombre** le nombre à ajouter à la séquence.

19.2.4 Procédure Textes_En_Nombres

procédure Textes_En_Nombres (Séquence : **in out** T_Séquence;
Longueur : **in** Positive);

Description Convertit toutes les chaînes de caractères contenues dans la séquences en une nouvelle séquence de nombres. Le nombre de caractères donnant lieu à un nombre est défini par le paramètre `Longueur`.

Remarques

- les nombres éventuels contenus dans la séquence sont effacés avant le début de l'opération de conversion ;
- à la fin de la procédure, les chaînes contenues dans la séquences sont effacées.

Paramètres

- **Séquence** la liste contenant la séquence des chaînes de caractères à convertir ;
- **Longueur** définit la taille en nombre de caractères d'une chaîne donnant lieu à un nombre.

19.2.5 Procédure Nombres_En_Textes

procédure Nombres_En_Textes (Séquence : **in out** T_Séquence);

Description Convertit tous les nombres contenus dans la séquences en une nouvelle séquence de chaînes de caractères.

Remarques

- les chaînes de caractères éventuelles contenues dans la séquence sont effacées avant le début de l'opération de conversion ;
- à la fin de la procédure, les nombres contenus dans la séquences sont effacés.

Paramètres

- Séquence la liste contenant la séquence des nombres à convertir.

19.2.6 Procédure Obtenir_Nombres_G**generic**

```
with procedure Traiter (Nombre : in T_Long_Entier);
procedure Obtenir_Nombres_G (Séquence : in out T_Séquence);
```

Description Cette procédure générique appelle la procédure **Traiter** pour tous les nombres contenus dans la séquence. À la fin de la procédure, tous les nombres contenus dans la séquence sont effacés.

Paramètres

- Séquence la liste contenant la séquence des nombres à obtenir.

19.2.7 Procédure Obtenir_Textes_G**generic**

```
with procedure Traiter (Chaîne : in String);
procedure Obtenir_Textes_G (Séquence : in out T_Séquence);
```

Description Cette procédure générique appelle la procédure **Traiter** pour toutes les chaînes de caractères contenues dans la séquence. À la fin de la procédure, toutes les chaînes contenues dans la séquence sont effacées.

Paramètres

- Séquence la liste contenant la séquence des chaînes de caractères à obtenir.

19.3 Paquetage grands_entiers_G

Ce paquetage générique permet d'effectuer des opérations mathématiques sur de très grands entiers. Il met à disposition le type **T_Long_Entier** et les opérations suivantes :

- les opérateurs de comparaisons ;
- les opérateurs d'addition, soustraction, multiplication, division et exponentiation ;
- les fonctions racine carrée et logarithmique ;
- une fonction permettant de retourner le résultat d'une exponentielle modulo ;
- les fonctions de conversions de chaînes de caractères en nombres et vice-versa ;
- les constantes **Premier_Entier**, **Dernier_Entier** et **Zéro** retournant respectivement le plus petit entier représentable dans le type **T_Long_Entier**, le plus grand entier représentable et enfin le nombre 0.

19.3.1 Partie formelle générique**generic**

```
Puissance_Max : in Natural;
package Grands_Entiers_G is
```

Description Le paramètre générique `Puissance_Max` définit la taille du type privé `T_Long_Entier` tel que :

$$-(2^{Puissance_Max}) < T_Long_Entier < 2^{Puissance_Max}$$

19.3.2 Types déclarés

Bases utilisées

```
Exposant : constant := 31;
Base      : constant := 2 ** Exposant;
```

```
type T_Base      is range 0..Base - 1;
type T_Long_Base is range 0..Base ** 2 - 1;
```

Description Deux bases sont utilisées pour la représentation des grands nombres. La première, 2^{31} est celle utilisée pour le type exporté `T_Long_Entier`. La seconde, d'une taille deux fois plus importante, est utilisée en interne pour les calculs afin de tenir compte de la retenue lors de l'addition ou de la multiplication de deux entiers.

Choix Le choix de l'exposant de la base utilisée est justifié par le fait que le type prédéfini `Integer` permet de mémoriser des entiers entre $-(2^{31})$ et $2^{31} - 1$ ce qui permet d'exploiter au maximum l'espace mémoire mis à disposition pour notre type sous-jacent.

Tableau représentant les grands entiers

```
subtype T_Large_Puissance is Natural range 0..(Puissance_Max / Exposant) * 2 + 2;
subtype T_Puissance      is Natural range 0..(Puissance_Max / Exposant);

type T_Tab_Puissance is array (T_Large_Puissance range <>) of T_Base;
```

Description Le type `T_Tab_Puissance` représente le tableau dans lequel est stocké un grand entier. `T_Puissance` représente l'indice permettant de représenter le nombre max défini à l'instanciation du paquetage. `T_Large_Puissance` représente l'indice utilisé en interne pour les calculs. En effet, certaines opérations mathématiques nécessitent des calculs intermédiaires dépassant la taille maximale choisie par l'utilisateur du paquetage.

Type `T_Long_Entier`

```
type T_Long_Entier is
  record
    Nombre      : T_Tab_Puissance (T_Puissance) := (others => 0);
    Puissance   : T_Puissance                    := T_Puissance'First;
    Positif      : Boolean                        := True;
  end record;
```

Description Il s'agit du type exporté permettant de représenter un entier compris entre les bornes inférieures et supérieures définies lors de l'instanciation du paquetage.

Champs

- **Nombre** représente le tableau stockant un grand entier ;
- **T_Puissance** indique l'indice du tableau contenant le poids le plus fort de notre nombre actuellement stocké. Bien que pas absolument nécessaire, cet indicateur permet d'accélérer les calculs en évitant de balayer tout le tableau afin de rechercher l'emplacement où débute le nombre, les autres étant à zéro ;
- **Positif** vaut **True** si le nombre stocké est positif et **False** dans le cas contraire.

19.3.3 Constantes

Zéro

Zéro : **constant** T_Long_Entier;

Description Représente le nombre 0.

Dernier_Entier

Dernier_Entier : **constant** T_Long_Entier;

Description Représente le plus grand entier pouvant être contenu dans le type T_Long_Entier. Ce nombre est égal à $2^{Puissance_Max} - 1$.

Premier_Entier

Premier_Entier : **constant** T_Long_Entier;

Description Représente le plus petit entier pouvant être contenu dans le type T_Long_Entier. Ce nombre est égal à $-(2^{Puissance_Max} - 1)$.

19.3.4 Exceptions

Division_Par_Zéro

Division_Par_Zéro : **exception**;

Description Cette exception est levée lors d'une tentative de division par zéro.

Erreur_De_Contrainte

Erreur_De_Contrainte : **exception**;

Description Cette exception est levée si le résultat d'une opération est supérieur à la constante **Dernier_Entier** ou inférieur à la constante **Premier_Entier**.

Erreur_De_Conversion

Erreur_De_Conversion : **exception**;

Description Cette exception est levée par la procédure **Valeur** si la chaîne passée en paramètre ne peut pas être convertie en T_Long_Entier

19.3.5 Opérateurs de comparaison

```
function "<" (Gauche, Droite : T_Long_Entier) return Boolean;  
function ">" (Gauche, Droite : T_Long_Entier) return Boolean;  
function "<=" (Gauche, Droite : T_Long_Entier) return Boolean;  
function ">=" (Gauche, Droite : T_Long_Entier) return Boolean;
```

Description Ces quatre fonctions effectuent les tests de comparaisons mathématiques classiques entre les membres de Gauche et de Droite.

Paramètres

- Gauche représente le membre de gauche de la comparaison.
- Droite représente le membre de droite de la comparaison.

19.3.6 Fonction unaire « + »

```
function "+" (Nombre : T_Long_Entier) return T_Long_Entier;
```

Description Cette fonction retourne $+Nombre$.

Paramètres

- Nombre représente le nombre à retourner.

Implémentation Cette fonction se contente de retourner le nombre passé en paramètre. Bien que rarement utilisée, elle permet de simplifier l'écriture d'expressions mathématiques.

19.3.7 Fonction unaire « - »

```
function "-" (Nombre : T_Long_Entier) return T_Long_Entier;
```

Description Cette fonction retourne $(-Nombre)$.

Paramètres

- Nombre indique le nombre dont on aimerait le complément.

Implémentation Cette fonction effectue le complément sur le champs *Positif*.

19.3.8 Fonction unaire « abs »

```
function "abs" (Nombre : T_Long_Entier) return T_Long_Entier;
```

Description Cette fonction retourne la valeur absolue du nombre passé en paramètre.

Paramètres

- Nombre représente le nombre dont on aimerait la valeur absolue.

Implémentation Cette fonction force le champs `Positif` du type `T_Long_Entier` à `True`.

19.3.9 Fonction binaire « + »

function "+" (Gauche, Droite : T_Long_Entier) **return** T_Long_Entier **is**

Description Cette fonction retourne la somme mathématique des deux nombres passés en paramètre.

Paramètres

- `Gauche`, `Droite` les deux nombres à additionner.

Implémentation Si les deux nombres passés en paramètre ne sont pas de même signe, cette fonction se contente de retourner $Gauche - (-Droite)$. Dans le cas contraire, l'algorithme d'addition peut commencer.

Algorithme 8 Algorithme d'addition

Entrées: $(u_{n-1} \dots u_1 u_0)_b$ et $(v_{n-1} \dots v_1 v_0)_b$

Sorties: $(w_n \dots w_1 w_0)_b$

$j \leftarrow 0$

$k \leftarrow 0$

tantque $j < n$ **faire**

$w_j \leftarrow (u_j + v_j + k) \bmod b$

$k \leftarrow \lfloor (u_j + v_j + k)/b \rfloor$

$j \leftarrow j + 1$

fin tantque

$w_n \leftarrow k$

Retourner $(w_n \dots w_1 w_0)_b$.

19.3.10 Fonction binaire « - »

function "-" (Gauche, Droite : T_Long_Entier) **return** T_Long_Entier **is**

Description Cette fonction retourne la différence mathématique des deux nombres passés en paramètre.

Paramètres

- `Gauche`, `Droite` les deux nombres à soustraire.

Implémentation A l'entrée de la fonction, les cas suivants peuvent se produire :

- `Gauche` = 0. Dans ce cas, la fonction retourne $-Droite$;
- `Droite` = 0. Dans ce cas, la fonction retourne `Gauche` ;
- `Gauche` et `Droite` n'ont pas le même signe. Dans ce cas, la fonction retourne $Gauche + (-Droite)$;

- `Gauche` et `Droite` > 0 et `Gauche` $<$ `Droite`. Dans ce cas, la fonction retourne $-(\text{Droite} - \text{Gauche})$;
- `Gauche` et `Droite` < 0 et `Gauche` $>$ `Droite`. Dans ce cas, la fonction retourne $-(\text{Droite} - \text{Gauche})$;
- Dans les autres cas, l'algorithme 9 peut commencer.

Algorithme 9 Algorithme de soustraction

Entrées: $(u_{n-1} \dots u_1 u_0)_b$ et $(v_{n-1} \dots v_1 v_0)_b$

Sorties: $(w_{n-1} \dots w_1 w_0)_b$

$j \leftarrow 0$

$k \leftarrow 0$

tantque $j < n$ **faire**

$w_j \leftarrow (u_j - v_j + k) \bmod b$

$k \leftarrow \lfloor (u_j - v_j + k)/b \rfloor$

$j \leftarrow j + 1$

fin tantque

Retourner $(w_{n-1} \dots w_1 w_0)_b$.

19.3.11 Fonction binaire « * »

function "*" (`Gauche`, `Droite` : `T_Long_Entier`) **return** `T_Long_Entier` **is**

Description Cette fonction retourne le produit mathématique des deux nombres passés en paramètre.

Paramètres

- `Gauche`, `Droite` les deux nombres à multiplier.

Implémentation La multiplication est implémentée selon l'algorithme 10.

19.3.12 Procédure Division

procédure `Division` (`Gauche`, `Droite` : **in** `T_Long_Entier`;
 `Quotient` : **out** `T_Long_Entier`;
 `Reste` : **out** `T_Long_Entier`);

Description Cette procédure effectue la division entière Euclidienne de

$$\text{Gauche}/\text{Droite} \text{ tel que } \text{Gauche} = \text{Quotient} \times \text{Droite} + \text{R}$$

Paramètres

- `Gauche` le numérateur;
- `Droite` le dénominateur;
- `Quotient` le quotient de la division `Gauche / Droite`;
- `Reste` le reste de la division `Gauche / Droite`.

Algorithme 10 Algorithme de multiplication

Entrées: $(u_{m-1} \dots u_1 u_0)_b$ et $(v_{n-1} \dots v_1 v_0)_b$ **Sorties:** $(w_{m+n-1} \dots w_1 w_0)_b$ $i \leftarrow 0$ **tantque** $i < m$ **faire** $j \leftarrow 0$ **tantque** $j < n$ **faire** $k \leftarrow 0$ $t \leftarrow u_i \times v_j + w_{i+j} + k$ $w_{i+j} \leftarrow t \bmod b$ $k \leftarrow \lfloor t/b \rfloor$ $j \leftarrow j + 1$ **fin tantque** $w_{j+m} \leftarrow k$ $i \leftarrow i + 1$ **fin tantque**Retourner $(w_{m+n-1} \dots w_1 w_0)_b$

Implémentation Lors de la division, il y a lieu de différencier 2 cas :

- le dénominateur est inférieur à la base utilisée (ici 2^{31}). Dans ce cas, il est possible d'utiliser un algorithme de division simple (voir algorithme 11) ;
- le dénominateur est supérieur ou égal à la base utilisée. Cette opération est déjà nettement plus compliquée. Nous avons utilisé la méthode proposée par Donald E. Knuth¹³ (voir algorithme 12 en page 55).

Algorithme 11 Algorithme de division 1

Entrées: $(u_{n-1} \dots u_1 u_0)_b$ et v **Sorties:** $(q_{n-1} \dots q_1 q_0)_b$ $j \leftarrow n - 1$ $r \leftarrow 0$ **tantque** $j \geq 0$ **faire** $q_j \leftarrow \lfloor (rb + u_j)/v \rfloor$ $r \leftarrow (rb + u_j) \bmod v$ **fin tantque**Retourner $(q_{n-1} \dots q_1 q_0)_b$ Retourner r

19.3.13 Fonction « / »**function** "/" (Gauche, Droite : T_Long_Entier) **return** T_Long_Entier;**Description** Cette fonction retourne le quotient de la division entière Euclidienne de Gauche / Droite.

¹³DONALD E. KNUTH. *The Art of Computer Programming, Third edition*. Addison-Wesley, 1998.

Algorithme 12 Algorithme de division 2

Entrées: $(u_{m+n-1} \dots u_1 u_0)_b$ et $(v_{n-1} \dots v_1 v_0)_b$
Sorties: $(q_m \dots w_1 w_0)_b$ et $(r_{n-1} \dots r_1 r_0)_b$

$d \leftarrow \lfloor b / (v_{n-1} + 1) \rfloor$
 $(u_{m+n} \dots u_1 u_0)_b \leftarrow (u_{m+n-1} \dots u_1 u_0)_b \times d$
 $(v_{n-1} \dots v_1 v_0)_b \leftarrow (v_{n-1} \dots v_1 v_0)_b \times d$
 $j \leftarrow m$

tantque $j \leq 0$ **faire**

$\hat{q} \leftarrow \lfloor (u_{j+n} b + u_{j+n-1}) / v_{n-1} \rfloor$
 $\hat{r} \leftarrow (u_{j+n} b + u_{j+n-1}) \bmod v_{n-1}$

tantque $\hat{r} < b$ et $\hat{q} v_{n-2} > b \hat{r} + u_{j+n-2}$ **faire**

$\hat{q} \leftarrow \hat{q} - 1$
 $\hat{r} \leftarrow \hat{r} + v_{n-1}$

fin tantque

$(u_{j+n} u_{j+n-1} \dots u_j)_b \leftarrow (u_{j+n} u_{j+n-1} \dots u_j)_b - \hat{q} (v_{n-1} \dots v_1 v_0)_b$
 $q_j \leftarrow \hat{q}$

si $(u_{j+n} u_{j+n-1} \dots u_j)_b < 0$ **alors**

$q_j \leftarrow q_j - 1$
 $(u_{j+n} u_{j+n-1} \dots u_{j+1} u_j)_b \leftarrow (v_{n-1} \dots v_1 v_0)_b$

fin si

fin tantque

$(r_{n-1} \dots r_1 r_0)_b \leftarrow (u_{n-1} \dots u_1 u_0) / d$
Retourner $(q_m \dots w_1 w_0)_b$
Retourner $(r_{n-1} \dots r_1 r_0)_b$

Paramètres

- Gauche le numérateur ;
- Droite le dénominateur.

Implémentation Cette fonction fait appel à la procédure `Division`.

19.3.14 Fonction « mod »

```
function "mod" (Gauche, Droite : T_Long_Entier) return T_Long_Entier;
```

Description Cette fonction retourne le modulo de la division entière Euclidienne de Gauche / Droite.

Paramètres

- Gauche le numérateur ;
- Droite le dénominateur.

Implémentation Cette fonction fait appel à la procédure `Division`.

19.3.15 Fonction « rem »

function "rem" (Gauche, Droite : T_Long_Entier) **return** T_Long_Entier;

Description Cette fonction retourne le reste de la division entière Euclidienne de Gauche / Droite.

Paramètres

- Gauche le numérateur ;
- Droite le dénominateur.

Implémentation Cette fonction fait appel à la procédure Division.

19.3.16 Fonction « ** »

function "**" (Base : T_Long_Entier;
Exp : Natural) **return** T_Long_Entier;

function "**" (Base : Natural;
Exp : Natural) **return** T_Long_Entier;

Description Ces deux fonctions retournent le résultat de Base à la puissance Exp.

Paramètres

- Base le nombre à la base.
- Exp l'exposant.

Implémentation L'algorithme utilisé est tiré du livre *math-info* de Raymond Séroul¹⁴.

Supposons que nous devions calculer

$$x^{59} = \underbrace{x \times x \times x \dots x \times x \times x}_{58 \text{ multiplications}}$$

Il est possible de faire mieux : la décomposition de $x^{59} = x \times (x^2)^{29}$ montre que 30 multiplications suffisent :

$$y = x \times x; \quad x^{59} = x \times \underbrace{y \times y \times y \dots y \times y \times y}_{28 \text{ multiplications}}$$

Réutilisons cette idée en introduisant $z = y^2$. Alors $x^{59} = x \times y \times z^{14}$ montre que 17 multiplications suffisent :

$$y = x \times x; \quad z = y \times y; \quad x^{59} = x \times y \times \underbrace{z \times z \times z \dots z \times z \times z}_{13 \text{ multiplications}}$$

¹⁴RAYMOND SÉROUL. *math-info*. InterEditions, 1995.

La formule générale peut être exprimée de la façon suivante :

$$u^n \times v = \begin{cases} (u^2)^{n/2} \times v & \text{si } n \text{ est pair,} \\ (u^2)^{(n-1)/2} \times (u \times v) & \text{sinon.} \end{cases}$$

19.3.17 Fonction *Puissance_Modulo*

```
function Puissance_Modulo (Base   : T_Long_Entier;
                           Exp    : T_Long_Entier;
                           Modulo  : T_Long_Entier) return T_Long_Entier;
```

Description Cette fonction retourne la relation de congruence de

$$Base^{Exp} \pmod{Modulo}$$

Paramètres

- Base le nombre à la base ;
- Exp l'exposant ;
- Modulo le modulo de la relation de congruence.

Implémentation Cette fonction est implémentée selon l'algorithme suivant :

Algorithme 13 Algorithme de la Puissance modulo

Entrées: *base*, *exp* et *mod*

Sorties: *n*

j ← *n* - 1

r ← *base*

x ← 1

tantque *exp* > 1 **faire**

si *exp* mod 2 = 0 **alors**

exp ← *exp*/2

r ← (*r*²) mod *mod*

sinon

x ← (*x* × *r*) mod *mod*

exp ← *exp* - 1

fin

fin tantque

Retourner (*x* × *r*) mod *mod*

19.3.18 Fonction *Racine_Carrée*

```
function Racine_Carrée (Nombre : T_Long_Entier) return T_Long_Entier;
```

Description Cette fonction retourne la partie entière de la racine carrée du nombre passé en paramètre.

Paramètres

- Nombre le nombre dont on aimerait la racine carrée.

19.3.19 Fonction Log

function Log (Base : Positive;
 Nombre : T_Long_Entier) **return** Natural;

Description Cette fonction retourne la partie entière du logarithme en base Base du nombre passé en paramètre.

Paramètres

- Base la base du logarithme;
- Nombre le nombre dont on aimerait le logarithme.

19.3.20 Fonction PGCD

function PGCD (A, B : T_Long_Entier) **return** T_Long_Entier;

Description Cette fonction retourne le plus grand diviseur commun de A et B.

Paramètres

- A, B nombres dont on aimerait le PGCD.

19.3.21 Fonction PPCM

function PPCM (N, M : T_Long_Entier) **return** T_Long_Entier;

Description Cette fonction retourne le plus petit multiple commun de M et N.

Paramètres

- M, N nombres dont on aimerait le PPCM.

19.3.22 Fonction Max

function Max (A, B : T_Long_Entier) **return** T_Long_Entier;

Description Cette fonction retourne A si $A > B$ et B dans tous les autres cas.

Paramètres

- A, B nombres dont on aimerait le maximum.

19.3.23 Fonction Min

function Min (A, B : T_Long_Entier) **return** T_Long_Entier;

Description Cette fonction retourne A si $A < B$ et B dans tous les autres cas.

Paramètres

- A, B nombres dont on aimerait le minimum.

19.3.24 Fonction Image

function Image (Nombre : T_Long_Entier) **return** String;

Description Cette fonction retourne l'image en base 10 du nombre passé en paramètre.

Paramètres

- Nombre nombre dont on aimerait une image sous forme de chaîne de caractères.

19.3.25 Fonction Valeur

function Valeur (Image : String) **return** T_Long_Entier;

function Valeur (Image : Integer) **return** T_Long_Entier;

Description Ces deux fonctions construisent un nombre T_Long_Entier en fonction de la valeur passée en paramètre.

Paramètres

- Image image d'un nombre de type T_Long_Entier.

19.4 Paquetage *grands_entiers_G-aleatoires_G*

Ce paquetage enfant de *grands_entiers_g* permet de générer des nombres pseudo-aléatoires portant sur des objets de type T_Long_Entier.

19.4.1 Exceptions**Aléatoire_Non_Initialisé**

Aléatoire_Non_Initialisé : **exception**;

Description Cette exception est levée si une des fonctions Initialise_Aléatoire n'a pas été appelée avant l'utilisation de la fonction Aléatoire.

19.4.2 Procédure Initialise_Aléatoire

procedure Initialise_Aléatoire

procedure Initialise_Aléatoire (Initialisation : **in** Integer);

Description Ces deux procédures initialisent le générateur pseudo-aléatoires. Alors que la première utilise l'heure du système, la seconde permet de passer en paramètre un entier qui sera utilisé pour l'initialisation du générateur. Cette option permet de générer plusieurs fois de suite la même séquence de nombres aléatoire en indiquant la même valeur d'initialisation.

Paramètres

- `Initialisation` le germe de la séquence pseudo-aléatoire.

Implémentation Ces deux procédures appellent la procédure `Reset` du paquetage `Ada.Numerics.Discrete_Random`.

19.4.3 Fonction Aléatoire

function Aléatoire **return** T_Long_Entier;

Description Cette fonction retourne un nombre pseudo-aléatoire compris dans l'intervalle `Premier_Entier ≤ Aléatoire ≤ Dernier_Entier`

Implémentation Cette procédure remplit le tableau contenu dans `T_Long_Entier` de nombres tirés par la fonction `Random` du paquetage `Ada.Numerics.Discrete_Random`. De plus, elle génère une variable de type `Boolean` afin de définir le signe du nombre retourné.

19.5 Paquetage `jewl_windows_extension`

Le paquetage `jewl` d'origine ne permettant pas la création d'un champ de saisie multi-lignes sans ascenseur horizontal, nous avons décidé de contourner ce problème en écrivant un paquetage enfant de la librairie `jewl.windows` dans le but d'y ajouter cette fonctionnalité.

19.5.1 Fonction Multiline

```
function Multiline (Parent   : Container_Type'Class;
                    Origin   : Point_Type;
                    Width    : Integer;
                    Height   : Integer;
                    Font     : Font_Type := Parent_Font;
                    Editable : Boolean   := False) return Memo_Type;
```

Description Cette fonction crée un champ d'édition multi-lignes dans le conteneur `Parent`, aux coordonnées `Origin`, de dimensions `Width` et `Height`.

Paramètres

- `Parent` Conteneur parent du nouveau `Multiline` à créer.
- `Origin` Coordonnées relatives au parent.
- `Width` Largeur du champ `Multiline`
- `Height` Hauteur du champ `Multiline`
- `Font` Police de caractères du champ `Multiline`
- `Editable` Si = `True`, le champ est éditables par l'utilisateur. Dans le cas contraire, rien ne pourra être saisi dans ce champ.

19.6 Paquetage *nombres_premiers*

Ce paquetage met à disposition une liste de petits nombres premiers, des tests de primalité, une fonction de génération de très grands nombres premiers aléatoires et une fonction de factorisation.

19.6.1 Exceptions

Pas_De_Premiers_Inf_A_Deux

`Pas_De_Premiers_Inf_A_Deux` : **exception**;

Description Levée si le paramètre `Max` est plus grand ou égal à 2, car il n'existe aucun nombre premier inférieur à 2.

Nombre_Non_Positif

`Nombre_Non_Positif` : **exception**;

Description Levée si $N \leq 0$.

Factorisation_Impossible

`Factorisation_Impossible` : **exception**;

Description Levée si la factorisation ne peut se faire dans un temps raisonnable (de l'ordre d'une dizaine de minutes) ou si il y a plus de 256 facteurs.

19.6.2 Constantes

Constante Crible_Max

`Crible_Max` : **constant** `Positive := 600`;

Description Définit la limite supérieure des nombres premiers du crible d'Ératosthène.

Choix En effectuant des tests empiriques, nous avons constaté que la valeur 600 était la plus adaptée à une factorisation rapide (6.1.2, 19.6.12).

19.6.3 Fonction Crible

function `Crible` (`Max` : `Positive`) **return** `T_Tab_Positif`;

Description Cette fonction retourne la liste des nombres premiers inférieurs ou égaux à `Max`, la méthode du crible d'Ératosthène (voir 5.1 page 20).

Paramètres

- `Max` la fonction ne renvoie que les nombres premiers inférieurs à `Max`.

Implémentation Cette fonction commence par générer un tableau de valeurs booléennes dont les indices vont de 2 à Max. Toutes les valeurs sont vraies. Ensuite, pour chaque indice en partant de 2, les valeurs contenues dans les multiples de l'indice sont déclarées fausses. Enfin, pour gagner de la place et par commodité pour les calculs, la fonction normalise le résultat et le stocke dans un tableau d'entiers, qu'elle trie à l'aide d'un tri par sélection. Le crible est fabriqué à l'élaboration du paquetage et conservé en mémoire durant la durée de vie du programme.

Dans *AdaRSA*, le crible contient les nombres premiers inférieurs à 600. Cette valeur est fixée en constante dans le paquetage. D'après nos tests, c'est avec cette valeur que la fonction `Factorisation`, qui cherche des facteurs dans le crible avant d'utiliser l'algorithme ρ de Pollard, est la plus efficace.

Nous avons commencé par implémenter le crible d'Ératosthène à l'aide d'une liste dynamique, en pensant ainsi économiser de la mémoire. En fait, la fonction est bien plus rapide dans son implémentation par un tableau de booléens et n'occupe pas plus de mémoire.

19.6.4 Fonction `Est_Premier_Crible` (sur le type `Positive`)

```
function Est_Premier_Crible (N : Positive) return Boolean;
```

Description Cette fonction indique s'il existe un facteur de N dans le crible d'Ératosthène.

Paramètres

- N le nombre dont on cherche à savoir s'il est premier.

Implémentation Cette fonction tente de diviser N par tous les facteurs premiers présents dans le crible et indique si elle a trouvé un facteur (c'est-à-dire si le pgcd de N et du nombre pris dans le crible = 1). Si N est plus grand que le carré de la dernière valeur du crible, le test n'est pas suffisant pour déclarer le nombre premier. Dans le cas inverse, le test est déterministe.

19.6.5 Fonction `Est_Premier_Crible` (sur le type `T_Long_Entier`)

```
function Est_Premier_Crible (N : Positive) return Boolean;
```

Description Il s'agit de la même fonction que la fonction précédente, à la différence que celle-ci est implémentée de manière à traiter des nombres du type `T_Long_Entier`.

19.6.6 Procédure `Afficher_Crible`

```
procedure Afficher_Crible;
```

Description Affiche le contenu de la variable globale `Crible` de type `T_Queue`.

19.6.7 Fonction `Premiers_Entre_Eux`

```
function Premiers_Entre_Eux (N, M : T_Long_Entier) return Boolean;
```

Description Indique si deux nombres sont premiers entre eux, c'est-à-dire si leur pgcd vaut 1.

Paramètres

- N et M les nombres dont on veut savoir s'ils sont premiers entre eux.

19.6.8 Fonction `Est_Premier_Fermat`

```
function Est_Premier_Fermat (N          : T_Long_Entier;
                             Nb_Tests : Positive      := 20) return Boolean;
```

Description Test probabiliste indiquant si N est peut-être premier, à l'aide du petit théorème de Fermat (voir C.6 page 80). Si le test est positif, N est peut-être premier. S'il est négatif, N est composé. Le test manque de détecter les nombres composés une fois sur deux. En répétant le test 20 fois, la probabilité d'erreur est donc proche 0.0001%, sans toutefois être bornée du fait de l'existence de nombres particuliers, les nombres de Carmichael : (561, 1105, 1729, ...). Voir 5.2.1 page 21.

Paramètres

- N le nombre dont on veut tester la primalité;
- Nb_Tests le nombre de tests à réaliser.

Complexité $O(R \cdot n^3)$, pour R répétitions

19.6.9 Fonction `Est_Premier_Miller_Rabin`

```
function Est_Premier_Miller_Rabin (N          : T_Long_Entier;
                                    Nb_Tests : Positive := 10) return Boolean;
```

Description Test probabiliste indiquant si N est peut-être premier, à l'aide du test de Miller-Rabin (voir 5.2.2 page 22). Si le test est positif, N est peut-être premier. S'il est négatif, N est composé. Le test manque de détecter les nombres composés une fois sur quatre. En répétant le test 10 fois, la probabilité d'erreur est de 0.0001%.

Paramètres

- N le nombre dont on veut tester la primalité;
- Nb_Tests le nombre de tests à réaliser.

Implémentation L'implémentation est basée sur l'algorithme P présenté p. 379 du livre de Knuth [1].

Complexité $O(\log n)$

19.6.10 Fonction Est_Premier

```
function Est_Premier (N          : T_Long_Entier;  
                    Nb_Tests : Positive := 10) return Boolean;
```

Description Indique si un nombre est premier (voir 5.3 page 22).

Paramètres

- N le nombre dont on veut tester la primalité;
- Nb_Tests le nombre de tests à réaliser.

Implémentation Indique si N est premier en interrogeant successivement les tests du crible d'Ératosthène et de Miller-Rabin. Si au cours de ces test on détermine que le nombre est composé, le test s'arrête et retourne False. Sinon, le test est mené à terme et retourne True. Le nombre N a alors de très grandes chances d'être premier. Par défaut, le test de Miller-Rabin est répété 10 fois, pour obtenir une marge d'erreur de 0.0001%.

19.6.11 Fonction Premier_Aléatoire

```
function Premier_Aléatoire (Max : in T_Long_Entier) return T_Long_Entier;
```

Description Renvoie une nombre premier aléatoire inférieur à Max.

Paramètres

- Max la borne supérieure.

Implémentation La fonction travaille bêtement en générant des nombres aléatoires jusqu'à en trouver un qui soit premier. Par le théorème des nombres premiers (voir C.2 page 79), on sait que pour trouver un nombre premier de n chiffres, on doit générer en moyenne $2 \cdot n$ nombres aléatoires avant d'en trouver un qui soit premier. Cela reste très raisonnable !

Le générateur aléatoire est initialisé lors de l'élaboration du paquetage.

19.6.12 Fonction Factorisation

```
function Factorisation (Nombre : T_Long_Entier) return T_Tab_Long_Entier;
```

Description Renvoie la décomposition de N en facteurs premiers dans un tableau trié. Le tableau comprend également le chiffre 1. Voir aussi 6 page 23.

Paramètres

- N le nombre à décomposer.

Implémentation La fonction tente d'abord de trouver un facteur de N dans le crible d'Ératosthène, puis par la méthode ρ de Pollard (voir 6.1 page 24. Cette méthode permet de trouver efficacement les facteurs de nombres de moins de 20 chiffres. Une fonction récursive décompose les facteurs trouvés par la méthode de Pollard en facteurs premiers. La factorisation s'arrête si on ne trouve pas de facteur après un million d'itérations de l'algorithme ρ . La fonction lève une exception si le nombre comprend plus de 256 facteurs.

Nous avons implémenté plusieurs autres algorithmes de factorisation avant de choisir celui-ci car il était plus efficace.

19.7 Paquetage *queue_g*

Ce paquetage générique met à disposition un type de donnée abstrait permettant la gestion de queues dynamiques. Il met à disposition les opérations classiques (dépôt, copie, modification, suppression, ...) ainsi qu'une procédure générique permettant de parcourir une queue en effectuant un traitement sur chacun de ses éléments. À sa création, une queue est toujours vide.

19.7.1 Partie formelle générique

```
generic
  type T_Elément (<>) is private;
```

Description Le paramètre générique `T_Elément` définit le type des données contenues dans la queue.

19.7.2 Exceptions

Erreur_Queue_Vide

```
Erreur_Queue_Vide : exception;
```

Description Cette exception est levée quand l'opération demandée ne peut s'effectuer car la queue est vide.

Erreur_Queue_Pleine

```
Erreur_Queue_Pleine : exception;
```

Description Cette exception est levée quand la mémoire disponible est insuffisante pour réaliser l'opération demandée.

19.7.3 Procédure Déposer

```
procédure Déposer (Queue   : in out T_Queue;
                  Elément : in      T_Elément);
```

Description Dépose `Elément` en queue de la queue.

Paramètres

- Queue objet queue où l'élément doit être déposé ;
- Elément élément à déposer dans la queue.

19.7.4 Procédure Supprimer

procédure Supprimer (Queue : **in out** T_Queue);

Description Supprime l'élément se trouvant en tête de la queue.

Paramètres

- Queue objet queue dont on veut supprimer l'élément de tête.

19.7.5 Procédure Modifier

procédure Modifier (Queue : **in out** T_Queue;
Elément : **in** T_Elément);

Description Remplace l'élément de tête de la queue avec l'élément passé en paramètre.

Paramètres

- Queue objet queue.
- Elément élément à modifier.

19.7.6 Fonction Tête

function Tête (Queue : T_Queue) **return** T_Elément;

Description Renvoie l'élément se trouvant en tête de la queue.

Paramètres

- Queue objet queue.

19.7.7 Procédure Vider

procédure Vider (Queue : **in out** T_Queue);

Description Supprime tous les éléments se trouvant dans la queue.

Paramètres

- Queue objet queue.

19.7.8 Procédure Copier

procédure Copier (Source : **in** T_Queue;
Destination : **in out** T_Queue);

Description Copie la queue Source dans la queue Destination.

Paramètres

- Source queue source à copier.
- Destination queue de destination.

19.7.9 Fonction Longueur

function Longueur (Queue : T_Queue) **return** Natural;

Description Renvoie le nombre d'éléments qui se trouvent dans la queue.

Paramètres

- Queue objet queue.

19.7.10 Fonction Est_Vide

function Est_Vide (Queue : T_Queue) **return** Boolean;

Description Indique si la queue est vide.

Paramètres

- Queue objet queue.

19.7.11 Procédure Parcourir**generic**

with procedure Traiter(Element: **in** T_Élément);
procedure Parcourir(Queue: **in** T_Queue);

Description Parcours la queue et applique un traitement à chaque élément. Cette procédure doit être instanciée avec une procédure de traitement.

Paramètres

- Queue la queue à parcourir.

19.8 Paquetage rsa

Ce paquetage met à disposition des types et des fonctions permettant de générer et stocker des clés RSA, ainsi que de chiffrer et de déchiffrer un message. La valeur de l'exposant public est fixée à 65537 (voir 2.5 page 15).

19.8.1 Exceptions**Cle_Trop_Courte**

Cle_Trop_Courte : **exception**;

Description Cette exception est levée dans la fonction **Chiffrer** quand le message clair est plus petit que l'exposant public moins un, soit $M < N - 1$.

19.8.2 Type T_Clés

```
subtype T_Clé is T_Long_Entier;
```

Description Définition du type des valeurs entières utilisés dans les clés.

19.8.3 Type T_Clé_Publique

```
type T_Clé_Publique is
  record
    E : T_Clé;
    N : T_Clé;
  end record;
```

Description Définition de la structure stockant une clé publique RSA

19.8.4 Type T_Clé_Privée

```
type T_Clé_Privée is
  record
    D : T_Clé;
    N : T_Clé;
  end record;
```

Description Définition de la structure stockant une clé privée RSA.

19.8.5 Type T_Couple_Clés

```
type T_Couple_Clés is
  record
    Publique : T_Clé_Publique;
    Privée   : T_Clé_Privée;
  end record;
```

Description Définition de la structure stockant un couple de clés RSA.

19.8.6 Type T_Souche

```
type T_Souche is
  record
    P : T_Long_Entier;
    Q : T_Long_Entier;
    Phi : T_Long_Entier;
  end record;
```

Description Définition de la structure stockant une souche RSA, soit les valeurs P, Q et Phi.

19.8.7 Fonction Générer_Souche

function Générer_Souche (Taille : Positive) **return** T_Souche;

Description Renvoie les valeurs de base P, Q et Phi servant à générer un couple de clés RSA, en fonction de la taille des clés souhaitée. Cette fonction est utile pour le calcul des clés mais aurait pu rester privée. Elle est publique dans un but pédagogique.

Paramètres

- Taille la taille des clés souhaitée (en bits).

19.8.8 Fonction Générer_Clés

function Générer_Clés (Souche : T_Souche) **return** T_Couple_Clés **is**

Description Génère un couple de clés RSA en fonction d'une souche donnée.

Paramètres

- Souche les valeurs P, Q et Phi permettant le calcul des clés.

Implémentation L'implémentation suit la méthode discutée en 2.1 page 11. Il est possible de choisir un E aléatoire en décommentant les lignes adéquates.

19.8.9 Fonction Chiffrer

function Chiffrer (M : T_Long_Entier;
Clé : T_Clé_Publique) **return** T_Long_Entier **is**

Description Chiffre un message à l'aide d'une clé publique RSA.

Paramètres

- M le message à chiffrer ;
- Clé la clé publique utilisée.

Implémentation L'implémentation suit la méthode discutée en 2.2 page 12. Une exception est levée si le message clair est plus petit que l'exposant public moins 1.

19.8.10 Fonction Déchiffrer

function Déchiffrer (C : T_Long_Entier;
Clé : T_Clé_Privée) **return** T_Long_Entier **is**

Description Déchiffre un message à l'aide d'une clé privée RSA.

Paramètres

- C le message à déchiffrer ;
- Clé la clé privée utilisée.

Implémentation L'implémentation suit la méthode discutée en 2.2 page 12.

Quatrième partie

Annexes

A Cahier des charges

État au 1er avril 2003.

Thème Étude des grands nombres premiers et applications cryptographiques.

Buts envisagés du programme

1. Génération aléatoire de grands nombres premiers, études des algorithmes.
2. Détermination de la primalité d'un nombre, étude des algorithmes.
3. Applications cryptographiques : étude et implémentation d'un ou plusieurs algorithmes (probablement l'algorithme RSA).
4. « Fabrication » d'une petite application graphique permettant de crypter et décrypter des fichiers, en utilisant les algorithmes que nous aurons implémentés.
5. Cette application n'a pas la prétention d'implémenter un système de cryptage particulièrement « solide ». Il devra être avant tout didactique et expliquer les opérations effectuées.

Points particuliers à étudier

1. Il nous faudra trouver un moyen de gérer les très grands nombres (hors des bornes du type `Ada Long_Long_Integer`). Pour cela, il faudra à la fois déterminer une structure de données adéquate et les opérations mathématiques s'y rapportant.
2. Il nous faudra nous assurer que le générateur de nombres aléatoires ait une entropie maximale. Autrement dit, il faudra que le nombre de questions à poser pour trouver la clé soit le plus grand possible.

Structure et technologies mises en œuvre

1. Le programme principal s'appuiera sur différents paquetages. L'un d'entre eux sera consacré à la génération et l'étude de grands nombres premiers. Les implémentations d'algorithmes particuliers feront elles aussi l'objet de paquetages. Cette structure modulaire est celle envisagée lors de la planification du projet. Il est probable qu'elle évoluera au cours de l'élaboration du programme.
2. Le programme sera codé en Ada 95. Pour la représentation graphique, nous utiliserons une librairie graphique telle que `Jewel`.

Planification

Définition des algorithmes et structures de données	mi-avril
Spécifications et codage des paquetages	mi-mai
Phase de tests	fin mai
Implémentation graphique	début juin
Test finaux	mi-juin
Établissement du rapport	début juillet

B Journal du projet

Nous présentons ici le déroulement du projet, qui a eu lieu entre avril et juillet 2003. On trouvera le travail effectué en regard du travail planifié.

B.1 Début avril

Travail planifié Définition des algorithmes et structures de données.

Travail effectué

Préparation

- recherche de documentation sur la cryptographie, le calcul sur des grands entiers, les tests de primalité et la génération de nombres pseudo-aléatoire ;
- répartition du travail.

B.2 Fin avril et début mai

Travail planifié Spécifications et codage des paquetages.

Travail effectué

Paquetage `grands_entiers_g`

- définition des structures de données ;
- implémentation des fonctions d'addition, soustraction, multiplication, division et exponentiation ;
- implémentation des opérateurs de comparaisons ;
- implémentation des opérateurs unaires "+", "-" et "abs" ;
- implémentation des fonctions de conversion de `String` en `T_Long_Entier` et vice-versa ;
- implémentation des fonctions PPCM et PGCD ;
- implémentation des fonctions `Max` et `Min` ;
- implémentation de la fonction `Puissance_Modulo` ;
- implémentation de la fonction `Racine_Carrée`.

Paquetage `grands_entiers_g.aléatoire_g`

- implémentation des fonctions et procédures permettant l'initialisation du générateur et le retour de valeurs aléatoires.

Paquetage `nombres_premiers`

- implémentation du crible d'Ératosthène à l'aide d'une queue dynamique ;
- implémentation du test de Fermat ;
- implémentation du test de Miller-Rabin.

Paquetage rsa

- définition des structures de données ;
- génération des clés publiques et privées ;
- chiffrement et déchiffrement.

B.3 Fin mai

Travail planifié Phase de tests.

Travail effectué**Programme principal adarsa**

- définition de l'aspect visuel de l'interface graphique.

Paquetage grands_entiers_g

- tests des fonctions et procédures exportées du paquetage.

Paquetage nombres_premiers

- ajout de la fonction Factorisation ;
- abandon du paquetage Queue_Chainee_G, utilisation de Queue_G ;
- modification de Factorisation pour retourner un tableau non contraint.

Paquetage rsa

- ajout de la fonction Générer_Souche pour renvoyer des valeurs intermédiaires ;
- fixé l'exposant public à 65537, pour gagner du temps ;
- gestion de la taille des clés en bits.

Paquetage queue_g

- codage du paquetage générique permettant la création de queues dynamiques.

Paquetage conversion_textes_nombres

- définition de la structure de données ;
- implémentation des procédures permettant d'ajouter des éléments dans la structure ;
- implémentation des procédures de conversions.

B.4 Début juin

Travail planifié Implémentation graphique.

Travail effectué

Programme principal adarsa

- écriture du programme ;
- rédaction des fichiers d'aide ;
- recherche des poèmes à inclure avec le logiciel.

Paquetage grands_entiers_g

- implémentation de la fonction `log` ;
- modification de la structure interne afin de permettre l'exécution des fonctions `"/"`, `"mod"`, `"rem"`, `Division` et `Puissance_Modulo` avec toute la gamme des valeurs de `T_Long_Entier`. Auparavant ces fonctions nécessitaient une certaine marge de réserve au niveau de la taille ;
- correction d'un bug dans la fonction `Puissance_Modulo`.

Paquetage nombres_premiers

- réimplémentation du crible d'Ératosthène sur un tableau de booléen ;
- découverte d'un bug dans le test de Miller-Rabin et réimplémentation complète ;
- corrections de bugs dans le test de Fermat ;
- corrections de bugs dans le test de Miller-Rabin ;
- fixé la marge d'erreur des tests de primalité à 0.0001% ;
- améliorations à la fonction `Factorisation`, puis implémentation de plusieurs algorithmes avant d'opter pour le Rho de Pollard ;
- affinage de la gestion des exceptions.

Paquetage rsa

- affinage de la gestion des exceptions.

Paquetage jewl.windows.extensions

- implémentation de la fonction `Multiline`.

Rapport

- rédaction du rapport avec \LaTeX .

B.5 Mi-juin

Travail planifié Test finaux.

Travail effectué**Programme d'installation**

- définition de la procédure d'installation du logiciel *AdaRSA* ;
- définition des fichiers à inclure lors de l'installation ;
- rédaction du script d'installation.

Rapport

- rédaction du rapport avec \LaTeX ;
- rédaction du guide de l'utilisateur avec \LaTeX .

B.6 Fin juin et début juillet

Travail planifié Etablissement du rapport.

Travail effectué

Rapport

- rédaction du rapport avec \LaTeX .

C Formulaire et tables

C.1 Théorème fondamental de l'arithmétique

Théorème 1 *Tout entier supérieur ou égal à 2 se laisse décomposer en un produit de nombres premiers. De plus, la décomposition est unique à l'ordre des facteurs près.*

C.2 Théorème des nombres premiers

Théorème 2 *Il y a environ $\frac{n}{\ln n}$ nombres premiers entre 1 et x .*

En d'autres termes, un nombre de C chiffres a un peu moins de une chance sur $2C$ d'être premier.

Ce théorème a été démontré en 1851 par Tchebychev qui montra que, plus généralement, en notant $\pi(x)$ le nombre de nombres premiers inférieurs ou égaux à x , on a :

$$0.9 \cdot \frac{x}{\ln x} \leq \pi(x) \leq 1.2 \cdot \frac{x}{\ln(x)} \quad x > 30$$

C.3 La fonction indicatrice de Euler

La fonction indicatrice de Euler, notée ϕ , est définie dans \mathbb{Z} . $\Phi(n)$ est le nombre d'entiers premiers avec n , inférieurs à n .

Quelques propriétés

1. Φ est toujours plus grand ou égal à $\frac{n}{2}$.
2. Si p est premier, alors $\phi(p) = p - 1$.
3. Si n et m sont premiers entre eux, alors $\phi(nm) = \phi(n)\phi(m)$.
4. Si n est une puissance d'un nombre premier, $n = p^k$, alors $\phi(p^k) = p^k - p^{k-1}$.
5. En particulier, si p et q sont premiers, alors $\phi(pq) = (p - 1)(q - 1)$.

C.4 Le théorème de Bezout

Théorème 3 *Soient a et b deux entiers de $d = \text{pgcd}(a, b)$. Alors il existe u et v dans \mathbb{Z} tels que*

$$au + bv = d$$

C.5 Le théorème de Euler

Théorème 4 *Soient ϕ la fonction indicatrice de Euler, n un entier et a un entier premier avec n , alors*

$$a^{\phi(n)} \pmod n = 1$$

C.6 Le petit théorème de Fermat

Théorème 5 Soit m un nombre premier, alors pour tout entier a :

$$a^m \pmod m = a$$

En particulier, si m ne divise pas a , alors :

$$a^{m-1} \pmod m = 1$$

C.7 L'algorithme d'Euclide

L'algorithme d'Euclide permet, étant donnés deux entiers a et b , de calculer leur plus grand diviseur commun (pgcd) d . Cet algorithme se base sur la propriété suivante :

$$\text{pgcd}(a, b) = \begin{cases} a & \text{si } b = 0 \\ \text{pgcd}(b, a \pmod b) & \text{sinon} \end{cases}$$

Algorithme 14 Algorithme d'Euclide

Entrées: $a, b \in \mathbb{N}$

Sorties: le pgcd de a et b

$x \leftarrow \max(a, b)$

$y \leftarrow \min(a, b)$

tantque $y \neq 0$ **faire**

$x \leftarrow y$

$y \leftarrow x \pmod y$

fin tantque

Retourner x .

Complexité Cet algorithme est polynomial en $O(\log_n + \log_m)$.

C.8 L'algorithme d'Euclide étendu

Cet algorithme calcule le pgcd d de deux entiers a et b , mais il retourne en plus un entier c tel que

$$bc \pmod a = d$$

Remarquons que dans le cas où a et b sont premiers entre eux, d vaut 1 donc c est l'inverse multiplicatif de b modulo a .

La complexité de cet algorithme est linéaire.

Algorithme 15 Algorithme d'Euclide étendu

Entrées: $a, b \in \mathbb{N}, a > b$ **Sorties:** le pgcd d de a et b et l'entier c tel que $bc \pmod a = 1$ $r_1 \leftarrow a$ $r_2 \leftarrow b$ $t_1 \leftarrow 0$ $t_2 \leftarrow 1$ **tantque** $r_2 \neq 0$ **faire** $aux_1 \leftarrow r_1$ $r_1 \leftarrow r_2$ $r_2 \leftarrow aux_1 \pmod{r_2}$ $aux_2 \leftarrow t_1$ $t_1 \leftarrow t_2$ $t_2 \leftarrow aux_2 - (aux_1/r_2)t_2$ **fin tantque**Retourner r_1 et t_1 .

C.9 Les clés RSA

Nb. de bits	Nb. de chiffres	MIPS.an	Factorisation
128	39		
256	78	< 1	1985
512	155	8000	1999
1024	309	3×10^9	2030
2048	617	7×10^{19}	2080
4096	1233	2×10^{31}	?

Un MIPS.an correspond à 2^{45} opérations élémentaires

Sous Factorisation, on trouve la date réelle ou prévue de la factorisation.

D Jeux de tests

D.1 Cadre *Chiffrement RSA*

Ouvrir des clés publiques ou privées

Contenu du fichier	Résultat
<rien>	Le fichier choisi ne contient pas une clé valide
65537 <rien>	Le fichier choisi ne contient pas une clé valide
65537 <Un nombre de 700 chiffres>	Le fichier choisi ne contient pas une clé valide
X Y	Le fichier choisi ne contient pas une clé valide
0 0	<La clé est chargée en mémoire> ¹⁵

Chiffrer et déchiffrer des messages

Message	Action	Résultat
<rien>	Chiffrer	0
0	Déchiffrer	<rien>
<rien>	Déchiffrer	<rien>
Bonjour <Le nombre N>	Chiffrer	un nombre N
	Déchiffrer	Bonjour
Bonjour <800 chiffres>	Déchiffrer	Ce texte n'est pas un message chiffré
	Déchiffrer	Ce texte n'est pas un message chiffré

D.2 Cadre *Nombres premiers*

Longueur du nombre premier aléatoire	Résultat
<rien>	Veillez saisir un entier entre 1 et 616
asd	Veillez saisir un entier entre 1 et 616
123.456	Veillez saisir un entier entre 1 et 616
123 678	<un nombre premier aléatoire de 123 chiffres> Veillez saisir un entier entre 1 et 616

¹⁵Un message d'erreur sera généré lors de l'opération de chiffrement ou de déchiffrement du message.

Nombre considéré	Action	Résultat
<rien> asd 123,456	<toutes> <toutes> <toutes>	Le champ 'Nombre considéré' ne contient pas un nombre valide. Le champ 'Nombre considéré' ne contient pas un nombre valide. Le champ 'Nombre considéré' ne contient pas un nombre valide.
-1, 0 ou 1	Test de primalité	Ce nombre n'est pas premier.
-1	Factorisation	La factorisation est possible uniquement avec des valeurs positives.
0	Factorisation	La factorisation est possible uniquement avec des valeurs positives.
1	Factorisation	1×1
2	Crible d'Eratosthène	Ce nombre est premier.
2	Fermat	Ce nombre est peut-être premier.
2	Miller-Rabin	Il y a 99.9999% de chances que ce nombre soit premier.
2	Factorisation	1×2
123456789	Test de primalité	Ce nombre n'est pas premier.
123456789	Factorisation	$1 \times 3 \times 3 \times 3607 \times 3803$
75361	Crible d'Eratosthène	Ce nombre n'est pas premier.
75361	Fermat	Ce nombre est peut-être premier. ^a
75361	Miller-Rabin	Ce nombre n'est pas premier.
75361	Factorisation	$1 \times 11 \times 13 \times 17 \times 31$
345907739	Crible d'Eratosthène	Ce nombre est peut-être premier.
345907739	Fermat	Ce nombre est peut-être premier.
345907739	Miller-Rabin	Il y a 99.9999% de chances que ce nombre soit premier.
345907739	Factorisation	1×345907739
3149126826180217	Crible d'Eratosthène	Ce nombre est peut-être premier. ^b
3149126826180217	Fermat	Ce nombre n'est pas premier.
3149126826180217	Miller Rabin	Ce nombre n'est pas premier.
3149126826180217	Factorisation	$1 \times 51606307 \times 61022131$
<800 chiffres>	Test de primalité	Le nombre à considérer est trop grand.
<800 chiffres>	Factorisation	Le nombre à considérer est trop grand.
522343470305884975641501842557 ^c	Factorisation	Il est impossible de factoriser ce nombre dans un temps raisonnable.

^aIl s'agit d'un nombre de Carmichael.^bLe crible ne contient aucun diviseur de ce nombre.^c556547443271317 \times 938542574619721

E Contenu du CD-ROM

Le CD-ROM de développement contient le programme *AdaRSA*, la documentation, le programme d'installation et tout les fichiers sources utilisés pour réaliser le développement de notre projet de semestre. Nous avons organisé le CD de la façon suivante :

La racine du CR-ROM contient le programme d'installation `setup.exe`. Celui-ci installe *AdaRSA* sur le disque dur de l'utilisateur.

Le répertoire `AdaRSA` contient le programme *AdaRSA* et les fichiers associés tels qu'ils doivent être installés dans un environnement de production. En copiant la structure de ce répertoire sur un disque local, il est possible de ne pas utiliser le programme d'installation fourni. Il est également possible de lancer l'application directement depuis ce répertoire du CD-ROM.

Le répertoire `Documentation` contient le présent rapport et le guide de l'utilisateur au format PDF. Le sous-répertoire `sources` contient les fichiers sources \LaTeX ayant servi à l'élaboration de la documentation.

Le répertoire `Développement` contient trois sous-répertoires organisés de la façon suivante :

- `Sources AdaRSA` ce répertoire contient toutes les sources Ada écrites par nos soins ;
- `Jewl` ce répertoire contient les sources et la documentation de la librairie du même nom permettant la création de fenêtre graphique ;
- `Installateur` ce répertoire contient le fichier source permettant de générer le programme d'installation ainsi que tous les fichiers nécessaires à son exécution. Le sous-répertoire `Compilateur NSIS` contient le programme d'installation du compilateur NSIS.

F Listages complets

Listings