RAPPORT PROJET COMPLEX :

TESTS DE PRIMALITÉ

Réalisé par :

BAMBA Ibrahim

NITCHEU TCHEUFFA Paul

INTRODUCTION

Le but de ce projet est de présenter certaines méthodes utilisées pour tester la primalité d'un nombre entier, un des problèmes les plus importants et les plus utiles en arithmétique et donc en cryptographie. Nous avons implanté dans le langage de programmation python trois algorithmes pour ce test dont un naïf et déterministe et les deux autres probabilistes. Dans la suite de ce rapport, nous allons expliquer les résultats obtenus avant d'étaler nos choix d'implémentation et les difficultés rencontrées.

I. PRÉSENTATION DU TRAVAIL

Notre travail final (rendu sous forme d'archive .tar.gz) contient 2 types de fichiers :

- rapport.pdf : ce document-ci, contient la description du travail
- Fichiers sources (.py) :
    - arithmetic.py : contient le code source de l'exo 1
    - carmichael.py : code source de l'exo 2
    - fermat.py : implémentation de l'exo 3
    - miller_rabin.py : celle de l'exo 4
    - courbes.py : contient les codes pour les tracées de courbes afin de déterminer expérimentalement les complexités de nos fonctions

II. RÉPONSES AUX QUESTIONS

Dans cette partie, nous répondons aux questions posées dans le sujet :

1) Réponses aux questions de Exercice 1 : Arithmétique dans Zn

1a) Pour le programme de **my_pgcd** (cf. code), notre stratégie de test a été la suivante :
Nous avons généré aléatoirement a,b dans 8 intervalles différents à savoir de [1,10], $[10, 10^2], ..., [10^8, 10^9],$ et nous avons ensuite exécuter **my_pgcd** sur chaque nombre tiré des intervalles. Le tableau 1 présente un exemple de résultat obtenu :

Tableau 1. Résultats du test de my_pgcd

a	5	71	328	9001	83979	585203	9372276	69389442	213952814
b	8	86	459	8777	79153	722101	1257501	74985507	399743084
my_pgcd(a, b)	1	1	1	1	1	1	771	9	2

1b) De même qu'en 1a) on a généré des nombres aléatoires dans les cinq intervalles énumérés précédemment et nous avons testé notre programme de **my_inverse**. Un exemple de résultat obtenu est présenté dans le tableau ci dessous :

Tableau 2. Résultats du test de my_inverse

a	8	72	850	5285	47378
N	5	37	482	5824	46308
my_inverse(a,N)	2	18	(850, ' et ', 482, ' ne sont pas premiers entre-eux')	(5285, ' et ', 5824, ' ne sont pas premiers entre-eux')	(46308, ' et ', 5824, ' ne sont pas premiers entre-eux')

1c) Afin de déterminer expérimentalement la complexité de nos fonctions **my_pgcd** et **my_inverse**, nous avons généré aléatoirement a,b (respectivement a et N) dans 15 intervalles [1,10], $[10, 10^2]$, ..., $[10^{14}, 10^{15}]$, ensuite nous avons évalué le temps de calcul correspondant. Ci-dessous, un exemple de courbe du calcul du pgcd de a et b en fonction du temps d'exécution avec notre fonction **my_pgcd** (respectivement du calcul de l'inverse de a dans Z/NZ* avec la fonction **my_inverse**).

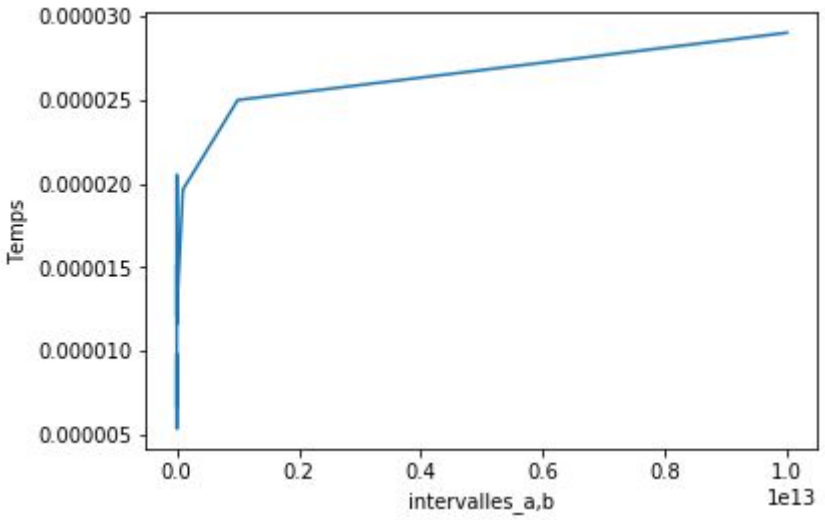

Figure 1: Temps de calcul du pgcd(a, b) en fonction de l'ordre de grandeur de a et b

Interprétation : A première vue, nous remarquons que notre fonction décrit la trajectoire d'une courbe logarithmique, ce qui colle avec la réalité car on démontre que le pgcd pour un algorithme récursif s'effectue le pire des cas le nombre d'appels de l'ordre de $log_2(n)$ (avec |a|,|b| ≤ n) Néanmoins il est important de noter que si on tire par exemple a1,b1 et a2,b2 respectivement dans des intervalles [i1,i2] et [k1,k2] tel que i1<i2<k1<k2 cela n'implique pas forcément que le temps de calcul de **my_pgcd(a1,b1)** soit inférieur au temps de calcul de **my_pgcd(a2,b2)**. Tout dépendra du nombre d'appels de fonction effectuée dans le programme.

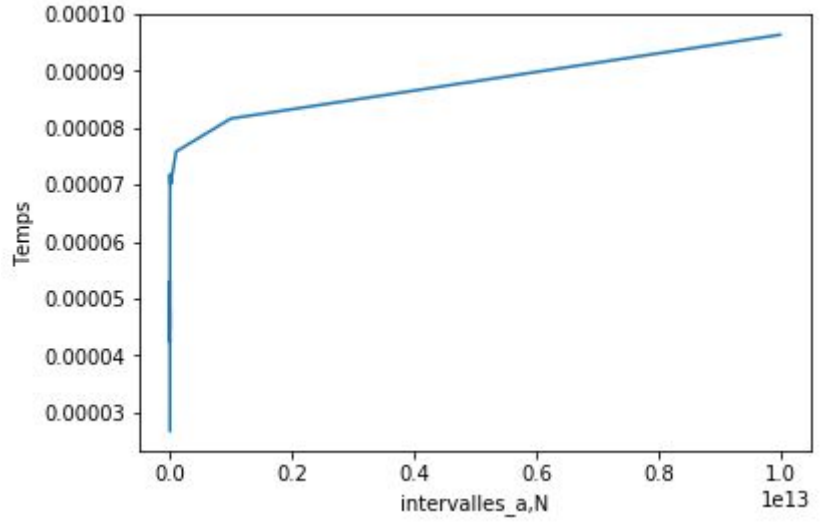

figure 2: Le temps de calcul de my_inverse(a, N) en fonction de l'ordre de grandeur de a et N

Interprétation: A l'instar de la courbe précédente, celle-ci suggère aussi une complexité logarithmique, ce qui semble correspondre à la complexité de notre algorithme.

2) Réponses aux questions de Exercice 2 : Test naïf et recherche de nombres de Carmichael

2b) Complexité de l'algorithme **first_test** :
On fait un parcours des entiers (i) <= $\sqrt{N}$. On fait donc au plus $\sqrt{N}$ test du type n%i==0 (n est divisible par i). On serait donc tenté à affirmer que la complexité de **first_test** est en O(N). Sauf que ce n'est pas le cas car pour chaque i, on effectue une division euclidienne naïve de N par i et on teste le reste. On sait que le module (par la division euclidienne naïve) en binaire est exponentielle en la taille de l'entrée, donc en la taille de N. On en conclut que la complexité de f**irst_test** est en O(2^N), logarithmique en la taille de l'entrée.

2c) Après avoir implémenter une fonction **compt_first** visant à compter les nombres premiers inférieur 10^5 nous avons déterminé qu'il y'en a exactement 9593 .

2d) A l'aide de la fonction **gen_carmichael (N) ,** nous avons généré tous les nombres de Carmichael inférieur à 10^5 a savoir : 561, 1105, 1729, 2465, 2821, 6601, 891, 10585, 15841, 29341, 41041, 46657, 52633, 62745, 63973, 75361.

2f) Notre algorithme qui a une assez bonne vitesse de traitement nous a permis de lister tous les nombres de Carmichael inférieur à 10^5 avec un temps d'environ **14 secondes.**
Afin de tester davantage la robustesse de notre programme, nous avons décidé de tester pour chaque minute le nombre maximal de Carmichael qui serait généré. Le tableau 3 ci dessous nous donnes un résultat au bout de 5 min.

tableau 3 : Evaluation de l'algorithme gen_carmichael sur 5 mins

Temps (en minute)	1	2	3	4	5
Nombre max de carmichael	188461	334153	449065	530881	658801

Ce tableau nous montre que notre fonction qu'au bout de 5 min max nous pouvons générer tous les nombres de carmichael <7* 10^5

2.g) 1. soit n un nombre de carmichael de la forme pqr avec p<q<r . En utilisant le critère de Korselt, montrons qu'il existe un entier h $\in \{2, ..., p-1\}$ tel que pq-1=h(r-1).
on a : n=pqr
comme r divise n c'est a dire que r divise pqr

donc, d'apres le critere de korselt r-1 divise pqr-1 autrement dit il existe k entier naturel tel que pqr-1=k(r-1) ce qui équivaut à pqr - pq + pq - 1= k(r - 1)
c'est à dire pq(r - 1) + pq - 1=k(r - 1) **(1)**
ce qui équivaut à pq - 1 = (k - pq)(r - 1)
prendre donc h=k-pq (avec h>0 car d'après **(1)** pq(r - 1) <= k(r - 1))
d'où pq - 1=h(r - 1)
il nous reste donc a montrer que h $\in \{2, ..., p-1\}$
comme p<q<r par hypothèse alors r-1>q (car q est impair). On a donc: h(r - 1) > hq (car h>0) ce qui équivaut à pq > qh (car h(r - 1) = pq - 1) d'ou p > h
Comme r est premier alors r ≠ pq c'est à dire pq-1 ≠ r-1 et donc h(r - 1) ≠ r-1
d'où h ≠ 1
on peut donc conclure que h $\in \{2, ..., p-1\}$

2. Montrons qu'il existe k entier tel que (hk-p^2)(q-1)=(p+h)(p-1)
puisque q divise pqr - 1, d'apres le critere de Korselt on sait que : q-1 divise pqr - 1 c'est à dire qu'il existe un entier k tel que pr-1=k(q-1) c'est a dire que k(q-1)=pr-p+p+1 ce qui équivaut a k(q-1)=p(r-1)+(p-1)=(p/h)(pq-1)+(p-1) et donc hk(q-1)=p(pq-1)+h(p-1)
nous obtenons hk(q-1)=p^2 (q-1)+p(p-1)+h(p-1)
d'où (hk-p^2)(q-1)=(p+h)(p-1)

3. D après la (1), nous avons h $\in \{2, ..., p-1\}$. Pour p fixé, il n'y a qu'un nombre fini de valeurs h possibles et pour chacune d'elles, il n'y a qu'un nombre fini de diviseurs de (p+h)(p-1) et donc qu'un nombre fini de valeurs (hk-p^2) et pour (q-1) .
Nous en déduisons donc qu'il existe un nombre fini de valeurs possibles pour q.
Et comme pq-1=h(r-1) nous donne l'unique valeur de r possible . Nous pouvons donc conclure qu'il n'existe qu'un nombre fini de nombres de carmichael de la forme pqr avec p<q<r.

2h) Trouvons tous les nombres de carmichael de la forme 3qr et de la forme 5qr avec q et r premiers

- pour p=3 nous avons nécessairement h=2 (d'après 2g-1, h $\in \{2, ..., p-1\}$)
  et q-1 est un diviseur de (p+h)(p-1)=2*5 (d'après 2g-b);
  donc q-1 $\in \{1, 2, 5, 10\}$ et comme q est un nombre premier impair strictement supérieur à p alors q-1=10 et donc q=11. Puisque pq-1=h(r-1), 3*11-1=2*(r-1) ce qui donne r=17
  l'entier n=561=3*11*17 est donc le seul nombre de carmichael de la forme 3qr .
- pour p=5 cet a dire $h \in \{2, 3, 4\}$, pour h=2, en utilisant le même procédé que précédemment, on trouve q=13,r=17. pour h=3 on trouve q=17 et r=29 et pour h=4 on trouve q=29 et r=73
  donc on a 3 nombres de carmichael de la forme 5qr a savoir n1=1105=5*13*17, n2=2465=5*17*29 et n3=10585=5*29*73

3. Réponses aux questions de Exercice 3 : Test de Fermat

3.b) Après avoir implanté le test de Fermat du cours, nous l'avons tester sur différents types de nombres. Notre constat est que ce test de primalité, testé sur des nombres générés par **gen_charmichael** (testé avec des nombres de Carmichael), renvoie toujours PROBABLEMENT PREMIER quelque soit la base utilisée (nous choisissons la base aléatoirement pour chaque test) bien que ces nombres soient composés. Toutefois, testé sur des nombres composés, cet algorithme renvoie toujours COMPOSÉ. Notre conclusion est que le test de Fermat n'est pas fiable. Utiliser donc cette méthode pour générer des nombres premiers peut rendre vulnérable la sécurité.

3.c) La fonction **proba_erreur_fermat** permet de définir la probabilité d'erreur du test de fermat. Elle prend en entrée un entier N et retourne le rapport entre le nombre de fois que le test de fermat se trompe et N. Ainsi la probabilité d'erreur du test de Fermat pour des entiers < 10^5 est : 0.00078

4. Réponses aux questions de Exercice 4 : Test de Rabin et Miller

4.b) Après avoir implanté le test de Miller-Rabin, nous l'avons testé, comme avec Fermat, sur des nombres générés par **gen_charmichael.** Et le constat est que cette fois, le test renvoie bien COMPOSÉ pour ces nombres composés un peu spéciaux (pseudo premiers). Pour des nombres composés, le test renvoie aussi COMPOSÉ. On en conclut que ce test beaucoup plus fiable que celui de Fermat bien qu'il existe une petite probabilité d'erreur.

4.c) De même que dans le test de fermat définit une fonction **proba_erreur_miller_rabin.** Elle prend en entrée un entier N et retourne le rapport entre le nombre de fois que le test de **Miller_Rabin** se trompe et N. La probabilité d'erreur du test de Miller-Rabin pour les nombres inférieurs à 10^5 est approximativement nulle : 0.

III. CHOIX D'IMPLÉMENTATION ET DIFFICULTÉS RENCONTRÉES

Le langage d'implémentation de ce projet était libre. Nous avons décidé de coder en python d'une part car pour nous, c'est le langage le plus adapté pour des calculs mathématiques mais aussi et surtout, car ce langage supporte les très grands nombres, nous n'avons donc pas eu à télécharger des bibliothèques spéciales. Toutefois, nous avons rencontré quelques difficultés, bien qu'elles aient été minimes.

Les deux seuls et principaux soucis que nous avons rencontré lors de l'implémentation de nos codes sont les suivantes :

- Lors de l'implémentation du test de Carmichael, notre fonction **gen_charmichael** qui devait lister les nombres de Carmichael inférieurs à 10^5 mettant plus de ***5 mins*** pour lister ces nombres. Cela était dû au fait que nous faisions de nombreux tests inutiles dans nos boucles. Nous avons enlevé ces tests et depuis, notre fonction est devenue très rapide. Elle trouve les nombres inférieurs à 10^5 en seulement ***14 secondes***.
- Dans l'implémentation du test de Miller-Rabin, nous avons implanté le test exactement comme présenté dans le sujet mais, lors du calcul de la probabilité d'erreur, notre test se trompe avec les nombres 4 et 6 et uniquement avec ces nombres car il renvoie, à chaque exécution, PREMIER alors qu'ils sont composés.. Sinon, il renvoie le bon résultat pour tous les autres. Perplexes, nous sommes dans le doute et ne savons pas si cela est normal (puisqu'on sait que ce sont des tests probabilistes, avec donc des probabilités d'erreurs, et donc ne renvoient pas toujours le bon résultat pour tous les nombres) ou est une erreur de notre part.

CONCLUSION

Il existe de nombreux algorithmes permettant de déterminer la primalité d'un nombre. L'algorithme déterministe se basant sur le crible d'Eratosthène permet ce test avec certitude. Mais lorsque la taille des entrées devient très grands, il n'est plus utilisable vu sa complexité exponentielle. La solution que nous avons utilisées est donc la randomisation, avec deux algorithmes probabilistes, beaucoup plus efficace en temps de calcul que la naïf mais avec des probabilités d'erreurs. Nous pouvons toutefois démontrer qu'avec certains de ces tests un peu plus sûr (comme celle de Miller-Rabin), cette probabilité est faible.