ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΣΥΣΤΗΜΑΤΩΝ ΜΕΤΑΔΟΣΗΣ ΠΛΗΡΟΦΟΡΙΑΣ
ΚΑΙ ΤΕΧΝΟΛΟΓΙΑΣ ΥΛΙΚΩΝ

Ανάλυση και Υλοποίηση Σύγχρονων Αλγορίθμων Εντοπισμού Πρώτων Αριθμών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Κωνσταντίνος Α. Τερτίκας

Επιβλέπων : Κωνσταντίνος Παπαοδυσσεύς
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2016

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΣΥΣΤΗΜΑΤΩΝ ΜΕΤΑΔΟΣΗΣ ΠΛΗΡΟΦΟΡΙΑΣ
ΚΑΙ ΤΕΧΝΟΛΟΓΙΑΣ ΥΛΙΚΩΝ

Ανάλυση και Υλοποίηση Σύγχρονων Αλγορίθμων Εντοπισμού Πρώτων Αριθμών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Κωνσταντίνος Α. Τερτίκας

Επιβλέπων : Κωνσταντίνος Παπαοδυσσεύς
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 13η Ιουλίου 2016.

.....	.....	.....
Κωνσταντίνος Παπαοδυσσεύς Καθηγητής Ε.Μ.Π.	Ηλίας Κουκούτσης Επίκουρος Καθηγητής Ε.Μ.Π.	Μαρία - Παρασκευή Ιωαννίδου Καθηγήτρια Ε.Μ.Π.

Αθήνα, Ιούλιος 2016

.....

Κωνσταντίνος Α. Τερτίκας

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

ΠΕΡΙΛΗΨΗ

Η παρούσα διπλωματική εργασία μελετά σύγχρονους υπολογιστικούς αλγορίθμους εύρεσης πρώτων αριθμών. Συγκεκριμένα, ασχολείται με δύο αλγορίθμους, έναν ντετερμινιστικό πολυωνυμικού λογαριθμικού χρόνου, τον Agrawal–Kayal–Saxena (AKS), και έναν πιθανοθεωρητικό αλγόριθμο ευρετικού χρόνου, τον Elliptic Curve Primality Proving (ECPP). Ο αλγόριθμος AKS είναι ο πρώτος αλγόριθμος πολυωνυμικού λογαριθμικού χρόνου και ταυτόχρονα ντετερμινιστικός που έχει αναπτυχθεί, ενώ ο ECPP είναι ένας γρήγορος και ευρέως διαδεδομένος στατιστικός αλγόριθμος αναζήτησης.

Με χρήση της γλώσσας C, και με τη βοήθεια δύο βιβλιοθηκών πολλαπλής ακρίβειας υπολογισμών, τη βιβλιοθήκη GMP και τη βιβλιοθήκη MPFR, υλοποιούμε τον αλγόριθμο AKS. Η υλοποίηση αυτή μας δείχνει ότι ο αλγόριθμος AKS είναι ακριβής, αλλά ταυτόχρονα και αρκετά αργός σε σύγκριση με τους ήδη υπάρχοντες αλγορίθμους για αναζήτηση μικρών πρώτων ($n < O(10^{19})$).

Η εργασία δομείται σε τέσσερα κεφάλαια. Στο πρώτο κεφάλαιο γίνεται εισαγωγή στο πρόβλημα εύρεσης πρώτων αριθμών, μέσα από μία ιστορική ανασκόπηση. Στο δεύτερο κεφάλαιο, παρουσιάζονται οι δύο αλγόριθμοι που μελετά η εργασία. Στο τρίτο κεφάλαιο, περιγράφεται η υλοποίηση των δύο αλγορίθμων. Τέλος, στο τελευταίο κεφάλαιο, γίνεται συζήτηση όσον αφορά τις δυνατότητες και τους περιορισμούς του αλγορίθμου AKS.

Λέξεις Κλειδιά: Πρώτοι αριθμοί, αλγόριθμος AKS, αλγόριθμος ECPP, εύρεση πρώτων αριθμών.

ABSTRACT

In this diploma thesis, we study modern computational primality testing algorithms. In particular, our main focus is on two algorithms, one deterministic polynomial logarithmic time algorithm, the Agrawal–Kayal–Saxena (AKS), and one probabilistic algorithm which runs heuristically, the Elliptic Curve Primality Proving (ECPP). The AKS algorithm is the first polynomial logarithmic time deterministic algorithm that has been implemented, while the ECPP algorithm is a fast and widely used statistic algorithm.

We implement the AKS algorithm with the use of the programming language C, and with the help of two muliple precision libraries, the GMP library, and the MPFR library. Our implementation shows that tha AKS algorithm is accurate, but also quite slow in comparison to other existing primality testing algorithms for small numbers (n < O(10*19)).

This diploma thesis is divided into four main sections. In the first chapter, an introduction to the problem of finding prime numbers is made, mainly through a historic literature review. In the second chapter, we present the two algorithms that this dissertation examines. In the third chapter, we explain the implementations of the two afore mentioned algorithms. Finally, in the last chapter, we discuss about the capabilities and the limitations of the AKS algorithm.

Key Words: Prime Numbers, AKS algorithm, ECPP algorithm, primality testing.

ΠΡΟΛΟΓΟΣ-ΕΥΧΑΡΙΣΤΙΕΣ

Στο πλαίσιο της παρούσας διπλωματικής εργασίας, θέλω να εκφράσω της θερμές μου ευχαριστίες στον επιβλέποντα καθηγητή κ. Παπαοδυσσέα Κωνσταντίνο για την ευκαιρία που μου έδωσε να ασχοληθώ με ένα τόσο ενδιαφέρον θέμα. Τον ευχαριστώ για την πάντα πρόθυμη βοήθειά του αλλά και την άριστη συνεργασία μας. Επίσης θέλω να ευχαριστήσω ιδιαιτέρως την οικογένειά μου και το φιλικό μου περιβάλλον, καθώς στάθηκαν δίπλα μου, με στήριξαν, και συνέβαλαν τα μέγιστα τόσο στην ολοκλήρωση της εργασίας όσο και καθ' όλη τη διάρκεια των σπουδών μου.

ΠΕΡΙΕΧΟΜΕΝΑ

ΚΕΦΑΛΑΙΟ 1: Εισαγωγή

1.1 Πρώτοι Αριθμοί: Ορισμός και η Σημασία τους

Οι πρώτοι αριθμοί αποτελούν μια από τις βασικές έννοιες της Θεωρίας Αριθμών. Πρώτος αριθμός είναι ένας θετικός ακέραιος ο οποίος έχει ακριβώς δύο θετικούς διαιρέτες, τον αριθμό 1 και τον εαυτό του. Οι θετικοί ακέραιοι που είναι μεγαλύτεροι από τη μονάδα και δεν είναι πρώτοι ονομάζονται σύνθετοι. Ο αριθμός 1 δεν θεωρείται ούτε πρώτος ούτε σύνθετος.

Οι πρώτοι αριθμοί έχουν απασχολήσει και απασχολούν όχι μόνο τους μαθηματικούς, αλλά και ερευνητές άλλων πεδίων όπως της πληροφορικής και της κρυπτογραφίας. Στα μαθηματικά πολλοί ερευνητές έχουν μελετήσει τη φύση και τις ιδιότητες των αριθμών αυτών ενώ μέχρι και σήμερα υπάρχουν προβλήματα τα οποία στη διατύπωση τους φαίνονται απλά αλλά παραμένουν άλυτα (π.χ η εικασία του Goldbach ότι κάθε άρτιος ακέραιος μεγαλύτερος του 2 μπορεί να γραφεί ως άθροισμα δύο πρώτων αριθμών). Στην πληροφορική η έρευνα κατευθύνεται σε υπολογιστικές διαδικασίες που στοχεύουν στην εύρεση και τον χαρακτηρισμό πρώτων αριθμών μέσα από ανάπτυξη υπολογιστικών αλγορίθμων. Στην κρυπτογραφία οι πρώτοι αριθμοί έχουν κυρίαρχη σημασία καθώς χρησιμοποιούνται για την κωδικοποίηση και αποκωδικοποίηση μηνυμάτων. Πολλά κρυπτογραφικά συστήματα εξαρτώνται από έρευνες που γίνονται στους πρώτους αριθμούς και σε σχετικά προβλήματα της Θεωρίας Αριθμών, καθώς και από τις εξελίξεις στον τεχνολογικό τομέα και στους αλγορίθμους [1]. Η ανάπτυξη της κρυπτογραφίας φαίνεται ότι εξαρτάται άμεσα από τη μελέτη των πρώτων αριθμών, τόσο στα μαθηματικά όσο και στην πληροφορική.

Η μελέτη των πρώτων αριθμών εστιάζει κυρίως σε θέματα που αφορούν το πλήθος τους, την κατανομή τους, τον έλεγχο του αν ένας αριθμός

είναι πρώτος (primality testing), και την εύρεση ειδικών μορφών πρώτων αριθμών. Συγκεκριμένα, γνωρίζουμε από την εποχή του Ευκλείδη ότι οι πρώτοι αριθμοί είναι άπειροι ενώ αργότερα οι Gauss, Legendre και Riemann χρησιμοποιούν μεθόδους της ανάλυσης για να αποδείξουν ότι όσο οι αριθμοί μεγαλώνουν, η πυκνότητα των πρώτων αριθμών μικραίνει [2] (σελίδα 133).

Οι αλγόριθμοι εύρεσης πρώτων αριθμών και η βελτίωσή τους, ώστε να υπολογίζουν ιδιαίτερα μεγάλους αριθμούς, απασχολεί και την πληροφορική για πολλά χρόνια. Αρχικά η μελέτη εάν ένας φυσικός αριθμός n είναι πρώτος βασίστηκε στον έλεγχο εάν οι μικρότεροι από αυτόν φυσικοί είναι ή όχι διαιρέτες του. Στη συνέχεια, ο έλεγχος αυτός περιορίστηκε στους $\sqrt{n}$ μικρότερους αριθμούς, μειώνοντας έτσι τον αριθμό των δοκιμών. Στην ιδέα αυτή στηρίζεται και το κόσκινο του Ερατοσθένη, το οποίο δουλεύει διαγράφοντας όλους τους σύνθετους αριθμούς σε ένα σύνολο διαδοχικών φυσικών αριθμών από το 1 μέχρι το n. Οι αλγόριθμοι αυτοί είναι απλοί αλλά λειτουργούν καλά για σχετικά μικρούς αριθμούς, ενώ είναι πρακτικά αδύνατον να εφαρμοστούν και να υπάρχει αποτέλεσμα σε πεπερασμένο χρόνο για μεγάλους αριθμούς. Αυτό οδήγησε στην αναζήτηση νέων αλγορίθμων που είτε μας οδηγούν σε μια σίγουρη απάντηση για το αν ο αριθμός είναι πρώτος ή όχι (ντετερμινιστικός αλγόριθμος), είτε μας λένε ότι μπορεί να είναι πρώτος με κάποιο βαθμό πιθανότητας (πιθανοθεωρητικός αλγόριθμος) [2]. Διάφοροι αλγόριθμοι έχουν αναπτυχθεί που ελέγχουν μεγάλους αριθμούς αν είναι πρώτοι και διαφέρουν ως προς την υπολογιστική τους δυνατότητα. Στην παρούσα διπλωματική θα υλοποιηθεί σε γλώσσα προγραμματισμού C ένας σύγχρονος ντετερμινιστικός αλγόριθμος, ο AKS [4] και θα αναλυθεί ένας πιθανοθεωρητικός αλγόριθμος ο Elliptic Curve Primality Proving (ECPP) [3].

1.2 Ιστορική Εξέλιξη των Μεθόδων Εύρεσης Πρώτων Αριθμών

1.2.1 Η μαθηματική οπτική

Η ιστορία των πρώτων αριθμών ξεκινά από σχεδόν 23 αιώνες πριν, όταν ο Ευκλείδης απέδειξε ότι οι πρώτοι αριθμοί είναι άπειροι. Η διατύπωση του θεωρήματός του παρουσιάζεται στο 9ο βιβλίο των στοιχείων του Ευκλείδη. Στην πραγματικότητα το θεώρημα το διατύπωσε ως εξής « Οι πρώτοι αριθμοί είναι περισσότεροι από κάθε δεδομένο σύνολο πρώτων αριθμών», και το απέδειξε για δεδομένο σύνολο πρώτων αριθμών, πλήθους τρία [5]. Ο Ευκλείδης επίσης διατύπωσε το Θεμελειώδες θεώρημα της Αριθμητικής, δείχνοντας ότι οι πρώτοι αριθμοί αποτελούν τους δομικούς λίθους με τους οποίους κατασκευάζονται όλοι οι αριθμοί, καθώς κάθε φυσικός αριθμός γράφεται κατά μοναδικό τρόπο ως γινόμενο πρώτων αριθμών. Η συνεισφορά αυτή του Ευκλείδη ήταν πολύ σημαντική για τη θεμελίωση των πρώτων αριθμών. Διαφορετικές αποδείξεις ακολούθησαν αυτές του Ευκλείδη, οι οποίες απασχόλησαν τους μαθηματικούς πολύ αργότερα. Για παράδειγμα, ο Euler έδωσε το 1737 την πρώτη του απόδειξη για την απειρία των πρώτων αριθμών, και μετά ακολούθησαν και άλλες από τον ίδιο και από άλλους μαθηματικούς [5].

Όμως, η απειρία των πρώτων αριθμών δεν έλυνε το πρόβλημα για το ποιοι είναι οι πρώτοι αριθμοί. Ήδη από την εποχή των Αρχαίων Ελλήνων κατασκευάζονταν πίνακες που διαχώριζαν δεδομένους φυσικούς αριθμούς σε πρώτους και σύνθετους [6]. Ο Ερατοσθένης ήταν ο πρώτος μαθηματικός που βρήκε τρόπο κατασκευής πρώτων αριθμών, το γνωστό κόσκινο του Ερατοσθένη. Ο αλγόριθμος αυτός στηρίζεται στη διαγραφή πολλαπλασίων πρώτων αριθμών. Για παράδειγμα φτιάχνουμε μια λίστα από διαδοχικούς ακεραίους από το 2 μέχρι κάποιον αριθμό n. Ξεκινώντας από τον 2 ως πρώτο αριθμό (p) διαγράφουμε όλα τα πολλαπλάσια του που είναι μικρότερα ή ίσα με το n. Ο επόμενος αριθμός

μετά το 2 που δεν έχει διαγραφεί είναι ο 3 και είναι ο επόμενος πρώτος. Στη συνέχεια διαγράφουμε όλα τα πολλαπλάσια του 3. Ο επόμενος πρώτος αριθμός p είναι αυτός μετά τον 3 που δεν έχει διαγραφεί. Συνεχίζουμε τη διαδικασία μέχρι το p^2 να είναι μεγαλύτερο από το n. Όλοι οι αριθμοί που έχουν απομείνει είναι πρώτοι αριθμοί. Να σημειώσουμε εδώ ότι η χρονική πολυπλοκότητα του κόσκινου του Ερατοσθένη είναι O(log(log(n))). Η κατασκευή πινάκων πρώτων αριθμών για μεγάλο πλήθος αριθμών συνεχίστηκε στο τέλος του 18ου και το 19ο αιώνα όπου υπολογίστηκαν πίνακες πρώτων μέχρι 400.000 (Marci), 2.856.00 (Felkel), 3.036.000 (Burckhardt), 10.006.721 (Lehmer) ([6]).

Ο Dickson [7] αναφέρει προσπάθειες εύρεσης πρώτων αριθμών που έγιναν από μεγάλους μαθηματικούς της εποχής του Euler και που αργότερα οδήγησαν στον υπολογισμό μεγάλου αριθμού πρώτων αριθμών. Για παράδειγμα, από την εποχή του Euler αναζητήθηκαν τύποι πολυωνυμικών εκφράσεων που θα υπολόγιζαν πρώτους αριθμούς. Αυτές οι συναρτήσεις, αν και έδιναν πρώτους αριθμούς για διάφορες τιμές της ανεξάρτητης μεταβλητής, δεν μπορούσαν να δώσουν για όλους τους ακεραίους. Για παράδειγμα, ο Euler διαπίστωσε ότι η έκφραση x^2-x+41 δίνει πρώτους αριθμούς για x=1,...40, η έκφραση x^2+x+17 για x=0,...15 και όχι για 16 και ότι η x^2+x+41 δίνει πρώτους αριθμούς για x=1,...15. Ανάλογα ο Legendre διαπιστώνει ότι η εκφραση x^2+x+41 δίνει πρώτους για x=0,...39 και η έκφραση $2x^2$+29 δίνει πρώτους για x=0,...28. Ακολούθησαν διερευνήσεις για πολυώνυμα τρίτου βαθμού και για αρνητικούς ακεραίους (π.χ. η έκφραση x^3+x^2+17 για x=-14, -13,...+10) καθώς και για μεγαλύτερες τιμές του x (π.χ. η έκφραση x^2 – 2999x + 2248541, για $1460 \le x \le 1539$).

Στην πορεία αυτή αναζήτησης πρώτων αριθμών διατυπώθηκαν εικασίες που είτε απορρίφθηκαν στη συνέχεια από άλλους μαθηματικούς, ή δεν αποδείχθηκαν ενώ φαίνεται να ισχύουν. Μια τέτοια εικασία είναι η εικασία του Goldbach και παραλλαγές της που αφορούν στο αν ένας ακέραιος μπορεί να γραφτεί ως άθροισμα πρώτων αριθμών. Ο Dickson [7] αναφέρει ότι ο Goldbach υποστήριξε ότι αν ένας αριθμός μπορεί να γραφτεί ως άθροισμα δύο πρώτων αριθμών, τότε μπορεί να γραφτεί ως

άθροισμα όσων πρώτων θέλουμε (μέχρι N), συμπεριλαμβανομένης και της μονάδας, και ότι κάθε αριθμός μεγαλύτερος του δύο είναι άθροισμα τριών πρώτων αριθμών. Η εικασία αυτή βασίστηκε σε μια άλλη εικασία του γνωστή ως η δυαδική (ισχυρή). Αυτή ισχυρίζεται ότι κάθε άρτιος αριθμός είναι το άθροισμα δύο πρώτων αριθμών, κάτι το οποίο δεν έχει αποδειχθεί έως σήμερα. Γύρω από αυτή την εικασία διάφοροι μαθηματικοί όρισαν καινούριες εικασίες και διαπίστωσαν σχέσεις ανάμεσα σε τυχαίους αριθμούς και σε πρώτους.

Μια άλλη σημαντική περιοχή είναι η ανάπτυξη τρόπων ελέγχου αν ένας αριθμός είναι πρώτος ή όχι πέρα από τον ορισμό. ([6], [7]) Για παράδειγμα, πολλοί γνωστοί μαθηματικοί όπως οι Leibniz, Lagrange, Lebesque διαπίστωσαν, όταν συζητούσαν το αντίστροφο του θεωρήματος του Fermat ως ένα τεστ για το πότε ένας αριθμός είναι πρώτος, ότι ο αριθμός n είναι πρώτος αν και μόνο αν διαιρεί τον 1+(n-1)!. Πολλά άλλα τεστ ακολούθησαν αυτό τον έλεγχο όπως για παράδειγμα ότι ένας περιττός αριθμός A είναι πρώτος αν και μόνο αν ο $A+\kappa^2$ δεν είναι τετράγωνος αριθμός για κ=1, 2,...(A-3)/2 (Montferrier).

Επιπλέον αριθμοί διαφορετικών μορφών θεωρήθηκαν ως πρώτοι. Αριθμοί της μορφής $2^n - 1$ μελετήθηκαν από το 15ο αιώνα και διατυπώθηκαν αρχικά εικασίες οι οποίες στη συνέχεια απορρίφθηκαν ότι ήταν πρώτοι. Το 1644 ο Mersenne ανακοίνωσε ότι οι αριθμοί της μορφής $2^n - 1$ είναι πρώτοι για n 2, 3, 5, 13, 19, 31, 67, 127 και 257. Παρόλο που αργότερα αποδείχθηκε ότι δεν ήταν σωστός ο ισχυρισμός του για n 61, 67, 89, 109 και 257 οι αριθμοί Mersenne ήταν μεγάλοι πρώτοι αριθμοί. Ανάλογης μορφής αριθμοί δώθηκαν από τον Fermat που θεωρούσε ότι αριθμοί της μορφής $F_n=2^{2^n}+1$ όπου n είναι πρώτοι αριθμοί, γεγονός που απορρίφθηκε από τον Euler το 1748 που ανακάλυψε ότι ο $F_5 = 2^{32}+1 = 641.6700.417$ δεν είναι πρώτος. Η προσπάθεια να ελέγξουν διάφορους αριθμούς Fermat αν είναι ή όχι πρώτοι αριθμοί δεν ήταν εύκολη διαδικασία και απασχόλησε πολλούς μαθηματικούς μέχρι τις αρχές του 20ου αιώνα.

Μια άλλη προσέγγιση που συνδέεται με την εύρεση πρώτων αριθμών ήταν ο έλεγχος αρχικά ύπαρξης πρώτων αριθμών ανάμεσα σε κάποιους

ακεραίους. Για παράδειγμα, ο Bertrand επαλήθευσε ότι για αριθμούς μικρότερους από 6.000.000 ισχύει η πρόταση ότι για κάθε ακέραιο n > 6 υπάρχει τουλάχιστον ένας πρώτος αριθμός ανάμεσα στο n-2 και στο n/2. Την πρόταση αυτή ακολούθησαν μια σειρά από άλλες προτάσεις που τεκμηρίωναν την ύπαρξη πρώτων αριθμών ανάμεσα σε διαφορετικές μορφές αριθμών. Η ύπαρξη πρώτων σε διαφορετικά διαστήματα των ακεραίων, συνδέεται με τη μελέτη της κατανομής των πρώτων αριθμών. Ο Ingham [8] περιγράφει τα βασικά θεωρήματα γύρω από την κατανομή των πρώτων στο σύνολο των φυσικών αριθμών και αναφέρει ότι «η γενική κατανομή των πρώτων αριθμών έχει συγκεκριμένα χαρακτηριστικά κανονικότητας η οποία μπορεί να περιγραφεί με ακριβείς όρους και έγινε το αντικείμενο μαθηματικής διερεύνησης» (σελίδα 2). Παρατηρώντας πίνακες πρώτων αριθμών βλέπουμε ότι το πλήθος τους όλο και μεγαλώνει και όσο προχωράμε το πλήθος αυξάνεται. Η πρώτη παρατήρηση οδηγεί στο θεώρημα που απέδειξε ο Ευκλείδης για την απειρία των πρώτων αριθμών που έχουμε ήδη συζητήσει παραπάνω. Το ότι το πλήθος αυξάνεται όσο μεγαλώνουν οι αριθμοί εκφράζεται από πολλά θεωρήματα που διατυπώθηκαν και αποδείχθηκαν. Για παράδειγμα, ο Legendre υπέθεσε ότι για μεγάλους αριθμούς x τότε των πλήθος των πρώτων που δεν ξεπερνούν το δίνεται από τη σχέση $\frac{x}{\log x - B}$ όπου B είναι μια αριθμητική σταθερά. Οι Gauss, Hadamard, Littlewood είναι άλλοι μαθηματικοί που μελέτησαν το θέμα της κατανομής των πρώτων μέσα από την σύνδεση της ανάλυσης με την πιο κλασσική θεωρία αριθμών.

1.2.2 Η οπτική της πληροφορικής

Αν και όπως έχουμε ήδη αναφέρει η μαθηματική οπτική και η οπτική της πληροφορικής δεν είναι τόσο ανεξάρτητες, στην πραγματικότητα η βασικότερη εστίαση στην πληροφορική δεν είναι η απόδειξη των θεωρημάτων γύρω από τους πρώτους αριθμούς αλλά η ανάπτυξη και υλοποίηση ελέγχων για το αν ένας αριθμός είναι πρώτος ή όχι με αλγορίθμους που μπορούν να υλοποιηθούν με τα τεχνολογικά μέσα που διατίθενται στην κάθε εποχή.

Εξέλιξη βλέπουμε και στο βασικό υπολογιστικό πρόβλημα που είναι η ανάπτυξη αλγορίθμων που ελέγχουν αν ένας μεγάλος αριθμός είναι πρώτος και υπολογίζουν όλο και μεγαλύτερες ακολουθίες πρώτων αριθμών. Στο βιβλίο των Crandall και Pomerance [1] παρουσιάζεται πολύ συστηματικά η εξέλιξη των υπολογιστικών μεθόδων. Οι συγγραφείς αναφέρουν ότι οι αριθμοί με τους οποίους ασχολούμαστε είναι «μικροί» με την έννοια ότι μπορούμε να βρούμε πάντοτε πεπερασμένο αριθμό αριθμών μικρότερων από τον συγκεκριμένο και άπειρο πλήθος μεγαλύτερων. Υποστηρίζουν ότι τα τελευταία 30 χρόνια το πλήθος των ψηφίων των αριθμών που μπορούμε να αναλύσουμε σε γινόμενο πρώτων παραγόντων είναι οκτώ φορές μεγαλύτερο από ότι παλαιότερα, και ότι το πλήθος των ψηφίων των αριθμών που μπορούμε να αποδείξουμε ότι είναι πρώτοι είναι περίπου 500 φορές μεγαλύτερο. Η εξέλιξη αυτή οφείλεται στην πρόοδο της τεχνολογίας και στην πρόοδο στην ανάπτυξη αλγορίθμων. Για παράδειγμα, αυτό που ξέρουμε σήμερα είναι ότι αριθμοί με 170 ψηφία μπορούν να αναλυθούν σε γινόμενο πρώτων παραγόντων, ενώ περίπου 15000 ψηφία είναι το όριο των ψηφίων τυχαίων πρώτων αριθμών για τους οποίους έχει αποδειχθεί ότι είναι πρώτοι. Ο μεγαλύτερος αριθμός που έχει αποδειχθεί ότι είναι πρώτος είναι ο Mersenne αριθμός $2^{74207281}-1$ [9] . Ο υπολογισμός τέτοιων αριθμών και η απόδειξη του ότι είναι πρώτοι με την εξέλιξη της τεχνολογίας απαιτεί ελάχιστο χρόνο σε σύγκριση με το χρόνο που χρειάζονταν πριν μια δεκαετία. Πέρα από τους αριθμούς Mersenne που έχουν αναπτυχθεί

γρήγορα τεστ (π.χ των Lucas-Lehmer), το πρόβλημα του υπολογισμού πρώτων αριθμών μη συγκεκριμένης μορφής, δηλαδή τυχαίων πρώτων αριθμών, είναι πιο δύσκολο. Ανάμεσα στις σύγχρονες προσπάθειες υπολογισμού πρώτων αριθμών τυχαίας μορφής, έχουμε τη δουλειά του Morain που εφάρμοσε ιδέες του Atkin και άλλων για να αναπτύξουν μια γρήγορη μέθοδο ελέγχου πρώτων αριθμών την ECPP (efficient elliptic curve primality proving), η οποία θα παρουσιαστεί στην εργασία αυτή, για να αποδείξει το 1998 ότι ο αριθμός $(2^{7331}-1)/4580728443161$ που είχε 2196 δεκαδικά ψηφία είναι πρώτος. Τον Ιούλιο του 2004 οι Franke, Kleinjung, Morain, and Wirth απόδειξαν ότι ο αριθμός Leyland number $4405^{2638} + 2638^{4405}$, που έχει 15071 ψηφία είναι πρώτος αριθμός. Από τότε εμφανίζονται πρώτοι αριθμοί που έχουν ειδικές μορφές (π.χ Sophie Germain πρώτοι αριθμοί, δίδυμοι πρώτοι αριθμοί) και ανακαλύπτονται συνέχεια καινούργιοι πρώτοι αριθμοί.
Αυτό επίσης που έχει εξελιχθεί είναι ο χρόνος ο οποίος απαιτείται για τον υπολογισμό τέτοιων μεγάλων πρώτων αριθμών. Για παράδειγμα, για τον υπολογισμό ενός μεγάλου Mersenne αριθμού το 2004 χρειαζόταν περίπου μια εβδομάδα υπολογιστικού χρόνου ενώ μια δεκαετία πριν αυτό θα απαιτούσε δέκα χρόνια υπολογιστικού χρόνου. Η εξέλιξη αυτή οφείλεται τόσο στην ανάπτυξη πιο γρήγορων αλγορίθμων όσο και στην ραγδαία ανάπτυξη της τεχνολογίας.

Υπολογιστικά χαρακτηριστικά των Τεστ ελέγχου

Το πιο απλό τεστ διαιρετότητας που μπορεί να επιλέξει κάποιος για να ελέγξει αν ένας αριθμός n είναι πρώτος ή σύνθετος είναι να τον διαιρέσει με όλους τους πρώτους αριθμούς που είναι μικρότεροι του. Αυτός ο έλεγχος μπορεί να περιοριστεί για τους πρώτους που είναι μικρότεροι από την τετραγωνική ρίζα του n. Οι Crandall & Pomerance στο [1] γράφουν τον απλό αλγόριθμο της ***διαίρεσης δοκιμής (trial division***) (σελ.119) ο οποίος προσδιορίζει το σύνολο F των πρώτων αριθμών που διαιρούν το N και είναι μικρότεροι από την τετραγωνική

ρίζα του n. Αν το σύνολο F είναι το κενό σύνολο τότε ο αριθμός n είναι πρώτος.

Παραθέτουμε παρακάτω τον αλγόριθμο αυτό όπως αναφέρεται στο [1]:

```
1. [Διαίρεση κατά 2]
        F = { };     // The empty multiset.
        N = n;
        while(2|N) {
              N = N/ 2;
              F = F ∪{2};
        }
2. [Κύρια επανάληψη διαίρεσης]
        d = 3;
        while(d2 ≤ N) {
              while(d| N) {
                    N = N/d;
                    F = F ∪{d};
              }
              d = d + 2;
        }
        if( N == 1) return F;
        return F ∪{ N};
```

Στην ιδέα αυτή στηρίχθηκε και το κόσκινο του Ερατοσθένη που έχουμε παρουσιάσει παραπάνω. Το ***κόσκινο του Ερατοσθένη*** ελέγχει τους αριθμούς από 2 μέχρι n-1 αν είναι πρώτοι ή σύνθετοι φτιάχνοντας έναν πίνακα nxn με αρχικές θέσεις μονάδα. Ξεκινώντας από ένα πρώτο αριθμό (π.χ 2, 3, p), μηδενίζει κανείς όλα τα πολλαπλάσια του. Οι θέσεις του πίνακα που θα έχουν στο τέλος παραμείνει μονάδες θα αντιστοιχούν στους πρώτους αριθμούς. Οι αριθμητικές πράξεις που γίνονται είναι προσθέσεις και ο αριθμός των βημάτων που απαιτούνται για αριθμούς μέχρι το N είναι ανάλογος του log (log N). Μπορεί ο αριθμός log log n να τείνει στο άπειρο αλλά πηγαίνει πολύ αργά. Ισχύει ότι log (log n)< 10 για κάθε $N \leq 10^{9565}$. Το βασικό μειονέκτημα του κόσκινου του Ερατοσθένη είναι ότι καταλαμβάνει τεράστιο χώρο. Αν και αυτό αντιμετωπίζεται με

διάσπαση του πίνακα δημιουργούνται άλλα προβλήματα που σχετίζονται με τον υπολογιστικό χρόνο.

Διάφορες παραλλαγές του κόσκινου του Ερατοσθένη αναπτύσσονται οι οποίες οδηγούν σε μικρότερο υπολογιστικό χρόνο καθώς και στον υπολογισμό των γινομένων των πρώτων αριθμών δηλαδή στην ανάλυση φυσικών αριθμών σε γινόμενο πρώτων παραγόντων.

Ο υπολογισμός ψευδοπρώτων αριθμών και το μκρό θεώρημα του Fermat αποτελούν τη βάση πολλών αλγορίθμων καθώς και του AKS αλγορίθμου που υλοποιούμε στην παρούσα διπλωματική. Η έννοια του ψευδοπρώτου αριθμού όπως περιγράφεται στο [1] στηρίζεται στον παρακάτω ισχυρισμό: Αν υπάρχει κάποιο θεώρημα της μορφής «Αν ο n είναι πρώτος τότε η αριθμητική πρόταση S που αφορά τον αριθμό n είναι αληθής», τότε μπορούμε να ξέρουμε ότι ίσως ο αριθμός n είναι πρώτος, ενώ αν το θεώρημα δεν ισχύει, είμαστε βέβαιοι ότι ο αριθμός n είναι σύνθετος. Για παράδειγμα, αν έχουμε ένα μεγάλο αριθμό n και θέλουμε να δούμε αν είναι πρώτος ή σύνθετος, ίσως μπορούμε να ελέγξουμε αν η αριθμητική πρόταση είναι αληθής. Αν η πρόταση δεν ισχύει τότε έχουμε αποδείξει ότι ο n είναι σύνθετος. Αν η πρόταση είναι αληθής τότε ίσως ο n είναι πρώτος αλλά ίσως είναι και σύνθετος αριθμός. Έτσι έχουμε την έννοια του S-ψευδοπρώτου, ο οποίος είναι ένας σύνθετος ακέραιος για τον οποίο η πρόταση S ισχύει. Έτσι ξέρουμε ότι το θεώρημα «Αν ο n είναι πρώτος τότε ο n είναι ίσος με 2 ή ο n είναι περιττός» Αυτή η αριθμητική πρόταση ελέγχεται εύκολα για κάθε αριθμό n. Όμως αυτό δεν σημαίνει ότι είμαστε σίγουροι για το ότι ο n είναι πρώτος, καθώς υπάρχουν περισσότεροι ψευδοπρώτοι γύρω από αυτό το τεστ από ότι πραγματικοί πρώτοι. Έτσι η έννοια του ψευδοπρώτου είναι χρήσιμη όταν υπάρχουν λίγοι τέτοιοι αριθμοί.

Το μικρό θεώρημα του Fermat χρησιμοποιεί το γεγονός ότι οι αριθμοί της μορφής $\alpha^{\beta} \pmod{n}$ μπορούν να υπολογιστούν με γρήγορους αλγορίθμους. Το θεώρημα λέει ότι “Αν n είναι πρώτος αριθμός τότε για κάθε ακέραιο α ισχύει η σχέση $\alpha^{n} = \alpha \pmod{n}$”. Αν η πρόταση ισχύει τότε λέμε ότι ο n είναι ψευδοπρώτος. Για παράδειγμα, ο n = 91 είναι ένας ψευδοπρώτος βάσης 3 γιατί ο 91 είναι σύνθετος και $3^{91} = 3 \pmod{91}$. Οι

ψευδοπρώτοι Fermat αριθμοί αποδεικνύεται ότι είναι σπάνιοι αν τους συγκρίνει κανείς με το πλήθος των πρώτων αριθμών. Ακολουθεί ο αλγόριθμος που υλοποιεί το μικρό θεώρημα του Fermat και ελέγχει αν είνας αριθμός είναι «πιθανόν» πρώτος:

Μας δίνεται ένας ακέραιος n > 3 και ένας ακέραιος a με $2 \leq a \leq n - 2$. Ο αλγόριθμος μας δίνει ως αποτέλεσμα είτε «ο n είναι ένας πιθανόν πρώτος αριθμός βάσης α» ή « ο n είναι σύνθετος αριθμός)

1. *[Υπολογισμός δύναμης]*
   b = a^{n-1} mod n;
2. *[Έλεγχος]*
   if (b == 1) return “n is a probable prime base a”;
   return “n is composite”;

Υπάρχουν ψευδοπρώτοι αριθμοί ως προς διαφορετικές βάσεις και μάλιστα αριθμοί που είναι ψευδοπρώτοι ως προς άπειρες βάσεις (π.χ ο αριθμός 341). Αυτοί οι αριθμοί είναι άπειροι και έτσι δημιουργούν πρόβλημα ώς προς τη σημασία τους για την εύρεση πρώτων αριθμών. Για λύση του προβλήματος αυτού αναπτύχθηκαν διάφορες παραλλαγές του μικρού θεωρήματος του Fermat και έτσι αναπτύχθηκαν πολλοί πιθανοθεωρητικοί αλγόριθμοι.

Θα αναφέρουμε σύντομα παρακάτω κάποια βασικά θεωρήματα που έχουν χρησιμοποιηθεί για τον έλεγχο ενός αριθμού αν είναι πρώτος μέσα από ντετερμινιστικούς αλγορίθμους. Ένα βασικός έλεγχος είναι ο ***n-1 έλεγχος*** ο οποίος λέει ότι αντί να ελέγχουμε τους διαιρέτες του n ελέγχουμε τους διαιρέτες του n-1. Με αυτή τη φιλοσοφία έχουμε το θεώρημα του Lucas το 1876: « Αν a, n είναι ακέραιοι αριθμοί με n > 1 και $a^{n-1} = 1 \pmod n$ αλλά $a^{(n-1)q} \neq 1 \pmod n$ για κάθε πρώτο q/n-1, τότε ο n είναι πρώτος». Η δυσκολία που έχει το θεώρημα του Lucas για τον έλεγχο ενός αριθμού αν είναι πρώτος δεν είναι η εύρεση του a, αλλά η πλήρης παραγοντοποίηση του n-1. Η παραγοντοποίηση είναι δύσκολη

γενικά αλλά όχι για τους αριθμούς Fermat της μορφής $F_k=2^{2^k}+1$. Το 1877 ο Pepin έδωσε ένα κριτήριο ελέγχου του αν ένας αριθμός Fermat είναι πρώτος. Ο έλεγχος του Pepin είναι:

$$\textit{Για } k\geq 1, \textit{ο αριθμός } F_k=2^{2^k}+1 \textit{ είναι πρώτος μόνο και μόνο αν } 3^{\left(F_k-1\right)/2}\equiv -1\left(mod F_k\right).$$

Θεωρήματα και αλγόριθμοι έχουν αναπτυχθεί για να λύσουν το πρόβλημα της πλήρους παραγοντοποίησης του n-1 αριθμού και το θεώρημα του Lucas έχει χρησιμοποιηθεί ως τεστ για αριθμούς ειδικών μορφών. Το 1983, Οι Adleman, Pomerance και Rumely (όπως αναφέρεται στο [1] δημοσίευσαν έναν έλεγχο (APR test) για πρώτους αριθμούς με χρόνο (ln n) $^{c \ln \ln \ln n}$ όπου n είναι πρώτος και το c θετική σταθερά. Δύο μορφές του συγκεκριμένου ελέγχου αναπτύσσονται, η μία μορφή είναι πιο πρακτική και είναι πιθανοθεωρητικής μορφής ενώ η άλλη είναι ντετερμινιστικής μορφής. Ακόμα και στην πιθανοθεωρητική εκδοχή του αλγορίθμου, η δήλωση ότι ένας αριθμός είναι πρώτος σημαίνει ότι σίγουρα είναι πρώτος. Το βασικό πρόβλημα είναι η πρόβλεψη του υπολογιστικού χρόνου. Αυτό που ακολούθησε ήταν παραλλαγές του ελέγχου APR, με μια κατεύθυνση την εύρεση πιο πρακτικών μορφών του ελέγχου, και με άλλη λιγότερο πρακτικής μορφής αλλά απλούστερης.

Στη συνέχεια έχουμε το 2002 την ανάπτυξη του AKS ελέγχου αν ένας αριθμός είναι πρώτος που θα περιγραφεί αναλυτικά στο επόμενο κεφάλαιο. Ο έλεγχος αυτός θεωρήθηκες ως σημαντική ανάπτυξη καθώς είναι ένας ντετερμινιστικός αλγόριθμος πολυωνυμικού χρόνου, κάτι που δεν χαρακτήριζε τους σύγχρονους αλγορίθμους ούτε και τον αλγόριθμο του Lucas που παρουσιάστηκε παραπάνω, όπου ο χρόνος είναι σχεδόν πολυωνυμικός καθώς η μεταβολή του είναι στην πραγματικότητα πολύ αργή. Όπως αναφέρεται στο [1], ενώ θεωρητικά ο AKS αλγόριθμος είναι μια μεγάλη ανακάλυψη, στην πράξη αν είναι κατάλληλος για τον υπολογισμό μεγάλων αριθμών είναι κάτι που χρειάζεται να δοκιμαστεί μέσα από την υλοποίηση του. Αυτό είναι κάτι που η παρούσα διπλωματική προσπαθεί σε κάποιο βαθμό να κάνει.

Τέλος υπάρχει μεγάλος αριθμός τεστ ελέγχου που είναι εκθετικοί, δηλαδή που ο υπολογιστικός τους χρόνος είναι στη χειρότερη περίπτωση μια σταθερή δύναμη του αριθμου που παραγοντοποιείται. Αν και οι αλγόριθμοι αυτοί δεν έχουν τον πολυωνυμικό χρόνο του AKS, έχουν χρησιμοποιηθεί για τον υπολογισμό πρώτων αριθμών και έχουν αναπτυχθεί διάφορες μέθοδοι, όπως η μέθοδος του Lehman, οι μέθοδοι του Monte Carlo, τετράγωνες μορφής μέθοδοι (quadratic methods) που έχουν αναφερθεί παραπάνω στη μαθηματική οπτική. Οι τετράγωνες μορφές σχετίζονται με τον ECPP αλγόριθμο που θα συζητηθεί και αυτός στο επόμενο κεφάλαιο. Για ακεραίους a, b, c θεωρούμε την τετραγωνική μορφή $ax^2+bxy+cy^2$ που είναι ένα πολυώνυμο, αλλά προτιμούμε να αναφερόμαστε σε μια τετραγωνική μορφή ώς τη διατεταγμένη τριάδα (a, b, c). Η σύγκριση διαφορετικών τετραγωνικών μορφών γίνεται με τη διακρίνουσα b^2-4ac, όπου δύο ισοδύναμες τετραγωνικές μορφές έχουν την ίδια διακρίνουσα (το αντίστροφο δεν ισχύει).

Τέλος μια άλλη περιοχή ελέγχου είναι με βάση τις ελλειπτικές καμπύλες, που αν και αναπτύχθηκαν στην κλασσική ανάλυση, έχουν λάβει τα τελευταία χρόνια σημαντική θέση στην υπολογιστική θεωρία αριθμών. Ο ECPP αλγόριθμος στηρίζεται στις ελλειπτικές καμπύλες και θα παρουσιαστεί αναλυτικά στη συνέχεια.

1.3 Σκοπός και Σημασία της Εργασίας

Το πόσο σημαντικός είναι ο τομέας ανάπτυξης και υλοποίησης αλγορίθμων εύρεσης πρώτων αριθμών φαίνεται από τις εξελίξεις που έχουν επέλθει σε αυτόν τον τομέα στη διάρκεια της ιστορίας. Έχουν περάσει τόσοι αιώνες από τις πρώτες προσπάθειες υπολογισμού πρώτων αριθμών, και όμως ακόμα το ζήτημα αυτό έχει μεγάλο ερευνητικό ενδιαφέρον. Πέρα από το καθαρά μαθηματικό και προγραμματιστικό ενδιαφέρον για τους πρώτους αριθμούς, οι εφαρμογές αυτών στον τομέα της κρυπτογραφίας, δημιουργούν μία άμεση ανάγκη για βελτίωση των

ήδη υπαρχόντων αλγορίθμων, ή ανάπτυξη νέων, μικρότερης χρονικής πολυπλοκότητας και ακρίβειας υπολογισμού.

Στο πλαίσιο αυτό, η παρούσα διπλωματική εργασία μελετά δύο σχετικά σύγχρονους αλγορίθμους, έναν ντετερμινιστικό πολυωνυμικού χρόνου (AKS), και έναν πιθανοθεωρητικό ευρετικού χρόνου (ECPP). Ο αλγόριθμός AKS, μας δίνει για πρώτη φορά τη δυνατότητα εύρεσης αν ένας αριθμός είναι πρώτος σε πολυωνυμικό χρόνο $O(\log^{11.5}(n))$, κάτι που έφερε μεγάλη επανάσταση στο χώρο της θεωρίας αριθμών. Από την άλλη, ο αλγόριθμός ECPP είναι ένας ευρέως διαδεδομένος και γρήγορος αλγόριθμος, αλλά η πιθανοθεωρητική του φύση αποτελεί έναν περιορισμό του.

Στη διπλωματική εργασία αυτή, γίνεται υλοποίηση του αλγορίθμου AKS, ενώ ο αλγόριθμός ECPP έχει υλοποιηθεί από μία άλλη πηγή, και χρησιμοποιείται για να έχουμε ένα μέτρο αναφοράς και σύγκρισης του AKS. Τέλος, ένα άλλο μέτρο αναφοράς αποτελεί το κόσκινο του Ερατοσθένη, για να κατανοήσουμε σε τι τάξης αριθμούς έχει αξία η χρήση του “αργού” αλγορίθμου AKS.

ΚΕΦΑΛΑΙΟ 2: Οι Αλγόριθμοι – Θεωρητικό Υπόβαθρο

2.1 Αλγόριθμος AKS

Ο αλγόριθμος AKS είναι ένας ντετερμινιστικός αλγόριθμος που αποδεικνύει αν ένας δοσμένος αριθμός n είναι πρώτος ή σύνθετος. Δημιουργήθηκε και παρουσιάστηκε τον Αύγουστο του 2002 από τρεις τότε μεταπτυχιακούς φοιτητές, τους Manindra Agrawal, Neeraj Kayal, και Nitin Saxena, από το Ινδικό Τεχνολογικό Ινστιτούτο της Καμπούρ. Ο αλγόριθμος παρουσιάστηκε στο άρθρο τους “Primes in in P”, δημοσιεύθηκε στο φημισμένο περιοδικό Annals of Mathematics το 2004, και πήρε το όνομα του AKS από τα αρχικά των τριών επιστημόνων.

Η σημασία αυτού του αλγορίθμου είναι πολύ μεγάλη, καθότι ήταν ο πρώτος ντετερμινιστικός αλγόριθμος (αλγόριθμος που μας οδηγεί σε σίγουρη απάντηση) πολυωνυμικού χρόνου – στο μέγεθος της εισόδου n που δίνουμε (logn).

Παρακάτω θα αναλύσουμε τα κύρια βήματα του αλγορίθμου AKS:

Είσοδος: Ακέραιος αριθμός n>1.

1 *Αν* $\left(n=a^b \text{ για } a\in\mathbb{N} \text{ και } b>1\right)$ *, επίστρεψε ΣΥΝΘΕΤΟΣ.*

2 *Να βρεθεί ο μικρότερος αριθμός r τέτοιος ώστε* $o_r(n)>\log^2 n$ *.*

3 *Αν* $1<(a,n)<n$ *για κάποιο* $a\le r$ *, επίστρεψε ΣΥΝΘΕΤΟΣ.*

4 *Αν* $n\le r$ *, επίστρεψε ΠΡΩΤΟΣ.*

5 *Για* $a=1$ *μέχρι* $\left\lfloor\sqrt{\varphi(r)}\log n\right\rfloor$ όπου φ(r) ορίζεται στο βημα 5 *κάνε*

αν $\left((X+a)^n\neq X^n+a\left(\mathrm{mod}X^r-1,n\right)\right)$ *, επίστρεψε ΣΥΝΘΕΤΟΣ.*

6 *Επίστρεψε ΠΡΩΤΟΣ.*

Βήμα 1:

Σε αυτό το βήμα ελέγχουμε αν ο αριθμός n είναι εκθετικός, δηλαδή αν μπορεί να γραφεί σε μορφή με βάση α και εκθέτη b φυσικούς αριθμούς, όπου b>1. Αυτός ο υπολογισμός έχει προσεγγιστικά ασυμπτωτική χρονική πολυπλοκότητα $O(\log^3 n)$.

Βήμα 2:

Στο βήμα αυτό ο στόχος μας είναι να βρούμε έναν αριθμό r που να ισχύει η παραπάνω συνθήκη. Ο συμβολισμός $o_r(n)$ ορίζεται ως η τάξη του n modulo r, δηλαδή είναι ο μικρότερος αριθμός k τέτοιος ώστε $n^k = 1 \pmod r$. Όπως αναφέρεται και στο άρθρο “Primes is in P” [3], αυτό το βήμα μπορεί να υλοποιηθεί δοκιμάζοντας διαδοχικές τιμές του r και ελέγχοντας αν $n^k \neq 1 \pmod r$ για κάθε $k \leq \log^2 n$. Για ένα συγκεκριμένο r, αυτό θα χρειαστεί στη χειρότερη περίπτωση $O(\log^2 n)$ πολλαπλασιασμούς modulo r. Επομένως, με χρήση και του λήμματος 2 που παρουσιάζεται παρακάτω, γνωρίζουμε ότι μόνο $O(\log^5 n)$ διαφορετικές τιμές του r πρέπει ν δοκιμαστούν. Η χρονική πολυπλοκότητα του βήματος αυτού είναι λοιπόν $O(\log^7 n)$.

Βήμα 3:

Αυτό το βήμα του αλγορίθμου AKS αποτελείται από διαδοχικούς ελέγχους του Μέγιστου Κοινού Διαιρέτη των αριθμών α και n, για α που τρέχει από 1 μέχρι τον αριθμό n. Αν βρεθεί ότι ο Μέγιστος Κοινός Διαιρέτης (α, n) κάποιου α βρίσκεται στο πεδίο (1, n), τότε ο αριθμός n είναι σύνθετος. Κάθε υπολογισμός του Μ.Κ.Δ. χρειάζεται χρόνο O(logn), οπότε για r υπολογισμούς ο χρόνος που απαιτείται είναι $O(r \log n) = O(\log^5 n * \log n) = O(\log^6 n)$. Αυτή είναι η χρονική πολυπλοκότητα του συγκεκριμένου βήματος.

Βήμα 4:

Στο βήμα αυτό απλά ελέγχουμε αν ο αριθμός n είναι μικρότερος ή ίσος από τον αριθμό r. Αν αυτό ισχύει τότε ο αριθμός n είναι πρώτος και ο αλγόριθμος επιστρέφει την τιμή ΠΡΩΤΟΣ. Η χρονική πολυπλοκότητα αυτού του βήματος είναι O(logn).

Βήμα 5:

Αυτό το βήμα αποτελεί το πιο ουσιαστικό και περίπλοκο βήμα του αλγορίθμου, καθώς και το βήμα που χρειάζεται τον περισσότερο υπολογιστικό χρόνο. Σε αυτό το βήμα λοιπόν, πρέπει πρώτα να κάνουμε τον υπολογισμό του πολυωνύμου $(X + \alpha)^n$, έπειτα να αφαιρέσουμε τον όρο X^n, να αφαιρέσουμε το α, και ύστερα από αυτούς τους υπολογισμούς να κάνουμε την πράξη modulo $X^r - 1$, και με το αποτέλεσμα αυτής της πράξης να κάνουμε modulo n. Να σημειώσουμε εδώ ότι τρέχουμε αυτή τη διαδικασία για κάθε αριθμό α από 1 μέχρι $\lfloor \sqrt{\varphi(r)}\, logn \rfloor$, οπότε βλέπουμε γιατί το βήμα αυτό έχει τις μεγαλύτερες χρονικές απαιτήσεις από όλα τα άλλα βήματα. Ο όρος φ(r) είναι γνωστός ως η συνάρτηση του Euler. Η συνάρτηση αυτή, μας δίνει το πλήθος των φυσικών αριθμών που είναι μικρότεροι από τον αριθμό r οι οποίοι είναι σχετικά πρώτοι με τον αριθμό r (ο Μέγιστος Κοινός Διαιρέτης τους είναι η μονάδα). Ας αναλύσουμε τώρα τη χρονική πολυπλοκότητα αυτού του βήματος. Κάθε ισότητα που πρέπει να ελέγξουμε απαιτεί O(logn) πολλαπλασιασμούς βαθμού r πολυωνύμων, με όρους πολυωνύμου μεγέθους O(logn). Επομένως, κάθε εξίσωση μπορεί να επαληθευτεί σε $O(r \log^2 n)$ βήματα. Επίσης, πρέπει να επαληθέυσουμε $\lfloor \sqrt{\varphi(r)}\, logn \rfloor$ εξισώσεις. Οπότε, η συνολική χρονική πολυπλοκότητα του βήματος 5 είναι $O\left(r\sqrt{\varphi(r)}\, log^3 n\right) = O\left(r^{3/2} \log^3 n\right) = O\left(\log^{21/2} n\right)$. Αυτός ο χρόνος υπερισχύει του χρόνου όλων των άλλων βημάτων, οπότε αυτή είναι η χρονική πολυπλοκότητα του αλγορίθμου AKS.

Παρακάτω θα αναφέρουμε τα βασικά λήμματα στα οποία στηρίχθηκε ο αλγόριθμος AKS για την απόδειξή του:

Λήμμα 1: Αν ο αριθμός n είναι πρώτος, τότε ο αλγόριθμος επιστρέφει ΠΡΩΤΟΣ.

Με τη χρήση αυτού του λήμματος αποδεικνύουμε ότι εάν ο αριθμός n είναι πρώτος, τα βήματα 1,3 και 5 δεν μπορούν ποτέ να επιστρέψουν ΣΥΝΘΕΤΟΣ.

Λήμμα 2: Υπάρχει ένας αριθμός $r \leq max\left(3,\left(\log^5 n\right)\right\}$ *τέτοιος ώστε* $o_r(n) > \log^2 n$.

Ουσιαστικά, αυτό το λήμμα και η απόδειξή του μας βοηθά στο να βρούμε τα όρια που βρίσκεται ο αριθμός r, και να κατανοήσουμε ότι σίγουρα υπάρχει τέτοιος αριθμός r που ικανοποιεί την παραπάνω συνθήκη.

2.2 Αλγόριθμος ECPP

Ο αλγόριθμος Elliptic Curve Primality Proving (ECPP) είναι ένας πιθανοθεωρητικός αλγόριθμος ευρετικού χρόνου, που αποδεικνύει αν ένας αριθμός n είναι πρώτος. Είναι ένας γενικού τύπου αλγόριθμος, δηλαδή δεν εξαρτάται από τη μορφή του αριθμού (π.χ Mersenne αριθμός). Ο ECPP είναι στην πράξη ο πιο γνωστός γρήγορος αλγόριθμος και χρησιμοποιείται αρκετά για να ελέγχει ιδιαίτερα μεγάλους αριθμούς. Η αρχική ιδέα του αλγορίθμου, που στηρίζεται στις τεχνικές των ελλειπτικών καμπυλών, αναπτύχθηκε από τους Goldwasser & Kilian το 1986 (όπως αναφέρεται στο [4]). Το 1991 οι Atkin & Morain προχώρησαν

την ιδέα και διαμόρφωσαν τον ECPP αλγόριθμο. Η γενική μορφή του αλγορίθμου όπως παρουσιάστηκε από τους Goldwasser & Kilian στηρίζεται στις ελλειπτικές καμπύλες της μορφής $y^2=x^3+ax+b$ και περιγράφεται από την παρακάτω πρόταση:

Έστω m ακέραιος αριθμός και q ένας διαιρέτης του m, όπου $q>\left(n^{1/4}+1\right)^2$. Άν P είναι ένα σημείο στην ελλειπτική καμπύλη $y^2=x^3+ax+b$ και ικανοποιεί τις συνθήκες:

- $m\cdot P=0$
- Ο $(m/q)\cdot P$ ορίζεται και είναι διάφορος από το 0

Τότε, αν ο αριθμός q είναι πρώτος αυτό συνεπάγεται ότι ο n είναι πρώτος.

Με βάση την παραπάνω γενική ιδέα, οι Atkin και Morain έφτασαν στην δημιουργία του αλγορίθμου ECPP. Όπως παρατηρούμε και στην παραπάνω γενική ιδέα, ο αλγόριθμος ECPP επιχειρεί να μειώσει τη δυσκολία εύρεσης αν ο δοσμένος αριθμός n είναι πρώτος, βρίσκοντας αν ένας μικρότερος αριθμός q με συγκεκριμένες ιδιότητες είναι πρώτος. Παρακάτω θα περιγράψουμε τα βήματα του αλγορίθμου του Atkin και Morain, όπως παρουσιάζεται στο βιβλίο των Crandall και Pomerance[1].

Βήμα 1: Επιλογή Διακρίνουσας

Επιλέγουμε μία διακρίνουσα D της εξίσωσης της καμπύλης για την οποία ισχύει ότι το σύμβολο του Jacobi $\left(\frac{D}{n}\right)=1$, και για την οποία υπάρχει λύση της εξίσωσης $u^2+|D|v^2=4n$, αποδίδοντας πιθανές καμπύλες τάξης m:

$$m\in\left(n+1\pm u,n+1\pm 2v\right),\gamma\iota\alpha\ D=-4\ ,$$

$$m \in \left(n+1 \pm u, n+1 \pm (u \pm 3v)/2\right), \text{για } D=-3 \text{ ,}$$
$$m \in \left(n+1 \pm u\right), \text{για } D<-4 \text{ .}$$

Να σημειώσουμε εδώ ότι το σύμβολο του Jacobi $\left(\frac{a}{n}\right)$ είναι ίσο με το γινόμενο των συμβόλων του Legendre. Δηλαδή:

$$\left(\frac{a}{n}\right)=\left(\frac{a}{p_1}\right)^{a_1}\left(\frac{a}{p_2}\right)^{a_2}\dots\left(\frac{a}{p_k}\right)^{a_k}, \text{ όπου } n=p_{1^{a_1}} p_{2^{a_2}} \dots p_{k^{a_k}}$$

όπου το σύμβολο του Legendre ισούται με

$$\left(\frac{a}{p}\right)=\left\{\begin{array}{c} 0 \text{αν } a=0(modp) \\ 1 \text{αν } a\neq 0(modp) \text{ και για κάποιο } x: a=x^2(modp) \\ -1 \text{αν } a\neq 0(modp) \text{ και δεν υπάρχει τέτοιο } x \end{array}\right\}$$

Βήμα 2: Παράγοντας τάξης

Βρίσκουμε μία πιθανή τάξη m η οποία παραγοντίζεται ως m = kq, όπου k > 1 και ο αριθμός q είναι πιθανός πρώτος αριθμός, με $q>\left(n^{1/4}+1\right)^2$.

Βήμα 3: Εύρεση παραμέτρων καμπύλης

Κατασκευάζουμε μία ελλειπτική καμπύλη με παραμέτρους a,b η οποία έχει τάξη m, αν θεωρήσουμε ότι ο αριθμός n είναι πρώτος.

Βήμα 4: Εύρεση σημείου στον γεωμετρικό τόπο της καμπύλης

Επιλέγουμε τυχαίο αριθμό x που ανήκει στο διάστημα [0, n-1] , τέτοιο ώστε η εξίσωση $Q = (x^3 + ax + b) \bmod n$ να έχει σύμβολο Jacobi $\left(\frac{Q}{n}\right)\neq 1$. Έπειτα βρίσκουμε έναν ακέραιο αριθμό y τέτοιο ώστε αν ο αριθμός n ήταν πρώτος, να ίσχυε η εξίσωση $y^2=Q(modn)$. Αν δε βρεθεί

τέτοιος αριθμός y, τότε επιστρέφουμε ότι ο αριθμός n είναι σύνθετος. Αν βρεθεί, κρατάμε το σημείο $P = (x, y)$ ως επιλεγμένο σημείο της καμπύλης.

Βήμα 5: Υπολογισμοί πάνω στο σημείο

Σε αυτό το τελικό βήμα, υπολογίζουμε τα πολλαπλά $U = [m/q] P$ (σε περίπτωση λανθασμένης αντιστροφής επιστρέφουμε ότι ο αριθμός n είναι σύνθετος). Αν ο αριθμός U είναι ίσος με 0 τότε υπολογίζουμε $V = [q]U$ (πάλι σε περίπτωση λανθασμένης αντιστροφής επιστρέφουμε ότι ο n είναι σύνθετος). Αν το V είναι διαφορετικό από το 0, τότε επιστρέφουμε ότι ο αριθμός n είναι σύνθετος. Στην αντίθετη περίπτωση, επιστρέφουμε ότι εάν ο αριθμός q είναι πρώτος, τότε και ο αριθμός n είναι πρώτος.

Σε αυτό το σημείο, καλό είναι να αναφερθεί ξανά ότι ο αλγόριθμος είναι ευρετικού χρόνου, δηλαδή γνωρίζουμε σε γενικές γραμμές τον χρόνο εκτέλεσης του αλγορίθμου, όμως δεν γνωρίζουμε τη χειρότερη περίπτωση υπολογισμού του. Έτσι, ειδικότερα στον ECPP, η χρονική πολυπλοκότητα του αλγορίθμου είναι $O((\log n)^{5+\varepsilon})$, για κάποιο $\varepsilon > 0$.

ΚΕΦΑΛΑΙΟ 3: Η Υλοποίηση των Αλγορίθμων

3.1 Αλγόριθμος AKS

Για την υλοποίηση του αλγορίθμου AKS χρησιμοποιήθηκε η γλώσσα προγραμματισμού C. Ειδικότερα, επειδή ο συγκεκριμένος αλγόριθμος περιλαμβάνει αρκετές μαθηματικές συναρτήσεις και υπολογισμούς, έγινε χρήση δύο μαθηματικών βιβλιοθηκών που είναι διαθέσιμες στο διαδίκτυο για τη γλώσσα C, της βιβλιοθήκης GMP, και της βιβλιοθήκης MPFR. Οι κυριότερες και πιο σημαντικές συναρτήσεις αυτών των δύο βιβλιοθηκών θα παρουσιαστούν παρακάτω.

3.1.1 Η Βιβλιοθήκη GMP

Η GMP είναι μία δωρεάν βιβλιοθήκη για ακρίβεια αριθμητικών υπολογισμών ακεραίων και αριθμών κινητής υποδιαστολής. Περιλαμβάνει ένα μεγάλο εύρος συναρτήσεων, και έχει σχεδιαστεί με στόχο την ταχύτητα υπολογισμών, τόσο σε μικρούς όσο και σε μεγάλους αριθμούς. Δεν έχει περιορισμούς στην ακρίβεια υπολογισμού, παρά μόνο τους περιορισμούς που προκύπτουν από τη διαθέσιμη μνήμη του μηχανήματος που τρέχει η βιβλιοθήκη. Η κύρια χρήση της βιβλιοθήκης GMP βρίσκεται σε εφαρμογές κρυπτογραφίας και ασφάλειας, σε αλγεβρικά συστήματα, καθώς και σε άλλα ερευνητικά πεδία. Η πρώτη έκδοση της βιβλιοθήκης διατέθηκε το 1991.

Παρακάτω παρουσιάζονται οι βασικές συναρτήσεις και κλήσεις μεταβλητών που χρησιμοποιήθηκαν στην υλοποίηση του αλγορίθμου AKS:

mpz_t

Ο τύπος ακεραίων για τη βιβλιοθήκη GMP. Κάθε ακέραιος που θα χρησιμοποιηθεί στις συναρτήσεις πρέπει να ορίζεται σαν mpz_t πριν τη χρήση του στην κάθε συνάρτηση.

mpz_init(mpz_t x)

Αρχικοποίηση της μεταβλητής x σε 0. Απαραίτητη εντολή για χρήση της μεταβλητής από συναρτήσεις της GMP.

mpz_clear(mpz_t x)

Απελευθέρωση της μνήμης που χρησιμοποιείται από τη μεταβλητή x. Καλούμε αυτή τη συνάρτηση για όλες τις μεταβλητές τύπου mpz_t όταν έχουμε τελειώσει με τη χρήση τους.

mpz_sgn(mpz_t op)

Επιστρέφει +1 αν op > 0, 0 αν op = 0 , και -1 αν op < 0.

mpz_cmp(mpz_t op1, mpz_t op2)

Σύγκριση των μεταβλητών op1 και op2. Επιστρέφει θετική τιμή αν op1 > op2, αρνητική τιμή αν op1 < op2, και μηδέν αν op1 = op2.

mpz_set(mpz_t rop, mpz_t op)

Αποθήκευση της τιμής της μεταβλητής op στη μεταβλητή rop.

mpz_add(mpz_t rop, mpz_t op1, mpz_t op2)

Άθροιση των μεταβλητών op1 + op2 και αποθήκευση στη μεταβλητή rop.

mpz_sub(mpz_t rop, mpz_t op1, mpz_t op2)

Αφαίρεση των μεταβλητών op1 – op2 και αποθήκευση στη μεταβλητή rop.

mpz_mul(mpz_t rop, mpz_t op1, mpz_t op2)

Πολλαπλασιασμός των μεταβλητών op1 * op2 και αποθήκευση στη μεταβλητή rop.

mpz_mod(mpz_t r, mpz_t n, mpz_t d)

Υπολογισμός της πράξης n *modulo* d και αποθήκευση στη μεταβλητή r.

mpz_gcd(mpz_t rop, mpz_t op1, mpz_t op2)

Υπολογισμός του μέγιστου κοινού διαιρέτη των μεταβλητών op1 και op2 και αποθήκευση στη μεταβλητή rop. Το αποτέλεσμα είναι πάντοτε θετικό, ακόμα και αν οι αριθμοί op1, op2 είναι αρνητικοί. Αν op1 = op2 = 0 τότε η συνάρτηση επιστρέφει 0.

mpz_sqrt(mpz_t rop, mpz_t op)

Υπολογισμός της τετραγωνικής ρίζας της μεταβλητής op, στρογγυλοποίηση και αποθήκευση στη μεταβλητή rop.

mpz_sizeinbase(mpz_t op, int base)

Επιστροφή του αριθμού των ψηφίων της μεταβλητής op με τιμή βάσης του αριθμού τη μεταβλητή base. Η βάση μπορεί να κυμαίνεται από 2 εώς 62. Το πρόσημο της μεταβλητής op αγνοείται, και χρησιμοποιείται μόνο η απόλυτη τιμή. Το αποτέλεσμα της συνάρτησης θα είναι είτε ακριβές

είτε μεγαλύτερο κατά 1. Αν η βάση είναι δύναμης του 2, τότε το αποτέλεσμα είναι πάντα ακριβές. Αν η μεταβλητή op είναι 0, τότε η συνάρτηση επιστρέφει πάντα τιμή 1.

mpz_powm(mpz_t rop, mpz_t base, mpz_t exp, mpz_t mod)

Υπολογισμός της πράξης $base^{exp}$ *modulo* mod και αποθήκευση στη μεταβλητή rop.

mpz_perfect_power_p(mpz_t op)

Επιστροφή ενός μη μηδενικού αριθμού εάν ο αριθμός της μεταβλητής op είναι τετραγωνικός, δηλαδή αν υπάρχουν ακέραιοι a, b με b > 1 τέτοιοι ώστε op = a^b.

*mpz_import(mpz_t rop, size_t count, int order, size_t size, int endian, size_t nails, const void *op)*

Αποθήκευση στη μεταβλητή rop ενός πίνακα λέξεων δεδομένων op. Οι παράμετροι καθορίζουν τη μορφή των δεδομένων. Count είναι ο αριθμός των λέξεων που διαβάζονται, κάθε μία μεγέθους size. Η μεταβλητή order είναι ίση με 1 αν θέλουμε την πιο σημαντική λέξη πρώτα, ενώ -1 αν θέλουμε την λιγότερο σημαντική λέξη. Μέσα σε κάθε λέξη η τιμή του endian είναι 1 για το πιο σημαντικό byte πρώτα, -1 για το λιγότερο σημαντικό byte, ή 0 για χρήση του συστήματος ταξινόμησης δεδομένων του επεξεργαστή του μηχανήματός μας. Η μεταβλητή nails παίρνει σαν τιμή τον αριθμό των πιο σημαντικών bits της κάθε λέξης που επιθυμούμε να παραλείψουμε. Αν θέλουμε να χρησιμοποιήσουμε όλη τη λέξη ορίζουμε την τιμή της μεταβλητής nails σε 0. Η μεταβλητή rop θα είναι πάντα χωρίς πρόσημο.

*mpz_export(void *rop, size_t *countp, int order, size_t size, int endian, size_t nails, mpz_t op)*

Αποθήκευση στο rop λέξεων δεδομένων από τη μεταβλητή op. Οι παράμετροι καθορίζουν τη μορφή των δεδομένων. Κάθε λέξη θα είναι μεγέθους bytes ίσα με την τιμή της μεταβλητής size. Η μεταβλητή order είναι ίση με 1 αν θέλουμε την πιο σημαντική λέξη πρώτα, ενώ -1 αν θέλουμε την λιγότερο σημαντική λέξη. Μέσα σε κάθε λέξη η τιμή του endian είναι 1 για το πιο σημαντικό byte πρώτα, -1 για το λιγότερο σημαντικό byte, ή 0 για χρήση του συστήματος ταξινόμησης δεδομένων του επεξεργαστή του μηχανήματός μας. Η μεταβλητή nails παίρνει σαν τιμή τον αριθμό των πιο σημαντικών bits της κάθε λέξης που επιθυμούμε να παραλείψουμε (τα παραλειπόμενα τα θέτουμε σε 0). Αν θέλουμε να χρησιμοποιήσουμε όλη τη λέξη ορίζουμε την τιμή της μεταβλητής nails σε 0. Ο αριθμός των λέξεων που παράγονται γράφεται στο δείκτη *countp, ή ορίζουμε την τιμή σε NULL αν θέλουμε να παραλείψουμε την αρίθμηση. Η μεταβλητή rop θα πρέπει να έχει αρκετό χώρο για την αποθήκευση των δεδομένων. Το πρόσημο της μεταβλητής αγνοείται και το αποτέλεσμα είναι σε απόλυτη τιμή.

3.1.2 Η Βιβλιοθήκη MPFR

Η MPFR είναι μία βιβλιοθήκη της γλώσσας προγραμματισμού C για υπολογισμούς κινητής υποδιαστολής πολλαπλής ακρίβειας με σωστή στρογγυλοποίηση. Η βιβλιοθήκη αυτή έχει ως στόχο την απόδοση και τον ορθό και σωστό ορισμό της σημασιολογίας της. Είναι μία βιβλιοθήκη που διατίθεται δωρεάν και χρησιμοποιείται περισσότερο σε μαθηματικές εφαρμογές υψηλής ακρίβειας.

Παρακάτω παρουσιάζονται οι βασικές συναρτήσεις και κλήσεις μεταβλητών που χρησιμοποιήθηκαν στην υλοποίηση του αλγορίθμου AKS:

mpfr_t

Ο τύπος ακεραίων για τη βιβλιοθήκη MPFR. Κάθε ακέραιος που θα χρησιμοποιηθεί στις συναρτήσεις πρέπει να ορίζεται σαν mfr_t πριν τη χρήση του στην κάθε συνάρτηση.

mpz_init_set(mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd)

Αρχικοποίηση της μεταβλητής rop και αποθήκευση σε αυτή της τιμής της μεταβλητής op με κατάλληλη στρογγυλοποίηση rnd. Απαραίτητη εντολή για χρήση της μεταβλητής από συναρτήσεις της MPFR.

mpz_clear(mpfr_t x)

Απελευθέρωση της μνήμης που χρησιμοποιείται από τη μεταβλητή x. Καλούμε αυτή τη συνάρτηση για όλες τις μεταβλητές τύπου mpfr_t όταν έχουμε τελειώσει με τη χρήση τους.

mpfr_get_z(mpz_t rop, mpfr_t op, mpfr_rnd_t rnd)

Μετατροπή της μεταβλητής op σε μεταβλητή rop τύπου mpz_t (GMP) με κατάλληλη στρογγυλοποίηση.

mpfr_log2(mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd)

Υπολογισμός του λογαρίθμου με βάση 2 της μεταβλητής op, και αποθήκευση στη μεταβλητή rop με την κατάλληλη στρογγυλοποίηση rnd.

mpfr_pow(mpfr_t rop, mpfr_t op1, mpfr_t op2, mpfr_rnd_t rnd)

Υπολογισμός της ύψωσης $op1^{op2}$ και αποθήκευση στη μεταβλητή rop με την κατάλληλη στρογγυλοποίηση rnd.

mpfr_ceil(mpfr_t rop, mpfr_t op)

Αποθήκευση της μεταβλητής op στη μεταβλητή rop με στρογγυλοποίηση στον ακριβώς μεγαλύτερο ή ίσο ακέραιο.

Η υλοποίηση

Στο παραπάνω κεφάλαιο αναλύσαμε τα βήματα του αλγορίθμου AKS. Παρακάτω θα περιγράψουμε και θα αναλύσουμε τις συναρτήσεις που κατασκευάστηκαν για το κάθε βήμα του αλγορίθμου. Ολόκληρος ο κώδικας του αλγορίθμου παρατίθεται στο Παράρτημα.

Βήμα 1

Στο βήμα 1 γίνεται έλεγχος αν ο αριθμός n είναι τετραγωνικός, δηλαδή αν υπάρχουν φυσικοί αριθμοί a,b με b > 1 τέτοιοι ώστε $n = a^b$. Αυτό το ελέγχουμε με τη συνάρτηση της GMP *mpz_perfect_power_p(n)*.

Βήμα 2

Σε αυτό το βήμα, όπως αναφέραμε στο προηγούμενο κεφάλαιο, πρέπει να βρούμε έναν αριθμό r, ο οποίος να ικανοποιεί τη συνθήκη $o_r(n) > \log^2 n$. Έτσι λοιπόν έγινε κατασκευή της εξής συνάρτησης:

void find_r(mpz_t r, mpz_t n)

Σε αυτή τη συνάρτηση εισάγουμε 2 μεταβλητές, r και n, και παίρνουμε σαν αποτέλεσμα την τελική τιμή του r ώστε να ισχύει η παραπάνω συνθήκη. Αυτό γίνεται αρχικοποιώντας την τιμή της μεταβλητής r σε 3, και ξεκινώντας μια επανάληψη που ελέγχει αν υπάρχει αριθμός k στο διάστημα $[1, \log^2 n]$ τέτοιος ώστε n^k *modulo* r = 1 (το k είναι η τιμή του $o_r(n)$). Αν δεν βρεθεί τέτοιος αριθμός k, τότε επαναλαμβάνουμε αυξάνοντας την τιμή του r κατά 1. Όταν βρεθεί ο αριθμός k που

ικανοποιεί τη συνθήκη, το αποτέλεσμα του r είναι το ζητούμενο για το βήμα αυτό. Σε κάθε περίπτωση, συνεχίζουμε στο επόμενο βήμα.

Βήμα 3

Στο βήμα 3 ελέγχουμε αν υπάρχει αριθμός a ≤ r τέτοιο ώστε 1 < (a, n) < n. Αυτό το καταφέρνουμε κατασκευάζοντας τη συνάρτηση:

int check_gcd(mpz_t n, mpz_t r)

Στη συνάρτηση αυτή τρέχουμε μία επανάληψη για τιμή του a από 1 μέχρι r (το r που βρήκαμε στο προηγούμενο βήμα), και με τη βοήθεια της συνάρτησης *mpz_gcd(mpz_t rop, mpz_t op1, mpz_t op2)* της βιβλιοθήκης GMP υπολογίζουμε το μέγιστο κοινό διαιρέτη του a και του n. Αν τελικά βρεθεί ότι κάποιο a έχει μέγιστο κοινό διαιρέτη με το n στο εύρος (1,n), τότε επιστρέφουμε ότι ο αριθμός n είναι σύνθετος και τερματίζουμε το πρόγραμμα. Σε αντίθετη περίπτωση, συνεχίζουμε στα επόμενα βήματα.

Βήμα 4

Σε αυτό το βήμα κάνουμε έναν απλό έλεγχο αν n ≤ r. Αν η ανισότητα ισχύει, τότε ο αριθμός n είναι πρώτος και τερματίζουμε το πρόγραμμα. Σε αντίθετη περίπτωση, προχωράμε στο πέμπτο βήμα.

Βήμα 5

Το βήμα αυτό αποτελεί το πιο σημαντικό και πολύπλοκο βήμα του αλγορίθμου. Όπως αναλύσαμε στο προηγούμενο κεφάλαιο, σε αυτό το βήμα ελέγχουμε την ισότητα $(x + a)^n = x^n + a \pmod{x^r-1, n}$ για κάθε τιμή του a από 1 μέχρι $\sqrt{\varphi(r)} * \log(n)$. Ο έλεγχος αυτός γίνεται μέσω της συνάρτησης:

int test_congruence(unsigned long a, mpz_t n, mpz_t r, mpz_t poly1, mpz_t* poly2)*

Όπως βλέπουμε παραπάνω, αυτή η συνάρτηση δέχεται ώς ορίσματα τις μεταβλητές a, n, r, και δύο δείκτες πινάκων, poly1 και poly2. Σημειώνεται ότι πριν από την κλήση της συνάρτησης πρέπει να έχουμε δεσμεύσει χώρο στη μνήμη για τους δύο πίνακες, κάτι το οποίο έχουμε κάνει στον κύριο κώδικα του αλγορίθμου μας. Η συνάρτηση test_congruence κάνει τον έλεγχο της ισότητας για το συγκεκριμένο a που μας δίνεται, και επιστρέφει μία τιμή, 0 αν δεν ισχύει η ισότητα, και 1 εάν ισχύει. Αν ισχύει η ισότητα για όλα τα a, τότε ο αριθμός n είναι πρώτος. Σε κάθε άλλη περίπτωση ο αριθμός n είναι σύνθετος. Ας αναλύσουμε όμως περισσότερο πώς υλοποιούμε το πέμπτο βήμα.

Καλώντας τη συνάρτηση test_congruence, αρχικοποιούμε το πρώτο πολυώνυμο – πίνακα που κατασκευάσαμε, ορίζοντας όλους τους όρους του πολυωνύμου σε 0, εκτός από τον πρώτο όρο που είναι ίσος με a και τον δεύτερο όρο που είναι ίσος με 1. Κοινώς, αποθηκεύουμε στον πρώτο πίνακα poly1 το πολυώνυμο x + a. Καλό είναι να σημειώσουμε εδώ ότι ο πίνακας poly1 έχει μέγεθος ίσο με r. Αυτό συμβαίνει διότι όταν εκτελούμε την πράξη modulo ενός μεγάλου πολυωνύμου n βαθμού (με n > r), τότε το αποτέλεσμα που παίρνουμε είναι ένα πολυώνυμο με το πολύ r όρους.

Έπειτα από την κλήση της συνάρτησης test_congruence, και μέσα στην ίδια της συνάρτησης, καλούμε μία άλλη συνάρτηση, την εξής:

*void polyonym_power(mpz_t *one, mpz_t *two, mpz_t power, unsigned long r, mpz_t mod)*

Αυτή η συνάρτηση δέχεται στα ορίσματα one και two τους δείκτες των δύο πολυωνύμων poly1 και poly2, στη μεταβλητή power την τιμή του n, και στη μεταβλητή mod πάλι την τιμή του n. Η polyonym_power δέχεται

αυτά τα ορίσματα, και μας βοηθάει στο να μειώσουμε τον αριθμό των υπολογισμών. Χρησιμοποιούμε μία μέθοδο γρήγορης ύψωσης σε δύναμη, ώστε να υπολογίσουμε το πολυώνυμο $(x+a)^n$ όσο το δυνατόν ταχύτερα. Αυτό επιτυγχάνεται με τον εξής τρόπο:

Παρατηρούμε ότι $(x+a)^n=\begin{cases}(x+a)\left[(x+a)^2\right]^{\left(\frac{n-1}{2}\right)}, \text{για } n \text{ περιττό} \\ \left[(x+a)^2\right]^{\left(\frac{n}{2}\right)}, \text{για } n \text{ άρτιο}\end{cases}$, οπότε με χρήση αυτής της ιδιότητας, χρησιμοποιώντας κάθε φορά την ακέραια διαίρεση του αριθμού n, και ελέγχοντας αν ο αριθμός που προκύπτει είναι περιττός ή άρτιος, μπορούμε να μειώσουμε κατά πολύ τους υπολογισμούς. Ειδικότερα, αν υπολογίζαμε το πολυώνυμο χωρίς αυτή την παρατήρηση θα χρειαζόταν να κάνουμε την πράξη του πολλαπλασιασμού n φορές. Με χρήση αυτής της ιδιότητας, οι υπολογισμοί μειώνονται σε logn. Παρατηρούμε λοιπόν πόσο χρήσιμη είναι αυτή η ιδιότητα. Έτσι λοιπόν, με τη συνάρτηση polyonym_power, σε κάθε ακέραια διαίρεση, και ανάλογα αν ο αριθμός που προκύπτει είναι περιττός ή άρτιος, καλούμε μία ακόμα συνάρτηση, η οποία κάνει το πιο ουσιαστικό βήμα, τον πολλαπλασιασμό του πολυωνύμου. Αυτή η συνάρτηση είναι η εξής:

void polyonym_multiplication_mod(mpz_t p1, mpz_t* p2, unsigned long r, mpz_t mod, mpz_t t1, mpz_t t2, mpz_t t3)*

Η polyonym_multiplication_mod καλείται όπως είπαμε και πριν σε κάθε βήμα ακέραιας διαίρεσης του n με το 2, και αυτό που κάνει είναι ότι παίρνει τα ορίσματα p1, p2 (δείκτες που δείχνουν στα πολυώνυμα που έχουμε κατασκευάσει από πριν), τις τιμές του r και του mod (n) που θέλουμε να εκτελέσουμε, και άλλες τρεις μεταβλητές t1, t2, και t3, τις οποίες τις δημιουργούμε εμείς σαν προσωρινές μεταβλητές, και μας επιστρέφει το αποτέλεσμα του πολλαπλασιασμού των πολυωνύμων, του modulo x^r-1, και του modulo n.

Ας κάνουμε όμως τώρα μία εισαγωγή, για να εξηγήσουμε πώς εκτελούμε όλες αυτές τις πράξεις. Αρχικά, όσον αφορά το $(x+a)^k$ *modulo* x^r-1 , παρατηρούμε ότι όταν διαιρούμε ένα πολυώνυμο με το πολυώνυμο x^r-1, η διαίρεση είναι ιδιαίτερα απλή. Όπως αναφέρουν οι Crandall και Papadopoulos [10], για να πάρουμε το αποτέλεσμα της πράξης αυτής, αρκεί να εξάγουμε τους πρώτους r όρους του πολλαπλασιασμένου πολυωνύμου, και έπειτα, κάνοντας δεξιά ολίσθηση τον αριθμό που έχει μείνει κατά r bits, προσθέτουμε τους εναπομείναντες όρους. Αν οι όροι είναι πάλι περισσότεροι από r, εκτέλουμε ξανά την ίδια διαδικασία, μέχρις ότου να μην υπάρχουν παραπάνω από r όροι. Ας δώσουμε όμως ένα παράδειγμα για να γίνει πιο κατανοητή η διαδικασία αυτή:

Έστω ότι έχουμε δύο πολυώνυμα, το πολυώνυμο $P1(x) = x^6 + 8x^5 + 40x^4 +102x^3 + 168x^2 +72x +9$ και θέλουμε να πάρουμε το υπόλοιπο της διαίρεσης (δηλαδή το modulo) με το πολυώνυμο $P2(x) = x^5 -1$. Αυτό θεωρητικά μπορεί να γίνει πολύ απλά στο χαρτί με την Ευκλείδεια διαίρεση, αλλά στο υπολογιστικό κομμάτι είναι λίγο δύσκολο. Πραγματοποιώντας την πράξη στο χαρτί, βρίσκουμε ότι P1(x) *modulo* $P2(x) = 40x^4 +102x^3 + 168x^2 + 73x +17$. Ας πάμε να κάνουμε και την πράξη με τον τρόπο που περιγράψαμε παραπάνω:

Αρχικά πάιρνουμε μόνο τους όρους του πολυωνύμου για ευκολία.

1 8 40 102 168 72 9

Το πολυώνυμο P2(x) είναι 5ου βαθμού (r = 5) οπότε εξάγουμε τους 5 πρώτους όρους του P1(x) και κάνουμε ολίσθηση δεξιά 5 θέσεις. Το αποτέλεσμα της εξάγωγής το ορίζουμε P3(x), με όρους 40 102 168 72 9, ενώ το P2(x) μετά την ολίσθηση έχει όρους 0 0 0 0 0 1 8. Τελικά, προσθέτοντας τα P2(x) και P3(x), παίρνουμε σαν αποτέλεσμα ένα πολυώνυμο με όρους:

40 102 168 72 9

+ 0 0 0 1 8
40 102 168 73 17

που είναι το αποτέλεσμα που βρήκαμε και με την Ευκλείδεια διαίρεση. Όσον αφορά την εκτέλεση του πολλαπλασιασμού, χρησιμοποιούμε μία μέθοδο γνωστή και ως δυαδική κατάτμηση (binary segmentation) [10]. Σύμφωνα με αυτή τη μέθοδο, έστω ότι έχουμε δύο πολυώνυμα $f=\sum_i f_i x^i$ και $g=\sum_i g_i x^i$ ίδιου βαθμού r-1. Τότε, μπορούμε να αναπαραστήσουμε τα δύο αυτά πολυώνυμα σαν δύο μεγάλους ακεραίους αριθμούς της μορφής

$$F=0\ldots0f_{r-1}0\ldots0f_{r-2}0\ldots..0f_0,$$
$$G=0\ldots0g_{r-1}0\ldots0g_{r-2}0\ldots..0g_0$$

όπου τα μέρη “0...0” υποδεικνύουν έναν αριθμό μηδενικών, τέτοιο ώστε αν πολλαπλασιάσουμε F * G να πάρουμε σαν αποτέλεσμα τον πολλαπλασιασμό των πολυωνύμων f * g. Η ύπαρξη των μηδενικών αυτών χρησιμέυει ώστε ο πολλαπλασιασμός των όρων των πολυωνύμων να μη δημιουργεί υπερχείλιση και να επηρεάζει τον πολλαπλασιασμό επόμενων όρων. Ουσιαστικά λοιπόν φτιάχνουμε μερικά “κελιά” αριθμών, όλα ίσα μεταξύ τους. Έπειτα από τον πολλαπλασιασμό F * G, αρκεί να χωρίσουμε το αποτέλεσμα σε κελιά ίσων bits με τα προηγούμενα, ώστε να πάρουμε το επιθυμητό αποτέλεσμα. Αυτή η μέθοδος, εκτός από ευκολία υλοποίησης, μας παρέχει και πιο γρήγορους υπολογισμούς, καθώς ο πολλαπλασιασμός πολυωνύμων ανάγεται σε πολλαπλασιασμό μεγάλων ακεραίων αριθμών, κάτι που είναι πολύ πιο γρήγορο.
Έτσι, για να το συνδέσουμε και με την υλοποίησή μας, χρησιμοποιώντας τις συναρτήσεις

*mpz_import(mpz_t rop, size_t count, int order, size_t size, int endian, size_t nails, const void *op)*

και

*mpz_export(void *rop, size_t *countp, int order, size_t size, int endian, size_t nails, mpz_t op)*

της βιβλιοθήκης GMP, εισάγουμε τους όρους των πολυωνύμων στις προσωρινές μεταβλητές t1 και t2, και αφού εκτελέσουμε τον πολλαπλασιασμό, επιστρέφουμε ξανά τις τιμές στους πίνακες του κάθε πολυωνύμου, αφού πρώτα κάνουμε και την πράξη modulo n. Έτσι, στις τιμές του πολυωνύμου poly2 έχουμε το αποτέλεσμα $[(x + a)^n$ *modulo* $x^r-1]$ *modulo* n. Αυτό που μένει λοιπόν είναι να κάνουμε $[(x^n + a)$ *modulo* $x^r-1]$ *modulo* n. Παρατηρούμε όμως ότι x^n *modulo* $x^r-1 = x^{n \text{ modulo } r}$, οπότε αρκεί να αφαιρέσουμε 1 από τον όρο poly2[n modulo r] του πίνακά μας, και έπειτα να κάνουμε την πράξη modulo n.

Τέλος, επιστρέφοντας ξανά στη συνάρτηση test_congruence, ελέγχουμε εάν όλοι οι όροι του poly2 είναι 0. Αν δεν είναι, ο δοσμένος αριθμός n είναι σύνθετος, και τερματίζουμε το πρόγραμμά μας. Αλλιώς, συνεχίζουμε μέχρι να τρέξουμε την test_congruence για όλες τις τιμές του a. Τέλος, αν καταφέρει και τρέξει η test_congruence για όλες τις τιμές a, τότε φθάνουμε στο τελικό βήμα.

Βήμα 6

Εκτυπώνουμε ότι ο αριθμός είναι πρώτος.

3.2 Αλγόριθμος ECPP

Η υλοποίηση του αλγορίθμου ECPP – Atkin-Morain έγινε από τον Ryan Lindeman [11] στα πλαίσια ενός μεταπτυχιακού προγράμματος. Παρακάτω θα γίνει μία μικρή περιγραφή της υλοποίησης αυτής.

Αρχικά, η υλοποίηση έχει γίνει στη γλώσσα προγραμματισμού C. Χρησιμοποιήθηκε και εδώ, όπως στον αλγόριθμο AKS, η μαθηματική βιβλιοθήκη GMP που είναι διαθέσιμη δωρεάν στο διαδίκτυο για τη γλώσσα C. Θα περιγράψουμε παρακάτω τα κυριότερα βήματα του αλγορίθμου:

Βήμα 1

Έχοντας κατασκευάσει έναν πίνακα από διακρίνουσες D, ξεκινάμε από την πρώτη τιμή του πίνακα, και ελέγχουμε με τη συνάρτηση της βιβλιοθήκης GMP *mpz_jacobi(mpz_t a, mpz_t b)* αν το σύμβολο του Jacobi (έχει γίνει περιγραφή στο προηγούμενο κεφάλαιο) $\left(\frac{D}{n}\right)=1$, όπως επίσης ελέγχουμε με τη συνάρτηση *ModifiedCornacchia(mpz_t* theU, mpz_t* theV, mpz_t& theP, mpz_t& theD)* αν το D αυτό ικανοποιεί λύση της εξίσωσης

$$u^2+|D|u^2=4n$$

Εάν δεν υπάρχει λύση, προχωράμε στην επόμενη διακρίνουσα του πίνακα.

Βήμα 2

Με χρήση της συνάρτησης *FactorOrders(mpz_t* theM, mpz_t* theQ, mpz_t& theU, mpz_t& theV, mpz_t& theN, mpz_t& theD, unsigned long*

Bmax), ελέγχουμε αν υπάρχει τάξη m που παραγοντίζεται σαν m = kq, με k > 1 και με q ακέραιο αριθμό ο οποίος είναι πιθανοθεωρητικά πρώτος, και ισχύει ότι

$$q > \left(n^{1/4}+1\right)^2$$

Αν δεν μπορεί να βρεθεί τέτοια τάξη m μετά από κάποιες επαναλήψεις που ορίζουμε εμείς στον κώδικα (K_max επαναλήψεις), τότε εκτελούμε την ίδια διαδικασία για άλλη τιμή του πίνακα D.

Βήμα 3

Με τη χρήση της συνάρτησης *ObtainCurveParameters(mpz_t* theA, mpz_t* theB, mpz_t& theN, mpz_t& theD, mpz_t& theG, unsigned int theK),* βρίσκουμε τις παραμέτρους a, b για μία ελλειπτική καμπύλη τάξης m, αν ο αριθμός n είναι πρώτος. Ουσιαστικά, βρίσκουμε τις παραμέτρους της εξίσωσης

$$y^2 = x^3 + ax + b$$

που χαρακτηρίζει την εξίσωση της ελλειπτικής καμπύλης.

Βήμα 4

Με τη βοήθεια της συνάρτησης *ChoosePoint(struct Point* theP, mpz_t& theN, mpz_t& theA, mpz_t& theB),* διαλέγουμε ένα τυχαίο σημείο x στο διάστημα [0, n-1], τέτοιο ώστε Q = (x^3 + ax + b) modulo n, και όπου το σύμβολο Jacobi είναι

$$\left(\frac{Q}{n}\right) \neq -1 \ .$$

Επίσης, με την ίδια συνάρτηση προσπαθούμε να βρούμε έναν ακέραιο y τέτοιο ώστε να ισχύει $y^2 = Q$ (modulo n), αν ο αριθμός n ήταν πρώτος. Τέλος, αν βρούμε ότι η παραπάνω εξίσωση δεν ισχύει, επιστρέφουμε ότι ο αριθμός n είναι σύνθετος.

Βήμα 5

Στο τελευταίο αυτό βήμα πρέπει να υπολογίσουμε όπως αναφέραμε στο προηγούμενο κεφάλαιο τα πολλαπλά U τέτοια ώστε U = (m/q)P. Έπειτα, υπολογίζουμε το V ώστε V = qU. Αν το V ≠ 0, τότε ο αριθμός n είναι σύνθετος. Σε αντίθετη περίπτωση, επιστρέφουμε ότι εάν ο αριθμός q είναι πρώτος, τότε και ο αριθμός n είναι πρώτος. Αυτό το βήμα έχει υλοποιηθεί με χρήση της συνάρτησης:

EvaluatePoint(Point theU, Point* theV, Point& P, mpz_t& theN, mpz_t& theM, mpz_t& theQ, mpz_t& theA, mpz_t& theB)*

ΚΕΦΑΛΑΙΟ 4: Σχόλια – Συμπεράσματα

Στην εργασία αυτή παρουσιάσαμε την εξέλιξη των υπολογιστικών αλγορίθμων εύρεσης πρώτων αριθμών, και ιδιαίτερα τους αλγορίθμους AKS και ECPP. Η υλοποίηση του αλγορίθμου AKS έγινε στη γλώσσα προγραμματισμού C, με χρήση των βιβλιοθηκών πολλαπλής ακρίβειας GMP και MPFR. Αν και η υλοποίηση του αλγορίθμου αυτού έχει πραγματοποιηθεί και από άλλους προγραμματιστές, έγινε προσπάθεια για μία καλύτερη υλοποίηση της 6ης έκδοσης του AKS. Γενικότερα, στο θεωρητικό επίπεδο γίνονται συνέχεια βελτιώσεις του αλγορίθμου ως προς την χρονική πολυπλοκότητά του. Στο πλαίσιο αυτής της διπλωματικής εργασίας υλοποιήσαμε την 6η έκδοση του αλγορίθμου, καθώς αυτή ήταν η πλέον γνωστή και διαδεδομένη έκδοσή του[3].

Παρακάτω συγκρίνουμε τον AKS με το κόσκινο του Ερατοσθένη, για να αναδείξουμε τις δυνατότητες και τους περιορισμούς του. Ο αλγόριθμος AKS, όπως αναφέρθηκε και σε προηγούμενο κεφάλαιο, έχει χρονική πολυπλοκότητα $O(\log^{21/2}n)$. Το κόσκινο του Ερατοσθένη έχει χρονική πολυπλοκότητα O(nlog(logn)). Πραγματοποιώντας μερικές συγκρίσεις μεταξύ των δύο πολυπλοκοτήτων, φτάσαμε στην παρατήρηση ότι για αριθμούς μεγαλύτερους του $1,0995448166189056 * 10^{18}$, ο αλγόριθμος AKS είναι γρηγορότερος από το κόσκινο του Ερατοσθένη, ενώ για μικρότερες τιμές το κόσκινο του Ερατοσθένη είναι προτιμότερο. Ειδικότερα, χρησιμοποιώντας το πρόγραμμα Matlab, δοκιμάσαμε μερικούς αριθμούς, ώστε να βρούμε τελικά ότι στον παραπάνω αριθμό, οι χρονικές πολυπλοκότητες του AKS και του κόσκινου του Ερατοσθένη είναι ίσες. Επομένως, μπορούμε να συμπεράνουμε ότι ο αλγόριθμος AKS έχει πρακτική χρήση για αρκετά μεγάλους αριθμούς. Βέβαια, σύμφωνα με τον Cao [12], οι απαιτήσεις μνήμης για τον AKS είναι πολύ μεγάλες, κάτι που μπορεί να κάνει την πρακτική υλοποίηση του αλγορίθμου πιο αργή από την αναμενόμενη. Οπότε, ο αριθμός που βρήκαμε παραπάνω,

είναι θεωρητικός, και στην πράξη ίσως ο αλγόριθμος AKS να είναι προτιμότερος για μεγαλύτερους αριθμούς n.

Η χρήση του αλγορίθμου ECPP είναι πιο διαδεδομένη από τον αλγόριθμο AKS. Παρ'όλα αυτά, λόγω της πιθανοθεωρητικής του φύσης, ο έλεγχος δεν είναι προσφέρει πάντοτε τελεσίδικη απόφαση. Επιπλέον, επειδή είναι ευρετικού χρόνου, δεν είμαστε σίγουροι για το χρόνο που χρειάζεται η εκτέλεση του. Για τους παραπάνω λόγους, δεν έγινε τελικά σύγκριση του AKS με τον αλγόριθμο ECPP.

Συνοψίζοντας, ο αλγόριθμός AKS αποτελεί μια πολύ σημαντική ανάπτυξη στην περιοχή της εύρεσης πρώτων αριθμών. Όμως, ο αλγόριθμος αυτός χρειάζεται βελτιώσεις, ώστε η υλοποίησή του να γίνει πιο γρήγορη, και να καταφέρει να γίνει πιο ευρεία η χρήση του.

ΠΑΡΑΡΤΗΜΑ

Αλγόριθμος AKS

```
#include <gmp.h>
#include <mpfr.h>
#include <stdlib.h>
#include <time.h>
#define FALSE 0
#define TRUE 1
#define PRIME 1
#define COMPOSITE 0

unsigned long limit;
```

```
/*Ypologismos logarithmou*/


void logarithm(mpz_t rop, mpz_t op){
    mpfr_t one;
    /* arxikopoihsh tis metavlitis one = op */
    mpfr_init_set_z(one, op, MPFR_RNDN);
    /* one = log(one) */
    mpfr_log2(one, one, MPFR_RNDN);
    /* rop = one*/
    mpfr_get_z(rop, one, MPFR_RNDN);
```

```
    /* katharismos xwrou tis metavlitis one */
    mpfr_clear(one);

}
```

```
/*Ypologismos logarithmou^2*/


void logarithm_square(mpz_t rop, mpz_t op){
    mpfr_t one;
    /* arxikopoihsh tis metavlitis one = op */
    mpfr_init_set_z(one, op, MPFR_RNDN);
    /* one = log(one) */
    mpfr_log2(one, one, MPFR_RNDN);
    /* one = one^2 */
    mpfr_pow_ui(one, one, 2, MPFR_RNDN);
    /* metatropi tis metavlitis one se int */
    mpfr_ceil(one, one);
    /* rop = one */
    mpfr_get_z(rop, one, MPFR_RNDN);
    /* katharismos xwrou tis metavlitis one */
    mpfr_clear(one);

}
```

```
/*Ypologismos mikroterou r gia to 2o vima tou algorithmou*/


void find_r(mpz_t r, mpz_t n){
    mpz_t limit, k, rop ;
    int found = FALSE;
    /*arxikopoihsh tis metavlitis limit,k,rop*/
    mpz_init(limit);
    mpz_init(k);
    mpz_init(rop);
    /* limit = (logn)^2 */
    logarithm_square(limit, n);
    mpz_set_ui(r, 3);
    while (TRUE){
        /* elegxos an n ^ k = 1 (mod r) */
        found = FALSE;
        for(mpz_set_ui(k, 1); mpz_cmp(k, limit); mpz_add_ui(k, k, 1)){
              mpz_powm(rop, n, k, r);
              if (mpz_cmp_ui(rop, 1)==0||(1 == mpz_tstbit(rop, 0) && 1 ==
mpz_sizeinbase(rop, 2))){
                  found = TRUE;
                   break;
               }
        }
        if (!found){
              break;
```

```
      }
      /* an oxi au3anoume kata 1 to r */
      mpz_add_ui(r, r, 1);

    }
    gmp_printf("r = %Zd\n", r);
}
```

```
/*Vima 3, elegxos an yparxei a<=r wste 1<(a,n)<n */


int check_gcd(mpz_t n, mpz_t r){
    /* arxikopoihsh twn metavlitwn a kai gcd*/
    mpz_t a, gcd;
    /* final h timh pou epistrefei h synartisi*/
    int final = FALSE;
    mpz_init(a);
    mpz_init(gcd);
    /* trexoume loop apo 1 mexri r me vima 1 */
    for (mpz_set_ui(a, 1); mpz_cmp(a, r) <= 0; mpz_add_ui(a, a, 1)){
          /* ypologismos gcd(a,n) */
          mpz_gcd(gcd, a, n);
          /* an 1<(a,n)<n tote final=1 kai exodos  apo to loop */
          if (mpz_cmp_ui(gcd, 1) > 0 && mpz_cmp(gcd, n) < 0){
                final = TRUE;
                break;
```

```
            }
        }
        /* katharismos xwrou twm metavlitwn */
        mpz_clear(a);
        mpz_clear(gcd);
        /* epistrofh ths timhs final: 0 h 1 */
        return final;
}
```

```
/* Ypologismos tou φ(r) */


void phi_r(mpz_t one, mpz_t r){
    mpz_t i, gcd;
    /* arxikopoihsh twn metavlitwn i kai gcd*/
    mpz_init(i);
    mpz_init(gcd);
    mpz_set_ui(one, 0);
    /* trexoume loop apo 1 mexri r me vima 1 */
    for (mpz_set_ui(i, 1); mpz_cmp(i, r) <= 0; mpz_add_ui(i, i, 1)){
            /* ypologismos gcd(a,n) */
            mpz_gcd(gcd, r, i);
           /* an i kai n prwtoi metaxy tous */
            if (mpz_cmp_ui(gcd, 1) == 0 ){
                    /* pros8etoume 1 sto rop */
                    mpz_add_ui(one, one, 1);
```

```
            }
        }
         /* ektypwsh tou φ(r) */
         gmp_printf("phi = %Zd\n", one);
         mpz_clear(i);
         mpz_clear(gcd);
    }
```

```
/* Ypologismos tou sqrt(φ(r)) * log(n) */


void upper_limit(mpz_t rop, mpz_t r, mpz_t n){
    /* arxikopoihsh metavlitwn phi, logn */
    mpz_t phi, logn;
    mpz_init(phi);
    mpz_init(logn);
    /* ypologismos φ(r) */
    phi_r(phi, r);
    /* riza tou φ(r) */
    mpz_sqrt(phi, phi);
    logarithm(logn, n);
    /* rop = sqrt(φ(r)) * log(n) */
    mpz_mul(rop, phi, logn);
    /* ektypwsh rop */
    gmp_printf("upper limit = %Zd\n", rop);
    mpz_clear(phi);
```

```
    mpz_clear(logn);
}
```

```
/* Binary Segmentation Pollaplasiasmos kai Modulo */


void polyonym_multiplication_mod(mpz_t* p1, mpz_t* p2, unsigned long
r, mpz_t mod, mpz_t t1, mpz_t t2, mpz_t t3){
    unsigned long i, bytes;
    char* s;
    mpz_mul(t3, mod, mod);
    mpz_mul_ui(t3, t3, r);
    /* bytes o ari8mos bit tou n^2 * r */
    bytes = mpz_sizeinbase(t3, 2);
    mpz_set_ui(t1, 0);
    mpz_set_ui(t2, 0);
    //pollaplasiasmos me binary segmentation
    /* katanomh mnhmhs kai deikths sti metavliti s */
    s = (char*) calloc(r*bytes, sizeof(char));
    for (i = 0; i < r; i++){
            /* h s pairnei tis times tou p1 me pros8hkh  psifiwn endiamesa */
            mpz_export(s + i*bytes, NULL, -1, 1, 0, 0, p1[i]);
    }
   /* apo8hkeush sth metavlhth t1 kai ka8arismos mnhmhs */
   mpz_import(t1, r*bytes, -1, 1, 0, 0, s);
   free((void*)s);
```

```
/* an ta polywnyma diaforetika */
if(p1 != p2){
  s = (char*) calloc(r*bytes, sizeof(char));
  for (i = 0; i < r; i++){
       /* h s pairnei tis times tou p2 me pros8hkh psifiwn endiamesa */
        mpz_export(s + i*bytes, NULL, -1, 1, 0, 0, p2[i]);
  }
  /* apo8hkeush sth metavlhth t2 kai ka8arismos mnhmhs */
   mpz_import(t2, r*bytes, -1, 1, 0, 0, s);
   free((void*)s);
}
/* an p1 kai p2 ta idia */
if(p1 == p2){
       /* t1 = t1^2 */
       mpz_mul(t1, t1, t1);
}
else{
       /* alliws t1 = t1 * t2*/
       mpz_mul(t1, t1, t2);
}
s = (char*) calloc(2*r*bytes, sizeof(char));
/* apo8ikeusi t1 san char sto s */
mpz_export(s, NULL, -1, 1, 0, 0, t1);
for (i = 0; i < r; i++) {
       /* Apo8ikeusi p1[i] sto panw meros, t3 sto katw meros. */
       mpz_import(p1[i], bytes, -1, 1, 0, 0, s + (i+r)*bytes);
```

```
            mpz_import(t3, bytes, -1, 1, 0, 0, s + i*bytes);
            /* Pros8esh kai mod */
            mpz_add(p1[i], p1[i], t3);
            mpz_mod(p1[i], p1[i], mod);
    }
    free((void*)s);
}
```

```
/* Square and Multiply Me8odos */


void polyonym_power(mpz_t *one, mpz_t *two, mpz_t power, unsigned
long r, mpz_t mod){
    unsigned long i;
    mpz_t mpower, tmp1, tmp2, tmp3;
    for (i=1; i<r; i++){
            /* arxikopoihsh tou 1ou polyonymou se 0 */
           mpz_set_ui(one[i],0);
    }
    /*  kai tou mhdenikou orou se 1*/
    mpz_set_ui(one[0],1);
    /* mpower = n */
    mpz_init_set(mpower, power);
    mpz_init(tmp1);
    mpz_init(tmp2);
    mpz_init(tmp3);
```

```
    /* methodos square and multiply */
    while(mpz_sgn(mpower)>0){
        if(mpz_odd_p(mpower)){
            polyonym_multiplication_mod(one, two, r, mod, tmp1,
tmp2, tmp3);
        }
        mpz_tdiv_q_2exp(mpower, mpower, 1);
        if(mpz_sgn(mpower)>0){
            polyonym_multiplication_mod(two, two, r, mod, tmp1,
tmp2, tmp3);
        }
    }
    mpz_clear(tmp1);
    mpz_clear(tmp2);
    mpz_clear(tmp3);
    mpz_clear(mpower);
}
```

```
/*Vima 5, ypologismos tou polywnymou*/
int test_congruence(unsigned long a, mpz_t n, mpz_t r, mpz_t* poly1,
mpz_t* poly2){
    unsigned long nmodr, terms, i;
    mpz_t tmp;
    int answer = TRUE;
    mpz_init(tmp);
```

```
terms = mpz_get_ui(r);
for(i=0; i<terms; i++){
      /* arxikopoihsh tou 1ou polyonymou se 0 */
      mpz_set_ui(poly1[i], 0);
}
/* ektos twn coefficients a0 kai a1 */
mpz_set_ui(poly1[0], a);
mpz_set_ui(poly1[1], 1);
/* ektelesh ths synarthshs ypologismou */
polyonym_power(poly2, poly1, n, terms, n);
/* nmodr = tmp = n mod r */
mpz_mod(tmp, n, r);
nmodr = mpz_get_ui(tmp);
/* afairesh tou nmodr orou, kai tou a */
mpz_sub_ui(tmp, poly2[nmodr], 1);
mpz_mod(poly2[nmodr], tmp, n);
mpz_sub_ui(tmp, poly2[0], a);
/* teleutaios oros mod n */
mpz_mod(poly2[0], tmp, n);
for(i=0; i<terms; i++){
     /* elegxos an oloi oi oroi einai 0 */
     if (mpz_sgn(poly2[i])!=0){
            answer = FALSE;
     }
}
mpz_clear(tmp);
```

```
    /* epistrofh timhs */
    return answer;
}
```

```
/* O algori8mos AKS */
int aks_prime (mpz_t n){
    int result;
    mpz_t r, rop, *p1, *p2;
    unsigned long new_r, i, a, uplimit;
    if (mpz_cmp_ui(n, 2) == 0) {
        /* to 2 einai prwtos */
        return 1;
    }
    /* elegxos an n>1 kai an diaireitai me to 2*/
    if (mpz_cmp_ui(n, 1) <= 0 || mpz_divisible_ui_p(n, 2)) {
            return 0;
    }
    printf("Checking step 1...\n");
    /* elegxos an n = a^b */
    if (mpz_perfect_power_p(n)) {
            return 0;
    }
    printf("Step 1 passed!\n");
    mpz_init(r);
    printf("Step 2...\n");
```

```
/* euresh katallhlou r */
find_r(r, n);
printf("Step 2 passed!\n");
printf("Checking step 3...\n");
/* elegxos vhma 3 */
if(check_gcd(n, r)){
        mpz_clear(r);
        return 0;
}
printf("Step 3 passed!\n");
printf("Checking step 4...\n");
/* vhma 4 */
if (mpz_cmp(n, r) <= 0) {
         mpz_clear(r);
         return 1;
}
printf("Step 4 passed!\n");
mpz_init(rop);
/* ypologismos limit gia to vima 5 */
upper_limit(rop, r, n);
uplimit = mpz_get_ui(rop);
limit = uplimit ^ (mpz_get_ui(n)-1);
mpz_clear(rop);
new_r = mpz_get_ui(r);
/* dhmiourgia kai arxikopoihsh polywnymwn */
```

```
p1 = (mpz_t*) malloc(new_r * sizeof(mpz_t));
p2 = (mpz_t*) malloc(new_r * sizeof(mpz_t));
for (i = 0; i < new_r; i++) {
   mpz_init(p1[i]);
   mpz_init(p2[i]);
}
printf("Checking step 5...\n");
/* vhma 5 */
for (a = 1; a <= uplimit; a++){
      result = test_congruence(a, n, r, p1, p2);
      if (result == 0){
            for (i = 0; i < new_r; i++) {
                  mpz_clear(p1[i]);
                  mpz_clear(p2[i]);
            }
            free((void*)p1);
            free((void*)p2);
            mpz_clear(r);
            return 0;
       }
}
printf("Step 5 passed!!\n");
/* ka8arismos polywnymwn */
for (i = 0; i < new_r; i++) {
     mpz_clear(p1[i]);
```

```
        mpz_clear(p2[i]);
    }
    free((void*)p1);
    free((void*)p2);
    mpz_clear(r);
    return 1;
}
```

```
int main(int argc, char** argv) {
  int final;
  clock_t begin, end;
  double time_spent;
  mpz_t n;
  mpz_init(n);
  gmp_scanf("%Zd", &n);
  /* xronos arxhs */
  begin = clock();
  final = aks_prime(n);
  /* xronos telous */
  end = clock();
  printf("answer = %d\n", final);
  time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
  printf("total time spent = %f sec\n", time_spent);
  mpz_clear(n);
```

```
  return 0;

}
```

Αλγόριθμος ECPP

```
/* Βασικό Μέρος Κώδικα */

bool AtkinMorain(mpz_t& theN)
{
  // True if theNumber is proven prime, false otherwise
  bool anResult = false;
  // True if theN was found to be prime or composite
  bool anDone = false;
  unsigned char anIndexD = 0;  // Index to selected discriminant
  unsigned int anIterations = 0;  // Number of iterations performed so far
  // Beginning limit for LenstraECM
  unsigned long Bmax = AVG_LENSTRA_B1;

  mpz_t n;  // The n to be tested
  mpz_t m;  // Curve order m
  mpz_t q;  // Factor q that if proven prime means that N is prime
  mpz_t u;  // Solution u
  mpz_t v;  // Solution v
  mpz_t g;  // Quadratic nonresidue g
  mpz_t a;  // Root a
  mpz_t b;  // Root b
  mpz_t Q;  // Q in ChoosePoint step
  Point P;  // Point P used in step 4: ChoosePoint
  Point U;  // Point U used in step 5: EvaluatePoint
  Point V;  // Point V used in step 5: EvaluatePoint
  mpz_t t;  // Temporary variable for testing y^2 mod n != Q

  // Initialize n, m, q, u, and v values
  mpz_init_set(n, theN);
  mpz_init(m);
  mpz_init(q);
  mpz_init(u);
  mpz_init(v);
```

```
mpz_init(g);
mpz_init(a);
mpz_init(b);
mpz_init(P.x);
mpz_init(P.y);
mpz_init(U.x);
mpz_init(U.y);
mpz_init(V.x);
mpz_init(V.y);
mpz_init(t);

// Initialize our fixed list of discriminants
InitDiscriminants();

// Loop through incrementally higher Bmax values if discriminant loop
// fails
do
{
  // Reset our discriminant index value
  anIndexD = 0;

  // Loop through each discriminant in the gD array or until our anDone
  // value is set to true.
  while(!anDone && anIndexD+1 < MAX_DISCRIMINANTS)
  {
    // Step 0: Use Miller-Rabin to test if theN is composite since there is
    // no guarrentee that ECPP will successfully find a u and v in Step 1,
    // but Miller-Rabin guarantees to find all composites quickly.
    int anMillerRabin = mpz_probab_prime_p(n, 10);

    // Did Miller-Rabin prove n is composite?
    if(0 == anMillerRabin)
    {
      if(gDebug)
      {
        printf("Miller-Rabin says n is composite!\n");
      }

      // N is composite
      anResult = false;

      // We are done
      anDone = true;

      // Exit the discriminant loop
      break;
    }
    // Did Miller-Rabin prove n is prime?
    else if(2 == anMillerRabin)
```

```
{
  // N is proven prime
  anResult = true;

  // We are done
  anDone = true;

  // Exit the discriminant loop
  break;
}
// Should we perform a sieve test instead?
else if(mpz_cmp_ui(n, SIEVE_TEST_CUTOFF) <= 0)
{
  // Use sieve test to prove n is prime
  anResult = SieveTest(n);

  // We are done
  anDone = true;

  // Exit the discriminant loop
  break;
}
// Otherwise we should just continue with the Atkin-Morain algorithm
else
{
  // Continue with Atkin-Morain algorithm to prove n is prime or
  // composite
}

// Step 1: ChooseDiscriminant will attempt to find a discriminant D
// that
// satisfies steps 1a, 1b, and step 2 by incrementing through each
// discriminant in our gD array, note that the gD array has a dummy
// entry
// 0 at the beginning so we increment first before testing

anIndexD++;

// Step 1a: Find a discriminant that yields a Jacobi(D,N) == 1
if(1 != mpz_jacobi(gD[anIndexD],n))
  continue; // Jacobi returned -1 or 0, try another discriminant

// Step 1b: Find a u and v that satisfies 4n = u^2 + ABS(D)v^2 using
// the modified Cornacchia algorithm (2.3.13)
if(!ModifiedCornacchia(&u, &v, n, gD[anIndexD]))
  continue; // No solution u or v found, try another discriminant

// Step 2/3: FactorOrders attempts to find a possible order m that
// factors as m = kq where k > 1 and q is a probable prime greater
```

```
    // than (n^0.25 + 1)^2. If this can't be done after K_max iterations
    // then return FALSE and choose a new discriminant D and curve m.
     // No factor q or curve m was found, try another discriminant
     if(!FactorOrders(&m, &q, u, v, n, gD[anIndexD], Bmax))
      continue;

    if(gDebug)
    {
     gmp_printf("Steps 1-3: d=%Zd, u=%Zd, v=%Zd, m=%Zd, q=
%Zd\n",
       gD[anIndexD], u, v, m, q);
    }

    // Step 4a: CalculateNonresidue will find a random quadratic
    // nonresidue
    // g mod p and if D=-3 a noncube g^3 mod p for use in step 4b
    CalculateNonresidue(&g, n, gD[anIndexD]);

    // Now that we have selected a curve m with factor q to be proven,
    //obtain curve parameters and test up to MAX_POINTS
    // to see if N is composite
    unsigned int points = 0;
    do
    {
     // Step 4b: ObtainCurveParameters will attempt to obtain the curve
     // parameters a and b for an elliptic curve that would have order m
     // if N is indeed prime.

     // Loop through each curves a and b values (k iterates over them)
     unsigned int k = 0;
     while(!anDone && ObtainCurveParameters(&a, &b, n,
gD[anIndexD], g, k))
      {
       if(gDebug)
       {
        gmp_printf("Step 4: a=%Zd, b=%Zd\n", a, b);
       }

       // Step 5: ChoosePoint will try and find a point (x,y) on the curve
       // using the a and b values provided from above.
       anDone = ChoosePoint(&P,n,a,b);

       if(gDebug)
       {
        gmp_printf("Step 5: P(%Zd,%Zd)\n", P.x, P.y);
       }

       // Did we find that N is composite while choosing a point?
       if(anDone)
```

```
{
  // N is composite
  anResult = false;

  // Exit the ObtainCurveParameters and Points loops
  break;
}

// Step 6: EvaluatePoint will compute the multiple U = [m/q]P.
// Based on these results N will be either composite or Q << N will
// need to be proven prime to prove that N is prime.
int anTest = EvaluatePoint(&U, &V, P, n, m, q, a, b);

// Did EvaluatePoint determine N was composite?
if(anTest < 0)
{
  // N is composite
  anResult = false;

  // We are done
  anDone = true;

  // Exit the ObtainCurveParameters and Points loops
  break;
}
// Did EvaluatePoint determine N was prime if Q is prime?
else if(anTest > 0)
{
  // N is prime if q can be proven prime
  anResult = true;

  if(gCertificate)
  {
    // Print certificate information
    gmp_printf("n[%d]=%Zd\n", anIterations++, n);
    gmp_printf("d=%Zd\n", gD[anIndexD]);
    gmp_printf("u=%Zd\n", u);
    gmp_printf("v=%Zd\n", v);
    gmp_printf("m=%Zd\n", m);
    gmp_printf("q=%Zd\n", q);
    gmp_printf("a=%Zd\n", a);
    gmp_printf("b=%Zd\n", b);
    gmp_printf("P(%Zd,%Zd)\n", P.x, P.y);
    gmp_printf("U(%Zd,%Zd)\n", U.x, U.y);
    gmp_printf("V(%Zd,%Zd)\n", V.x, V.y);
  }

  // Set n = q and start over
  mpz_set(n, q);
```

```
        // Reset our discriminant choice back to 0 and loop again
        anIndexD = 0;

        // Reset our Bmax value
        Bmax = AVG_LENSTRA_B1;

        // Exit the points loop
        points = MAX_POINTS;

        // TODO: If n < SOME_LIMIT use another algorithm like AKS to
        // prove that n is prime instead of iterating again. Be sure
        // to set anDone = true here if you do this.

        // Exit the ObtainCurveParameters loop
        break;
      }
      // Does EvaluatePoint want us to try another point?
      else
      {
        // Try another a and b value
      }

      // Increment our k value
      k++;
    } // while(!anDone && ObtainCurveParameters(&a,&b,...))

    // Try another point P, keep track of the number of points we
    // have tried (each iteration above adds k points (some curves
    // allow k = 6, k = 4, or k = 2, see ObtainCurveParameters.
    points += k;
  } while(!anDone && points < MAX_POINTS);
} // while(!anDone && anIndexD+1 < MAX_DISCRIMINANTS)

// Increment Bmax value by 10 which is used by LenstraECM to try
// and factor m into q.  If we go through all of our discriminants it
// could be that we didn't try hard enough in the FindFactors step to
// obtain a suitable q value from m, so this will make us try harder
// before we just give up entirely.
Bmax *= 10;

if(!anDone && gDebug)
{
  printf("Bmax increased to %lu\n", Bmax);
}

} while(!anDone && Bmax < MAX_LENSTRA_B1);

// If we ran out of discriminants to prove N is prime then use AKS to
```

```
  // prove N is prime. You could replace this with a different algorithm or
  // implement algorithm (7.5.9) in the ObtainCurveParameters method
  // to allow for more discriminants to try above.
  if(false == anDone &&
    (Bmax >= MAX_LENSTRA_B1 || anIndexD+1 ==
MAX_DISCRIMINANTS))
  {
    if(gDebug)
    {
      printf("Atkin-Morain proof incomplete! Running AKS...\n");
    }

    // Return the result of AKS
    //anResult = (aks_is_prime(n) == 1);
  }

  // Clear our values used above
  mpz_clear(V.y);
  mpz_clear(V.x);
  mpz_clear(U.y);
  mpz_clear(U.x);
  mpz_clear(P.y);
  mpz_clear(P.x);
  mpz_clear(b);
  mpz_clear(a);
  mpz_clear(g);
  mpz_clear(v);
  mpz_clear(u);
  mpz_clear(q);
  mpz_clear(m);
  mpz_clear(n);

  // Return true if value is proven prime, false otherwise
  return anResult;
}
```

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] R. Crandall and C. Pomerance, *Prime Numbers: A Computational Perspective (2nd edition)*. Springer: New York, 2005.

[2] B. Fine, and G. Rosenberg, *Number Theory: An Introduction via the Distribution of Primes*. Birkhauser:Boston , 2007.

[3] M. Agrawal, N. Kayal, and N. Saxena, Primes is in P, *Annals of Mathematics*, **160** (2004), 781-793.

[4] A.O.L. Atkin, and F. Morain, Elliptic Curves and Primality Proving, *Mathematics and Computation*, **61** (1993), 29-68.

[5] W. Narkiewicz, *The Development of Prime Number Theory: From Euclid to Hardy and Littlewood*. Springer-Verlag: Berlin, 2000.

[6] Σ. Πετρίδης, *Πρώτοι αριθμοί: Μία ιστορική παρουσίαση από τα Αρχαία Ελληνικά Μαθηματικά μέχρι τις συγχρονες εφαρμογές τους*, Διπλωματική Εργασία ΕΚΠΑ, και Πανεπιστήμιο Κύπρου, Αθήνα, 2011.

[7] L. A. Dickson, *History of the Theory of Numbers: Divisibility and Primality* (Volume 1). Carnegie Institution of Washington: Washingthon, 1919.

[8] A. E. Ingham, *The Distribution of Prime Numbers*. Stechert-Hafner Service Agency: New York and London, 1964.

[9] GIMPS Project Discovers Largest Known Prime Number, *www.mersenne.org .*

[10] R. Crandall, and J. Papadopoulos, On the implementation of AKS-class primality tests, *http://citeseerx.ist.psu.edu/.*

[11] R. Lindeman, *https://github.com/GatorQue.*

[12] Z. Cao, A Note on Storage Requirement for AKS Primality Testing Algorithm, *https://eprint.iacr.org/2013/449.pdf.*