ABSTRACT

COMPUTATIONALLY EFFICIENT SEARCH FOR PRIME NUMBERS

by
Wieslawa E. Amber

To satisfy the speed of communication and to meet the demand for the continuously larger prime numbers, the primality testing and prime numbers generating algorithms require continuous advancement. To find the most efficient algorithm, a need for a survey of methods arises. Concurrently, an urge for the analysis of algorithms' performances emanates. The critical criteria in the analysis of the prime numbers generation are the number of probes, number of generated primes, and an average time required in producing one prime. Hence, the purpose of this thesis is to indicate the best performing algorithm. The survey the methods, establishment of the comparison criteria, and comparison of approaches are the required steps to find the best performing algorithm.

In the first step of this research paper the methods were surveyed and classified using the approach described in Menezes [66]. While chapter 2 sorted, described, compared, and summarized primality testing methods, chapter 3 sorted, described, compared, and summarized prime numbers generating methods. In the next step applying a uniform technique, the computer programs were written to the selected algorithms. The programs were installed on the Unix operating system, running on the Sun 5.8 server to perform the computer experiments. The computer experiments' results pertaining to the selected algorithms, provided required parameters to compare the algorithms' performances. The results from the computer experiments were tabulated to compare the parameters and to indicate the best performing algorithm.

Survey of methods indicated that the deterministic and randomized are the main approaches in prime numbers generation. Random number generation found application in the cryptographic keys generation. Contemporaneously, a need for deterministically generated provable primes emerged in the code encryption, decryption, and in the other cryptographic areas.

The analysis of algorithms' performances indicated that the prime numbers generated through the randomized techniques required smaller number of probes. This is due to the method that eliminates the non-primes in the initial step, that pre-tests randomly generated primes for possible divisibility factors. Analysis indicated that the smaller number of probes increases algorithm's efficiency. Further analysis indicated that a ratio of randomly generated primes to the expected number of primes, generated in the specific interval is smaller than the deterministically generated primes. In this comparison the Miller-Rabin's and the Gordon's algorithms that randomly generate primes were compared versus the SFA and the Sequences Containing Primes. The name Sequences Containing Primes algorithm is abbreviated in this thesis as 6kseq. In the interval [99000,100000] the Miller Rabin method generated 57 out of 87 expected primes, the SFA algorithm generated 83 out of 87 approximated primes. The expected number of primes was computed using the approximation n/ln(n) presented by Menezes [66]. The average consumed time of originating one prime in the [99000,100000] interval recorded 0.056 [s] for Miller-Rabin test, 0.0001 [s] for SFA, and 0.0003 [s] for 6kseq. The Gordon's algorithm in the interval [1,100000] required 100578 probes and generated 32 out of 8686 expected number of primes.

Algorithm Parametric Representation of Composite Twins and Generation of Prime and Quasi Prime Numbers invented by Doctor Verkhovsky [108] verifies and generates primes and quasi primes using special mathematical constructs. This algorithm indicated best performance in the interval [1,1000] generating and verifying 3585 variances of provable primes or quasi primes. The Parametric Representation of Composite Twins algorithm consumed an average time per prime, or quasi prime of 0.0022315 [s]. The Parametric Representation of Composite Twins and Generation of Prime and Quasi Prime Numbers algorithm implements very unique method of testing both primes and quasi-primes. Because of the uniqueness of the method that verifies both primes and quasi-primes, this algorithm cannot be compared with the other primality testing or prime numbers generating algorithms.

The ((a!)^2)*((-1)^b) Function In Generating Primes algorithm [105] developed by Doctor Verkhovsky was compared versus extended Fermat algorithm. In the range of [1,1000] the [105] algorithm exhausted an average 0.00001 [s] per prime, originated 167 primes, while the extended Fermat algorithm also produced 167 primes, but consumed an average 0.00599 [s] per prime.

Thus, the computer experiments and comparison of methods proved that the SFA algorithm is deterministic, that originates provable primes. The survey of methods and analysis of selected approaches indicated that the SFA sieve algorithm that sequentially generates primes is computationally efficient, indicated better performance considering the computational speed, the simplicity of method, and the number of generated primes in the specified intervals.

COMPUTATIONALLY EFFICIENT SEARCH FOR LARGE PRIMES

by
Wieslawa E. Amber

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
In Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

Computer Science Department

August 2001

APPROVAL PAGE

COMPUTATIONALLY EFFICIENT SEARCH FOR LARGE PRIMES

Wieslawa E. Amber

Dr. Boris S. Verkhovsky, Thesis Advisor — Date
Professor of Computer Science Department, NJIT

Dr. James A. M. McHugh, Graduate Chairman — Date
Chairman of Computer Science Department, NJIT

Dr. Fadi P. Deek, Associate Professor of Information Systems — Date
Associate Dean of Computer Science Systems, NJIT

BIOGRAPHICAL SKETCH

Author:	Wieslawa E. Amber
Degree:	Master of Science
Date:	August 2001

Undergraduate and Graduate Education:

Master of Science in Computer Science
New Jersey Institute of Technology, Newark, NJ, 2001

Bachelor of Science in Mechanical Engineering
New Jersey Institute of Technology, Newark, NJ, 1995

Major: Computer Science

ACKNOWLEDGEMENT

I am very grateful for having Professor Boris S. Verkhovsky, Ph.D. as the thesis advisor and for being his student. I have been gratified for the valuable and countless advice, encouragement, information, support, and suggestions I received from Professor Verkhovsky, that helped bring to life this project.

My sincere words of appreciation are addressed to Dr. James A. M. McHugh, Graduate Chairman and to Dr. Fadi P. Deek, Associate Dean of Computer Science Systems for actively participating in my committee. I wish to thank the Professors and Faculty Members who taught me fundamentals to make this thesis possible. I appreciate the assistance and help I received from the CIS department staff over the years.

I would like to thank Ms. Clarisa Gonzalez-Lenahan for her assistance.

TABLE OF CONTENTS

Chapter **Page**

TABLE OF CONTENTS

Chapter **Page**

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Objective

The objective of this thesis is to survey the methods for primality testing, for generating prime numbers, to compare the efficiency of selected algorithms, and indicate the best performing algorithm. Initially this paper will review, survey, and compare different approaches. In the next step the performance of algorithms will be tested through the series of computer experiments. Uniformly designed computer programs will test algorithms in the specified interval [smallest,largest] and report the number of probes, number of generated primes, CPU time, and the average time required to generate one prime in the specified interval. Results obtained from the computer experiments will serve to compare the efficiency of algorithms in the specified interval using the time complexity, number of probes, and number of generated primes as comparison criteria. The most efficient algorithm will be indicated from the tabulated results furnished in the form of comparison tables.

1.2 Background Information

This subchapter provides an introduction to the survey of methods. This subchapter also lists and organizes different approaches, formulas, and techniques used in primality testing and prime numbers generation. Menezes [66] summarized primality testing criteria and outlined the following fundamentals;

1) Each test should prove that the candidate is a prime or a weaker test should verify the result that n is a probably prime.

2) If a method provides result such that the candidate is a probably prime then this approach is called a probabilistic primality test, or compositeness test.
3) If the applied technique that serves to determine the primality of the candidates does not employ random numbers then the approach is called deterministic and the results are reproducible, otherwise the technique is called randomized.

Menezes [66] divided primality field in the two distinct categories primality testing and prime number generation. Probabilistic approach and provable approach are the two fundamental methods applied in primality testing. The test that proves that the candidate is a prime is called true primality test or a provable test. If the test provides an outcome that furnishes weaker results, then the approach is called probabilistic. The prime number generation methodology introduces two approaches deterministic and randomized. The use of randomness is applied to judge the technique. If the candidates are generated through the randomly input technique, then the method is called randomized. Otherwise if the method does not use the random input, then the approach is called deterministic and the results are reproducible. The primality test that proves candidate's primality is called provable primality test. A test that establishes weaker results is called probabilistic.

The Fermat's and Euler's theorems provided foundation to numerous new and revised approaches in primality testing and prime number generation. The Fermat's Little Theorem was converted by Edouard Lucas into a formal primality test. The Fermat, Solovay-Strassen, and earlier Miller-Rabin tests are the examples of probabilistic approaches. The advanced Miller-Rabin algorithm for generating prime numbers is considered to be provable and randomized technique. The factorization method is considered to be a true primality test. True primality tests can determine with mathematical certainty whether the candidate is a prime. The newest techniques include the application of prime factorization [99] methods on a quantum computer as presented by Peter W. Shor. The Pocklington's theorem requires partial factorization of n-1. The Jacobi sum test requires testing sets of congruences in cyclotomic rings and the algorithm simulates behavior similar to the polynomial algorithm, since the exponent ln (ln (ln (n))) serves as a constant for the certain range of n values. Menezes [66] provides three trivial examples of prime number generation methods this includes the generation of probable primes, generation of strong primes, and generation of provable primes. Random search for a provable prime is exemplified by improved Miller-Rabin Algorithm. Gordon's algorithm represents a strong prime generation method, while Maurer's algorithm produces provable primes. Haas [45] developed a multiple prime random number generator utilizing the randomized approach.

The factoring techniques found application in factoring algorithms by addition, subtraction, and factoring with sieves. The Mairson's algorithm [62] presents an example of multiplicative sieve. The Sieve of Eratosthenes provided foundation to different sieving methods such as segmented sieve, linear sieve, sublinear additive sieve, linear segmented sieve, and other sieving techniques. The Pritchard's algorithm [83] exemplifies a sublinear additive sieve for finding prime numbers, while the Pritchard's publication [84] describes the wheel sieves, and the segmented wheel sieves techniques.

On the distribution of prime numbers Menezes [66] indicated that the number of primes in the specified interval $[2,x]$ is nearly equal to the quotient of upper bound x divided by ln (x.). On the distribution of prime numbers method will be utilized in estimating the number of generated primes. The Dirchelet theorem says that if the gcd of n and a is equal 1, then the number of primes congruent to a modulo n is infinite. For example, if a function $\pi(x, n, a)$ denotes a number of primes in the interval $[2,x]$ and these numbers are congruent to a modulo n, where gcd (a, n) is equal to 1, then, $\pi(x, n, a)$ is nearly equal to x divided by ($\phi(n)$ ln x). Prime numbers are uniformly distributed among the $\phi(n)$ congruence classes in Z_n* for any value of n, where Z_n denotes a set. Describing the approximation of prime numbers Menezes [66] states that if p_n is denoting the nth prime number that results in p_n being nearly equal to a product of n and ln (n). The product of n ln (n) is less than p_n where p_n is less than a product n*(ln n + ln ln n) for $n \geq 6$.

Thus, this thesis will group the primality testing algorithms in two categories: probabilistic primality tests, and the true primality tests. The true primality tests will be further broken into deterministic tests, randomized tests, and provable primality tests. Primality tests for the elliptic curves will be placed in the group of true primality tests. The primality tests that use the factorization of n-1 method will represent the provable primality tests. The prime number generation algorithms will be divided accordingly to Menezes [66] in two categories deterministic and randomized. Further these approaches will be refined in three categories; generating provable primes, generating strong primes, and generating probable primes. Miller-Rabin random search for primes exemplifies probable primes. These methods are described in details in the chapters 2 and 3.

CHAPTER 2

PRIMALITY TESTING ALGORITHMS

2.1 Problem Statement

A. J. Menezes, [66] classified algorithms for primality testing and provided fundamental details pertaining to the presented algorithms. New algorithmic ideas have been emerged and have been implemented since the Menezes book was published. Thus, using Menezes classification in this chapter the algorithms will be sorted, placed into appropriate categories, and compared, to indicate the best performing algorithm. The time complexity, the number of generated primes, and the number of probes will be used in the comparison criteria to indicate the most efficient algorithm.

2.2 Previous Works

Menezes [66] delineated generalized framework for Probabilistic Primality Testing, which is included in the Appendix A. Probabilistic approaches for primality testing are exemplified in the Fermat, Solovay-Strassen, and Miller-Rabin methods.

Menezes summarized the properties of Probabilistic Primality Tests in the general form, that for each odd positive integer n a set W(n) is a proper subset of Z_n such that the listed properties hold. The number a, that is an element of Z_n can be defined in the deterministic polynomial time if the number a is an element of W(n). If number n is a prime, then the set is empty and W(n) is equal to 0. If the number n is a composite, then an element in W(n) is greater or equal to number n divided by 2. If the number n is a composite, then the elements of W(n) are called witnesses to the compositeness of n, and the elements of the complementary set L(n) that are equal to Z_n in W(n) are called liars

An element a of Z_n is chosen at random, and it is tested if a belongs to the W(n) subset. The test will output composite if a is an element of W(n), and will output prime if a does not belong to subset W(n). Menezes [66] defines an integer n as a probable prime, if based upon a probabilistic primality test the number n is believed to be a prime.

2.2.1 Probabilistic Primality Tests

The Fermat's theorem referenced in Menezes [66] upholds, if the number n is a prime and the number a is an integer, such that a is greater then or equal to 1, but smaller than or equal to n-1, then a^(n-1) will be equal to 1 (mod n). The Fermat's theorem endorses, that the odd, composite integer n is called Fermat's witness to compositeness for n, if an odd, composite integer a is a pseudoprime to the base a, and if the number a raised to the power of n-1 is equal to 1 (mod n), then the integer a is called Fermat's liar to primality for n.

Brillhart, Tonascia, and Weinberger [18] in collaboration with D.H. and Emma Lehmer conducted a search for odd solutions of N for the congruence a^(N - 1) equal to 1 (mod N^2). Basic scheme developed by Brillhart, Tonascia, and Weinberger is included in the Appendix A.

The Solovay-Strassen Test outlined in Menezes [66] implements Euler's criterion. In this algorithm the number n is an odd prime and the number a raised to the power of (n-1) divided by 2 is equal to (a/n) mod n for all integers a, that satisfy the gcd (a,n) equal to 1. The quotient (n / a) denotes a Jacobi symbol and (n / a) is equal to Legendre symbol if n is a prime. The number a is called Euler witness for compositeness if a satisfies one of the following categories. If n is an odd composite integer and a is an integer in the interval [1, n-1], then the number a is Euler's witness for compositeness. If gcd (a, n) is greater than 1, or if the number a raised to the exponent (n-1)/2, is not equal to (a/n) mod n, then the number n is called Euler's pseudoprime to the base a. The number n is called Euler's pseudoprime to the base a only, if gcd (a, n) is equal to 1, and if a raised to the power (n - 1)/2 is equal to (a/n) mod n. If n is an odd composite integer, then ϕ(n)/2 for all the numbers a, are called Euler's liars. Where a is defined by [1, n-1].

The Miller-Rabin_probabilistic primality test presented in Menezes [66] is considered to be a strong pseudoprime test. In this delineation n is an odd prime and n-1 is equal to 2 to the power of s and multiplied by r, where r is an odd integer. If gcd (a, n) is equal 1, then base a raised to the power of r is equal to 1 (mod n) or for j in the range of [0, s-1] base a raised to the power of 2^j and multiplied by r is equal to -1 (mod n). The number n is called a strong pseudoprime to the base a, if a raised to the exponent r is not equal to 1 (mod n), or a raised to the power of (2^j)*r is not equal to -1 (mod n), for all j numbers in the range of [0,s-1]. The integer a is also called a strong liar to primality for n. Base a raised to the power of r is equal to 1 (mod n). Base a raised to the power of (2^j)*r is equal to -1 (mod n). If n is not equal to 0, then the number of strong liars for n is equal to ϕ(n)/4. If n is an odd composite integer, then 1/4 of all the numbers a that refer to [1,n-1] are strong liars for n.

Davenport [31] referred the Rabin's algorithm as widely adapted in the computer algebra and for primality testing purpose. Davenport suggested essential strengthening to Rabin's algorithm, and correlated to Fermat-Euler theorem. (Appendix A)

Gupta, et al, summarized Solovay-Strassen [103] probabilistic algorithm for primality testing. If x and n are relatively prime then computing x^((n - 1)/2) (mod n) and the Jacobi symbol (x/n) can be accomplished in logarithmic time. If n is prime then Z_n is cyclic and x^((n - 1)/2) ≡ (x/n) (mod n). Thus, when n is a prime, no x will be certified as witness, if n is composite according to Solovay and Strassen then the set of false witnesses, the numbers that violate the conditions (2) and (3) will form a proper subgroup Z_n. According to Gupta, et al the cardinality of the Z_n subgroup can be equal at most to (n - 1)/2. Solovay & Strassen [103] in their work demonstrated that the composites were recognizable in the random polynomial time. Similarly to Rabin's algorithm, the Solovay-Strassen probabilistic primality tests essentially performed probabilistic search for a proof of compositeness. Failure provides a proof that the number n is not a composite. This algorithm will always terminate in polynomial time on input. Upon termination, a certificate will be issued that the number is either a composite, or a probably prime. Although probably prime provides high probability, but there is no guarantee that the number n is a prime.

Adleman and Huang applied the Solovay-Strassen [103] results pertaining to probabilistic algorithm in the primality testing. The Adleman and Huang method was summarized by Gupta, et al. In the Solovay and Strassen algorithm x is an element of {1,..,n - 1} to be a witness to compositeness of n, if gcd (x, n) is greater then 1, or x raised to the exponent of ((n - 1)/2) (mod n) is not equal to (x / n). If x and n are relatively prime then computing x raised to the power of ((n - 1)/2) (mod n) and the Jacobi symbol (x / n) can be accomplished in logarithmic time. Gupta, et al indicated that if n is a prime then Zn is cyclic and x raised to the power of ((n - 1)/2), is indeed equal to (x/n) (mod n). Gupta, et al concluded that, when n is a prime, no x will be certified as witness, if n is composite then according to Solovay and Strassen the set of false witnesses, specifically the numbers that violate the conditions (2) and (3) will form a proper subgroup of Z_n. According to Gupta, et al the cardinality of the Z_n subgroup can reach at most (n - 1)/2.

Rabin [87] introduced algorithm for fast, randomized primality testing. The probabilistic primality tests essentially performed a probabilistic search for a proof of compositeness. Failure provided the evidence that the number was not a composite. This algorithm will always terminate in polynomial time on input. Upon termination, a certificate will be issued that the number is either a composite, or a probably prime. Probably prime provides high probability, but not a certainty that the number is a prime. Under the extended Riemann hypothesis and probabilistically by Rabin [88] Miller developed Analysis of Prime Testing Algorithm..

Davenport and Smith [28] suggested that the results of Rabin's algorithm indicate a single iteration applied to a number N, and that N has a probability of 0.25 claiming N as a probably prime. A proof of Group Theory View presented that 10 elements in the set of N provides a probability of less than 1 in 10^6 of giving wrong answer.

The tests results of Damgard and Landrock [29] that referred to Rabin's algorithm in which a single iteration applied to a number N has a probability of 0.25 claiming N as a probably prime. Damgard and Landrock proved that for 256-bit integers, in the 6 tests a probability of less than 2^(-51) of providing wrong answer is possible. The number of primes tested should be proportional to log N. The constant of proportionality in the explicit tests for the numbers is provided in the form (K+1)*(2K+1).

Rabin [88] introduced algorithm for fast, randomized primality testing. The probabilistic primality tests essentially perform a probabilistic search for a proof of compositeness. Failure provides evidence that the number is not a composite. This algorithm always will terminate in a polynomial time on input. Upon termination, a certificate is issued that the number n is either a composite, or a probably prime. While probably prime provides high probability, but not a certainty that the number is a prime.

Monier [70] compared Miller-Rabin algorithm versus Solovay-Strassen primality testing algorithm. These two described methods are considered to be the Monte Carlo variety algorithms because when n is a prime these algorithms can report only with a certain probabilistic measure of confidence, but no proof is provided.

Erdos and Spencer [36] introduced a small power packed monograph on non-constructive probabilistic methods in combinatorics. As Gupta, et al quoted, the sample space is so abundant with good points that the checking steps inherent to primality testing are dispensed with.

Lehmann [56] presented two algorithms for primality testing that referred to the extended Riemann hypothesis. Lehmann discovered an algorithm utilizing different number-theoretic properties for defining witnesses to compositenesses and primality.

Galligo and Watt [38] presented a new numerical absolute primality criterion for bivariate polynomials. Galligo and Watt test was based on a simple property of monomials appearing after a generic linear change of coordinates. Galligo and Watt method provided a probabilistic algorithm for detecting absolute factors.

Goldwasser and Kilian [41] emphasized on primality proving algorithm, a probabilistic primality test that generates short proofs of primality on the prime input. Goldwasser and Killian proved that the specified test is expected to run in polynomial time for all primes. Goldwasser and Killian test was based on a method applying a group theory to the problem of prime certification and the application of this methodology using groups generated by elliptic curves over the finite fields. This method provided an algorithm for generating large certified primes with a distribution that was statistically close to uniform. The gap between the consecutive primes is bounded by a polynomial in their size and the test yields the Las Vegas primality test.

Wunderlich [111] discussed the heuristics that will efficiently find certificates for some primes. The set of primes certifiable in this method is sparse and has not been proven to be infinite. It shows that these techniques, in much more general form are applied in the efficient generation of certificates of primality for most primes.

2.2.2 True Primality Tests

According to Menezes [66], true primality tests can determine with mathematical certainty whether a random candidate is a prime. If the input is sequential, but the primality tests can establish with mathematical certainty, that a candidate is a prime, then the technique is called deterministic and the results are reproducible.

2.2.2.1 Randomized Primality Tests. Pomerance, Selfridge, and Wagstaff [79] indicated that the primes should be presented in the form of x-values, because the value of x = 4 provides the information as the values of x = 2, and value 6 provides the information as the orders of 2 and 3 to be adjacent. Rabin's algorithm begins with a choice of a random seed x, not congruent to 0 modulo N.

Arnault [7] disputed Leech form N = (K+1) (2K+1) (3K+1) to defeat the theorem of the set of prime x-values {2,3,5,7,11,13,17,19,23,29}. According to Leech N is a Carmichael number.

Pratt's [80] short proofs of primality demonstrated that the primes are recognizable in non-deterministic polynomial time. For a known prime p, Pratt proved that p is prime by exhibiting that some g are elements of Z_p such that $O_p(g)$ is equal to p-1. To prove that $O_p(g)$ is equal to p-1 the decomposition of prime was introduced in the form p-1 is equal to p1^(e1) p2^(e2)...pk^(ek). Thus, pi's should be recursively proven primes. Algorithm was considered ineffective since it is hard to generate the decomposition of primes p-1.

Gupta, Smolka, and Bhaskar [44] approach emphasized on the conventional probabilistic and randomized algorithms. Gupta, et al presented five different techniques applying 12 randomized algorithms both sequential and distributed that reached a wide range of applications including primality testing.

In the Primality Testing the authors overviewed the following methods;

1) Ancient Chinese assertion that n is a prime if and only if n divides 2^n - 2 which appeared to be wrong according to Gupta, Smolka, and Bhaskar.
2) Schroeder's [95] article on prime numbers, their distribution, fractions, and congruences found application in the search for large prime numbers in primality testing as presented by Gupta et al. [44]. The mean distance between primes in the neighborhood of a number n is O(log n).
3) Wilson's theorem states that a number n is a prime if and only if n divides (n - 1)! + 1 without the reminder.
4) The Fermat's congruence x^(n - 1) ≡ (mod n) implies that "n divides x^(n-1)".

Thus, n can be a proven composite if n does not divide x or x^(n - 1) - 1, where x is a witnesses to the compositeness of n.

Adleman and Huang [4] invented algorithm that employed a separate Monte Carlo scheme to test for primality instead of deciding the primality. A random polynomial time algorithm exists for the set of prime numbers. Algorithm oscillates from searching for witnesses to compositeness and from witnesses to primality and eventually finding one in polynomial limited expected time. This algorithm represents the Las Vegas variety and per Adleman and Huang will never declare a composite number to be a prime or vice versa. This algorithm may not terminate in polynomial time for certain inputs.

Shanks [97] proved that for the incremented sequence, if k is equal to 1/2 log p, then the randomized algorithms of Lehmer and Shanks for computing square roots modulo a prime p have probability error that is equal to O(log p / √p).

Sipser [102] considered in his model that the random numbers are a scarce resource. To get probability error of O(p^(-1/2 + ε)) for the square root algorithm, it would be expected to have O(log p)^2 random bits. The results of the experiment implied that O(log p) is sufficient. Schoff [94] proved that the primes are recognizable in the random polynomial time.

2.2.2.2 Primality Tests Using Factorization n-1 Primality testing using factorization method n-1 is described in Menezes [66]. This method is delineated in details in the Appendix A. Let the number n to be an integer, that is greater than or equal to 3, then number n is a prime, if and only, if there exists an integer a, that satisfies three conditions: a^(n - 1) equal to 1 (mod n) and a^((n - 1)/q) is not equal to 1 (mod n) for each prime divisor q of n - 1.

The special case algorithms that represented the factorization methods are Pocklington's theorem and Jacobi sum test [66]. The Jacobi sum test is implemented in the Solovay-Strassen approach that exemplified a true primality test and that examed the sets of congruences analogous to Fermat's method [66]. The Jacobi sum test algorithm is detailed in the Appendix A.

Pocklington's theorem represents another method for proving primality by implementing partial factorization of n-1. Pocklington's algorithm presented in Menezes [66] is described in the Appendix A.

Brillhart, Lehmer, and Selfridge [18] invented an algorithm that was based on the work of Fermat and Gauss, which relied mostly on factoring and according to Goldwasser and Kilian this algorithm appeared to be impractical. Heintz and Sieveking [47] introduced a method of absolute factorization of polynomials with coefficients in a number field k, explained as factorization over the algebraic closure of k.

Carmichael_[21] presented a unique method of factorization of prime n including a proof that n can satisfy the Fermat's congruence a^(p -1) = p. Mignotte [67] surveyed the primality testing from the computational complexity perspective. Mihalescu_[68] invented a super-polynomial-time, fast algorithm based on the cyclotomy, that was designated for numbers comprised of a few thousand of digits to verify the primality proofs. Shor [100] invented a polynomial-time algorithm for prime factorization to be implemented on the quantum computer.

Mawata_[58] implemented the isomorphic image of the ring of polynomials in n variables with rational coefficients. This scheme is used to uniquely factorize the prime of positive integers to represent the multivariate polynomials. (The speed of Kronecker's trick to transform multivariate polynomials to univariate polynomials.)

Ribenboim [91] book includes generalization algorithm for primality proving tests on the factorization of N-1 where N is considered as a number to be a proven prime. According to Ribenboim there are the easy numbers for the N - 1 test, those N for which the N - 1 factorization is trivial, such as Fermat numbers N = 1 + k!, N = 1 + k*a^n,...

Bressoud and Wagon [20] focused on prime numbers in particular prime testing and certification. The application of Lucas sequences to primality testing is presented in chapter 8. Chapter 9 concerns Gaussian numbers and the decomposition of primes in sums of 2 squares. The Adleman-Manders-Miller [6] algorithm computes the q-th root modulo prime, when q has (p -1) divisor and k is equal to 1/2 log q_p

2.2.2.3 Primality Tests for Elliptic Curves._Bosma [16] introduced numbers called Elliptic Mersenne primes. These numbers found application in the Morain's article entitled Easy Numbers for the Elliptic Curve Primality Proving algorithm. Chudnovsky and Chudnovsky [24] presented a deterministic polynomial time algorithm for testing the primality applying the orders of elliptic curve.

Schoff's deterministic algorithm was designated to compute square roots modulo prime. Concurrently Schoff provides a polynomial time algorithm to compute the order of the group generated by an elliptic curve over a finite field.

The Goldwasser-Kilian [40] algorithm generated elliptic curves randomly and counts their points. This probabilistic primality test due to Goldwasser and Kilian was adopted first in the elliptic curve application and to produce a certificate of correctness for its assertion of primality. The recursive algorithm served as a model for Atkin's test and modifications. If qP denotes repeated addition of P + P +,...,+ P which probably will fail. If a failure had occurred, then the test terminated with an essential result, and p became a certificate of p's compositeness. This algorithm adopted Schoffs [83] O(log^8(p)) algorithm for computing the order of ER(a, b), where a and b were given. The following result provided the basis for a recursive call of Kaltofen-Valente Theorem:

Atkin [8] improved the Schoff's algorithm and developed a variant method in which groups and their orders are picked at the same time that runs much quicker. The Atkin's test used a concept of a complex multiplication field to compute the elliptic curve's order. This theorem certified that N is a prime provided Q is a prime.

Morain [72] presented new classes of numbers that are easier to test for primality with the Elliptic Curve Primality Proving Algorithm than average numbers. Morain presented an algorithm that generalized the Fermat-like primality proving tests based on the factorization of N -1, when N is the number to be the proven prime. An algorithm is included in the Appendix A.

The results obtained by Adleman-Huang [3] counterpart the Solovay-Strassen results considering the existence of a random polynomial time algorithm for the set of composites. In further work, Adleman and Huang [5] invented an algorithm that guaranteed to find the short certificates for all prime numbers. They correlated to Goldwasser and Kilian [40] analysis to set limits above the fraction of k-bit primes. The elliptic curve based algorithms could not certify quickly down to 2 to the power of -Ω(k). Adleman and Huang algorithm takes more than O(k^11) expected time on most k-bit primes.

2.2.2.4 Deterministic Primality Tests. Miller [69] demonstrated that the Riemann hypothesis for Dirichlet L-functions implied that the primes were decidable in deterministic polynomial time. Miller's algorithm exemplified a deterministic polynomial time procedure. As Miller indicated the declarations of primality made by the algorithm are always correct if the ERH (Extended Riemann Hypothesis) is true. If the ERH is false then the numbers declared as primes are composites. Thus, the ERH is not used to constrain the running time of the algorithm, but to test the correctness of the answer

Gupta, et al, provided that there were few heuristic methods for perfect hashing. The discussion on these methods is provided by Melhorn.[65] Melhorn demonstrated that a program of length $O((n^2)/(m + \log \log N)$ computes a perfect hash function for a given set $S \subseteq U$. To find an actual perfect hash function a family H of hash functions is presented $H = \{h_\kappa \mid h_\kappa(x) = (k*x \bmod N) \bmod m;$ where $1 \leq k < N\}$. Melhorn stated that without loss of notion, if $U = [0,..,N - 1]$ represents the totality of the keys with N prime. Primality of N can be achieved by adding nonexistent keys to U. As Melhorn uttered the resulting universe will not be substantially larger than the original U since prime numbers are sufficiently dense.

Cohen and Lenstra [26] provided nearly polynomial-time deterministic primality tests that do not relay on any unproven assumptions. The test required $k^{(\theta(\ln \ln k))}$ computational steps on an input N of length k. These tests did not provide any succinct primality proof related to the number that is declared as a prime.

Gupta, et al, outlined Miller's deterministic algorithm for primality testing.[69] Gupta indicated that Miller divided the composite numbers applying the following function $\lambda'(n) = \mathrm{icm}\{(p1-1),..,(pm-1)\}$ and paired the set of composite numbers to satisfy the Fermat's congruence. Miller did not apply the Carmichael λ function. The framework of the Miller's algorithm is presented in the Appendix A. Gupta, et al, outlined that Rabin's probabilistic algorithm for primality testing.[88] demonstrated that more than 1/2 of the values of x were elements of {1, 2, 3,..,n - 1} that satisfied the (2) and (3) conditions of the algorithm if n was certainly composite.

Konyagin [52] invented a deterministic polynomial time algorithm that proved primality for infinite sets of primes. Kronsjo [53] described the probabilistic algorithms and Rabin's algorithms for primality.

Kaltofen, Valente, and Yui [49] presented modifications to Goldwasser-Kilian-Atkin primality test. If n is an input, then the output is either prime or composite, and along with an output a certificate of correctness is issued that is verified in polynomial time. This modification substitutes the root of the Hilbert's class equation with the root of Watson's class equation for $Q(\sqrt{-D})$ and reconstructs to a root of the corresponding Hilbert equation. The Watson class equations have very small coefficients comparing to those of their Hilbert counterparts.

Miller [69] demonstrated that the Riemann hypothesis for Dirichlet L-functions have implied that the primes were decidable in deterministic polynomial time. Miller algorithm is a deterministic polynomial time procedure. The assertions of primality made in the algorithm are always correct if the ERH (Extended Riemann Hypothesis) is true. If the ERH is false then the numbers declared primes still could be composites. Thus, the ERH is not used to bind the running time of the algorithm, but to test the correctness of the answer. Pratt [80] used Lucas-Lehmer heuristic for primality testing to demonstrate that a succinct proof of primness of number n can be verified in O (log n) lines.

The algorithmic Schoff's [94] ideas provided foundation to Adleman and Huang [3] to prove the hypothesis, that the primes are recognizable in a random polynomial time. The Schoff's deterministic algorithm was intended to compute square roots modulo prime. Concurrently Schoff provided a polynomial time algorithm to compute the order of the group generated by an elliptic curve over a finite field.

Doctor Verkhovsky [108] algorithm verifies and generates primes and quasi primes using special mathematical constructs to verify the primes or quasi primes. Algorithm provided in the Appendix A. In the range [1,200] this algorithm generates and verifies 84 combinations of provable primes or quasi primes, while an average time per prime 0.0119048 [s]. In the range [1,1000] this algorithm generates and verifies 3585 variances of provable primes or quasi primes, while an average time per prime 0.0022315 [s]. In the range [1,10000] this algorithm generates and verifies 4492 variances of provable primes or quasi primes, while an average time per prime 0.0164737 [s].

Algorithms invented by Doctor Verkhovsky [106] and [107] that are particularly designated to generate prime numbers in the desired interval and described by functions m=12k+r, or m=6k+1 are using modulo arithmetic to verify the primness. The SFA and the 6kseq algorithms' performances are described in details in chapter 3. Chapter 3 compares various algorithms for generating prime numbers.

2.3 Algorithms' Performance Evaluation

Tables 1, 2, and 3 in the Appendix C provide the results pertaining to the Fermat's, Miller-Rabin's, and the Solovay-Strassen primality tests. The results of the computer experiments indicate that the number of probes in the Fermat's test is the largest among the three tested algorithms. While Fermat's, Miller-Rabin's, and Solovay-Strassen's algorithms are constructed using exponentiation, the Doctor Verkhovsky algorithms are simplified to the incremental multiplication and modulo arithmetic. The Fermat's, Miller-Rabin's, and Solovay-Strassen's algorithms are designated to test the numbers for primality.

The SFA, Sequences Containing Primes (also known as 6kseq) algorithms are designated for generating prime numbers in the specified intervals. The Parametric Representation of Composite Twins and Generation of Prime and Quasi-Prime Numbers algorithm's principal function is to verify the primality, while generating prime and quasi prime numbers. Thus, the algorithm named Parametric Representation of Composite Twins and Generation of Prime and Quasi Prime Numbers indicated best performance in the interval [1,1000] generating and verifying 3585 variances of provable primes or quasi primes, while an average time per prime, or quasi prime was 0.0022315 [s].

2.4 Summary

The Primality Testing algorithms are divided in two categories; probabilistic primality tests, and the true primality tests. The true primality tests are further broken into deterministic tests, randomized tests, and provable primality tests. Primality tests for the Elliptic Curves represent the true primality tests. The primality tests that use the factorization of n-1 method represent the provable primality tests.

The results from the computer experiments indicated that the number of probes in the probabilistic Fermat's test is the largest among the three tested algorithms. While Fermat's, Miller-Rabin's, and Solovay-Strassen's algorithms have been constructed using exponentiation. Among these three probabilistic algorithms the Solovay-Strassen indicates better computational efficiency.

Algorithm Parametric Representation of Composite Twins and Generation of Prime and Quasi Prime Numbers invented by Doctor Verkhovsky [108] verifies and generates primes and quasi primes applying special mathematical constructs to verify the primes or quasi primes. This algorithm indicated best performance in the interval [1,1000] generating and verifying 3585 variances of provable primes or quasi primes, while an average time per prime, or quasi prime was 0.0022315 [s]. This algorithm implemented very unique method of testing both primes and quasi primes that cannot be compared with the other primality testing algorithms. Results pertaining to the other algorithms developed by Doctor Verkhovsky particularly the SFA and Sequences Containing Primes (also known as 6kseq) designated for generating prime numbers in the specified intervals are described in the chapter 3.

CHAPTER 3

PRIME NUMBERS GENERATING METHODS

3.1 Problem Statement

Menezes, A.J. [66] classified algorithms for generating prime numbers and provided fundamental details pertaining to the introduced algorithms. Since the Menezes book was published, numerous advancements have been done and implemented in the area of prime numbers generation. Similarly to the chapter 2, the prime numbers generating algorithms will be classified, placed into appropriate categories, and compared, to indicate the best performing algorithm.

In this chapter, the SFA and the Sequences Containing Primes Algorithms will be compared with selected algorithms, that generate prime numbers. The comparison criteria are: the number of generated primes per interval, the number of required probes, and the average time required to find one prime in the specified interval.

3.2 Previous Work

Menezes [66] introduced following generalized approach to generate large prime numbers; Generate a candidate, a random odd number n of appropriate size, or search a sequence of candidates starting from n such as n, n+2, n+4,.., to find a prime, which will have desirable properties, called a priori. If the n candidates are specially constructed for mathematical reasoning to provide an establishment, then the method is called a constructive prime generation technique. This technique is designated for the true primality tests.

3.2.1 Random Search for Probable Primes

Algorithm Finding Square Roots modulo a prime p, which was presented in Menezes [66] exemplified randomized approach because of the method in which the quadratic non-residue b was selected in the step 2. A diverse range of algorithms for finding square roots modulo a prime p is furnished in the Appendix A.

Miller-Rabin random search algorithm presented in the Menezes book [66] generates randomly an odd k-bit integers n. Using a trial division, the numbers are pre-tested for divisibility factors. Miller-Rabin algorithm that is designated for primality testing outputs prime's certificate and returns n.

Adleman and Huang [3] approach emphasized on the existence of a random polynomial time algorithm for the set of primes. They proved that the primes are recognizable in a random polynomial time. In their analysis the authors applied the theory of Abelian varieties. Adleman and Huang [3] adopted results from Pratt, and Solovay-Strassen framework.

Pratt [80] provided short proofs of primality. Contrary to Adleman, Huang, Pomerance, and Rumely, Pratt demonstrated that the primes were recognizable in non-deterministic polynomial time. Pratt used Lucas-Lehmer heuristic for testing primness and proved that for some given prime p, that p is a prime by exhibiting property such that g belongs to Z_p such that $O_p(g) = p - 1$. To prove that $O_p(g)$ is equal to p - 1 Pratt implemented a decomposition of prime in the following form p - 1 is equal to p_1^(e_1) p_2^(e_2)...p_k^(e_k). Then, the p_i's were recursively proven primes by Pratt.

Bach [11] stated that the primality testing could be solved efficiently using a source of independent, but identically distributed random numbers. Eric Bach adopted Andre

Weil's [109] results concerning the number of points on algebraic varieties over the finite fields. Bach's method used iteratively generated sequences applying functions f(x) = αx + β (mod p) to the randomly chosen set of x and estimating the probability that an algorithm will fail. Bach considered 2 bounds; finding square roots modulo a prime p, where probability of failure is O (log (p /√ p)), and testing p for primality, where probability of failure is O(p^(-1/(4 + ε))) for any $\varepsilon > 0$. The number of Bach's trials in the cases 1) and 2) counted approximately 1/2 log p. Function f produces randomly a sequence x f(x), f^2(x),...,f^(k-1) (x); where x denotes a seed and k quantities denote random input to the algorithm successive trials. Analysis of the Square Root Algorithm by_Bach [11]_outlines the analysis due to the Lehmer's modular square algorithm, included in the Appendix A.

Lehmer [58] introduced a Binomial Congruence Algorithm where_x^q ≡ a (mod p) represents randomized method for solving congruences that was presented by Adleman-Manders, and Miller [6], where q denotes a prime divisor of p - 1. If q is equal 2 then it reduces to another square algorithm published by Shanks [96].

The Adleman and Huang [3] article emphasized on the existence of a random polynomial time algorithm for the set of primes. Adleman and Huang proved that the primes were recognizable in the random polynomial time.

Chang [22] presentd application of hash functions in the form of h(x) equal to C mod p(x) where C stands for an integer constant and where p(x) generates different prime for each introduced integer x. Author does not provide general method for finding p(x).

Cormen, Leiserson, and Rivest [27] encyclopedic introduction to algorithms covered a large diversity of randomized algorithms including primality testing.

The first part of Kilian's [50] dissertation described randomized algorithm that is designated to generate large prime numbers, which have short easily verified certificates of primality. This algorithm provides deterministically verifiable proofs of primality but vanishing fraction of prime numbers.

Lehmer [58] invented modular square root algorithm. This algorithm provided foundation to Eric Bach analysis on randomized algorithms. If a is equal to a nonzero square modulo p, then the randomized algorithm will compute $b = \sqrt{a}$ (mod p), this algorithm requires O(log p)^3 steps and is successful with 1/2 probability.

Blum, Blum, and Shub [15] introduced the cryptographicaly secure pseudo-random generators that produce high quality pseudo-random bits from a random initial seed in a deterministic process. The drawbacks are that setting up the generator is expensive to obtain large primes. The Blum et al, method requires one multi-precision multiplication per random bit. The randomness of these numbers is conjectured relying on the computational difficulty related to integer factoring problem.

Goldwasser and Killian suggested that Cramer's conjecture on gaps between primes implied that the primes were recognizable in random polynomial time. Goldwasser and Kilian indicated that in reference to the Prime Number Theorem for sufficiently large x, $\pi(x) \rightarrow (x / \log x)$, where the best known bound for the maximal gap between two primes is given by Heath-Brown [46]and Iwaniec [48] that for sufficiently large x, there always exists a prime in the interval [x, x + x^(11/20]. The density of primes in the small intervals was described by Heath-Brown [41] as follows for the integers a, b where #p[a,b] denote the number of primes x satisfying $a \leq x \leq b$. There exist constants c_1, c_2 such that for sufficiently large x, the number of intervals $[y, y + \sqrt{y}]$ where $x \leq y \leq 2x$ in which there are less than ($c_1\sqrt{y}$)/(log y) primes is less than x^(5/6) (log^(c_2) x). Haas [45] developed a multiple prime random number generator and in his article presented a partial Fortran code for this generator.

3.2.2 Deterministic Approach

Algorithm Finding Square Roots modulo a prime p yields a simple deterministic algorithm for a special case where $p \equiv 3 \pmod 4$, $p \equiv 5 \pmod 8$. Menezes indicates that computing square roots in a finite field can be extended to find square roots in any finite field F_q of an odd order q = p^m, where p is a prime and $m \geq 1$. Each element a $\in$ F2^m has exactly one square root, specifically a^(2^(m - 1)).

Adleman, Pomerance, and Rumely [2] introduced a method, that is based on the existence of c where c is an element of N, such that the primes are decidable in deterministic polynomial time O ((log n)^(c log log log n)). Thus, for all r that belong to N, if | r denotes the length of r when written in binary form, then for all c, l, r that represent the elements of N, there exists such c that is equal to the SLICE (l, r).

Dressler and Parker [34] presented two theorems. Theorem 1 by Reichert [89] stated that every integer greater than 6 can be represented as a sum of distinct primes. The theorem 2 by Dressler, Makowski, and Parker [35] considered that every integer greater than 1969, 1349, 1387, 1475 can be presented as a sum of distinct primes in the form 12n+1, 12n+5, 12n+7, 12n+11, respectively. Dressler and Parker [34] introduced a subscript to present p[i] as the ith prime and q[i] represented a very thin subsequence of the primes defined by q[i] = p[p[i]]. They concluded that every integer greater than 96 can be represented as a sum of distinct members of sequence equal to {q[i]}.

Goldwasser and Kilian[40] demonstrated that for given prime p of length k, the algorithm outputs a certificate of correctness of size O(k^2), verified in O(k^4) deterministic time.

Kuo and Chou [54] Article presented fast algorithm to generate essential primes without generating a prime cover of the Boolean function. A condition for detecting essential primes for a Boolean function with multiple-valued inputs. The detection of essential primes is performed with application of a tautology-checking algorithm.

Applying Exponential Function ((a!)^2)*((-1)^b) in Generating Primes [105] developed by Doctor Verkhovsky and comparing versus extended Fermat algorithm, that generates primes in the specified sequential interval provided following results. Extended Fermat algorithm was constructed from original Fermat algorithm adding a function that generates primes in the specified interval and fuction that tests generated probable primes for possible divisibility factors. The number of generated primes was verified in both algorithms applying divisibility test.

3.2.3 Generating Probable Primes

Goldwasser and Kilian_[40] publication emphasized on a new probabilistic primality test, which was different from Miller [68], Solovay-Strassen [103], and Rabin [87]. The results of the Goldwasser-Kilian test implicated that there exist infinite sets of primes, which can be recognized in expected polynomial time, and that the large certified primes can be generated in expected polynomial time.

3.2.4 Generating Strong Primes

Menezes [66] described that a number p is a strong prime, if the integers r, s, and t satisfy the three outlined conditions. A number p-1 has a large prime factor denoted r, p+1 has a large prime factor denoted s, and r-1 has a large prime factor denoted t. Primes s and t are generated in step 1 applying Miller-Rabin test. Each candidate is pre-tested for primality in steps 2 and 4. Gordon's algorithm exemplifies a method of generating strong prime.

3.2.5 Constructive Techniques for Provable Primes

Maurer's algorithm provided in Appendix A exemplifies a technique for generating provable primes. The Maurer's algorithm originates random provable primes that are almost uniformly distributed over the set of all primes of a specified size. Maurer's algorithm utilizes Pocklington's theorem.

3.2.6 Sieves

Doctor Verkhovsky developed two methods for generating primes using two functions m=6k+r, or m=12k+r to originate prime numbers. These two algorithms can be implemented using few different programming techniques, languages, to be coded in the C for UNIX, or even in the assembly language. Running programs on the layer closest to the kernel level increases the processing speed while generating very large numbers. Both methods produce provable primes. The algorithm Sequences Containing Primes [106] generates a set of primes in the specified interval of odd numbers only [Smlst, Lrgst]. The algorithm Sequences Containing Primes utilizes function m=6k+1 or m=6k-1 and eliminates multiples of k or 5k-1. The SFA algorithm [107] generates a set of primes or quasi-primes on the specified interval [Smlst, Lrgst]. Using two functions 12k + r, applying four sieves, and modulo arithmetic composites are eliminated and remaining primes are printed.

3.2.6.1 Sieve of Eratosthenes. Pettofrezzo and Byrkit [75] presented properties of prime numbers (sieve of Eratosthenes, prime number theorem, twin primes, Goldbach's conjecture), and primes factorization

Pintz, Steiger, and Szemeredi [76] article documents that there exists an infinite set of primes whose membership can be tested in the polynomial time. For every n $\in$ Q, a certificate of length O(log n) is produced at random, which can be verified in deterministic time O((log n)^3).

Xuedong Luo [113] invented a sieve algorithm that has a complexity coherent to Eratosthenes' sieve rendering more practical significance. Xuedong Luo in his paper refered to Eratosthenes algorithm for finding primes between [2,..,*N*], Mairson [62] primal sieve that executes in linear time, Gries and Misra [43] linear multiplicative sieve, and Pritchard's [83] method that improved multiplicative sieves and modified Mairson's algorithm. Xuedong Luo delineated an algorithm of the Eratosthenes' complexity with more practical significance.

3.2.6.2 Multiplicative Sieve. Giblin [39] presented primes by multiplication method that is based on elimination of composites. Method is included in the Appendix A.

Pritchard [83], presented comparison of various sieves, a method for improving multiplicative sieve and modified Mairson's algorithm to attain the O [N/log logN] additive sieve. Mairson [62] introduced an algorithm that uses a doubly linked list and pointers. All the non-primes are eliminated from the list by deletion using the sub-procedure crossoff.

3.3 Algorithms' Performances Evaluation

To compare the prime numbers generating algorithms and to provide the desired results, the C++ computer programs were written for the selected algorithms. The results from the computer experiments pertaining to the specified algorithms are furnished in the tables. Each table provides the lower and the upper interval limits, the CPU time, the number of probes required to generate the primes in the specified range, the number of primes generated in the specified interval, and the average CPU time to generate one prime in the specified interval. The CPU time is the time captured while generating primes in the specified interval.

The following algorithms have been selected for the comparison of performance; Gordon's algorithm for finding strong primes and Miller-Rabin algorithm for finding random primes. Among the selected sieve algorithms were Xuedong Luo's algorithm 1, Gries-Misra's algorithm, Pritchard's algorithm, conventional sieve of Eratosthenes, and extended Fermat's algorithm. While the Xuedong Luo's, Gries-Misra's, Pritchard's, and sieve of Eratosthenes algorithms are array based algorithms and their performance is limited by the size of array. Tables 17 and 18 summarize the results and compare the methods in the specified range of limits.

Considering the number of probes and the average required time to generate one prime in the specified interval, the Gordon's algorithm in the range [1,10000000] generated 29 random primes, the average time spend per prime 0.000285 [s], and required 1000512 probes. The Doctor Verkhovsky algorithm described by a function m=6k +1 in the range of [9999900,10000000] required 4752 probes, generated 3 primes. Comparing the algorithms in the range of [999000,1000000] the Miller-Rabin produced 55 random primes, required 9000 probes, and consumed an average time per one prime 0.378 [s]. The SFA produced 84 primes, required 142897 probes, and consumed an average time of producing one prime 0.013 [s], while the 6k+1 algorithm generated 14 primes, required 14028 probes, and required an average time of 0.009 [s] per one prime. Considering that the number of generated primes in the specified interval is described by the formula n/ln(n) and comparing the computed versus the number of generated primes, the obtained results are as follows. The Gordon's algorithm in the range [1,10000000] generated only 29 of 620421 estimated primes, while an average time of producing one prime took 0.0002 [s]. Comparing the generated number of primes versus estimated (computed) number of primes in the interval [99000,100000] the Miller-Rabin algorithm generated 57 out of 87 estimated, the SFA generated 83 out of 87, and the 6kseq generated 23 out of 87. In the range [9000,10000] the Miller-Rabin produced 80 out of 109 primes, required an average 0.026 [s] per prime. SFA generated 90 out of 109 primes and required an average 0.00001 [s] per prime. The 6kseq provided 28 out of 109 primes and an average time of originating one prime reached 0.00003 [s] per prime. In the range [9900,10000] Miller-Rabin produced 5 primes, while SFA produced 10 primes.

The application of Factorial Function ((a!)^2)*((-1)^b) in Generating Primes method [105] developed by Doctor Verkhovsky was compared versus extended Fermat algorithm, that generates primes in the specified sequential interval. The Factorial

Function ((a!)^2)*((-1)^b algorithm provided best results in the lowest interval [1,1000]. In the range [1,1000] factorial algorithm exhausted an average 0.00001 [s] per prime, originating 167 primes, while extended Fermat algorithm also produced 167 primes and consumed an average 0.00599 [s] per prime. In the range [1,10000] both algorithms generated 1228 primes, the factorial algorithm consumed an average 0.00651 [s] per prime, while extended Fermat algorithm consumed an average 0.01465 [s] per prime. In the range [1,20000] both algorithms generated 2261 primes, factorial algorithm consumed an average 0.01371 [s] per prime, while extended Fermat algorithm consumed an average 0.02307 [s] per prime.

3.4 Summary

The Miller-Rabin algorithm requires smaller number of probes because the algorithm generates random numbers in the specified interval, pretests the randomly generated numbers for possible divisibility factors and if the number did not pass the test then it is rejected.

Thus, the number of probes in the Miller-Rabin method is significantly reduced in the first probe. The rate of generated primes against the existing number of primes in the specified interval places the Miller-Rabin approach after the SFA algorithm. The number of required probes in the SFA algorithm was reduced in the first loop to the difference between the upper and lower limits divided by 12, plus the number of iterations in four sieves. The SFA algorithm generates more primes per interval and the computational speed per prime is faster than the Miller-Rabin algorithm, because the non-primes are eliminated sequentially and in parallel method applying two equations m=12k+r.

Thus, the SFA algorithm has advantage over the Miller-Rabin's algorithm, which utilizes the random search technique for finding primes, and over the Gordon's algorithm, which randomly generates strong primes. As the table 18 indicates, comparing the number of generated primes vs. the number of expected primes, Gordon's algorithm indicates large inefficiency. Gordon's algorithm similarly to Miller-Rabin's method eliminates the non-primes in the first step by pre-testing the randomly generated numbers for possible divisibility factors.

While comparing the Miller-Rabin and SFA algorithms in the range of [999000, 1000000] the SFA was still found as a very efficient algorithm. The Miller-Rabin produced 55 random primes, required 9000 probes, and consumed an average time per one prime 0.378 [s]. The SFA produced 84 primes, required 142897 probes, and exhausted an average time per one prime 0.013 [s], while the 6kseq algorithm generated 14 primes, required 14028 probes, and required an average time of 0.009 [s] per one prime.

Applying Factorial Function ((a!)^2)*((-1)^b) in Generating Primes approach [105] developed by Doctor Verkhovsky provided best results in the lowest interval [1,1000]. In the range [1,1000] the factorial algorithm exhausted an average 0.00001 [s] per prime, originated 167 primes, while the extended Fermat algorithm produced 167 primes, but consumed an average 0.00599 [s] per prime.

Thus, among the selected algorithms the SFA algorithm indicated the best performance considering the computational speed, the simplicity of method, and the number of generated primes in the specified interval.

CHAPTER 4

CONCLUSIONS

The Primality Testing algorithms are grouped in two categories: probabilistic primality tests, and the true primality tests. The true primality tests are further broken into deterministic tests, randomized tests, and provable primality tests. Primality tests for the Elliptic Curves represent the true primality tests. The primality tests that use the factorization of n-1 method represent the provable primality tests.

The results of the computer experiments indicated that the number of probes in the probabilistic Fermat's test was the largest, among the three tested algorithms. The Fermat's, Miller-Rabin's, and Solovay-Strassen's are probabilistic algorithms. Among these three techniques the Solovay-Strassen approach indicates better computational efficiency.

The results from the computer experiments pertaining to the specified algorithms are furnished in the tables. Each table provides the lower and the upper interval limits, the CPU time, the number of probes required to generate the primes in the specified range, the number of primes generated in the specified interval, and the average CPU time to generate one prime in the specified interval. The CPU time is the time consumed to generate all primes in the specified interval. Selected algorithms for the comparison were sieve algorithms, Gordon's algorithm for finding strong primes, and Miller-Rabin algorithm for finding random primes. The selected sieves algorithms were Xuedong Luo's algorithm 1, Gries-Misra's, Pritchard's, conventional sieve of Eratosthenes, and extended Fermat's algorithm. While the Xuedong Luo's, Gries-Misra's, Pritchard's, and sieve of Eratosthenes algorithms are array based algorithms and their performance is limited by the size of array. Tables 17 and 18 summarize the results and compare the methods in the specified range of limits.

Considering the number of probes and the average required time to generate one prime in the specified interval, the Gordon's algorithm in the range [1,10000000] generated 29 random primes, used an average time 0.000285 [s] per one prime, and required 1000512 probes. The Doctor Verkhovsky algorithm described m=6k +1 in the range of [9999900,10000000] required 4752 probes, and generated 3 primes. Comparing the algorithms in the interval of [999000, 1000000] the Miller-Rabin test produced 55 random primes, used an average 0.378 [s] per prime, and required 9000 probes. The SFA produced 84 primes, consumed an average 0.013 [s] per prime, and required 142897 probes, while the 6k+1 algorithm generated 14 primes, used an average 0.01 [s] per prime, and required 14028 probes.

The Gordon's algorithm in the range [1,10000000] generated only 29 out of 620421 estimated primes. Comparing the generated number of primes versus approximated (computed) number of primes in the interval [99000,100000] the Miller-Rabin algorithm generated 57 out of 87 estimated, the SFA generated 83 out of 87 in the average time 0.0001 [s] per one prime, and the 6kseq generated 23 out of 87. In the range [9000,10000] the Miller-Rabin produced 80 out of 109 primes, SFA 90 out of 109 primes, and 6kseq provided 28 out of 109 primes. In the range [9900,10000] Miller-Rabin produced 5 out of 1 primes, while SFA produced 110 out of 11 primes.

The Miller-Rabin algorithm requires smaller number of probes because this algorithm generates random numbers in the specified interval and pre-tests these randomly generated numbers for possible divisibility factors. If the number did not pass the test then it is rejected. Thus, the number of probes in the Miller-Rabin method is significantly reduced in the first stage. The rate of generated primes against the existing number of primes in the specified interval places the Miller-Rabin approach after the SFA algorithm. The number of required probes in the SFA algorithm is reduced in the first loop to the difference between the upper and lower limits divided by 12, plus the number of iterations in four sieves. The SFA algorithm generates more primes per interval and the computational speed per prime is faster than the Miller-Rabin algorithm, because the non-primes are eliminated sequentially and in parallel method. The SFA processes two equations m=12k+r concurrently.

Thus, the SFA algorithm has advantage over the Miller-Rabin's, which randomly searches for the primes and over the Gordon's algorithm, which randomly generates strong primes. As the table 18 indicates, the Gordon's algorithm has a very small rate of the number of generated prime versus the number of expected primes. Gordon's algorithm similarly to Miller-Rabin's method eliminates the non-primes in the first step by pre-testing the numbers for possible divisibility factors.

Algorithm Parametric Representation of Composite Twins and Generation of Prime and Quasi Prime Numbers invented by Doctor Verkhovsky [108] verifies and generates primes and quasi primes using special mathematical constructs. This algorithm indicated best performance in the interval [1,1000] generating and verifying 3585 variances of provable primes or quasi primes, while an average time per prime, or quasi prime was 0.0022315 [s]. The Parametric Representation of Composite Twins and Generation of Prime and Quasi Prime Numbers algorithm implements very unique method of testing both primes and quasi primes that cannot be compared with the other primality testing algorithms.

The Factorial Function ((a!)^2)*((-1)^b) in Generating Primes algorithm [105] developed by Doctor Verkhovsky provided best results in the lowest interval [1,1000]. In the range [1,1000] factorial algorithm exhausted an average 0.00001 [s] per prime, originating 167 primes, while extended Fermat algorithm produced 167 primes and consumed an average 0.00599 [s] per prime.

Thus, among the selected algorithms the deterministic sieve SFA algorithm that generates provable primes invented by Doctor Verkhovsky indicated best performance considering the computational speed, the simplicity of method, and the number of primes generated in the specified interval.

APPENDIX A

PRIMES TESTING AND GENERATING ALGORITHMS

This appendix provides all algorithms presented and compared in the survey of methods.

PROBABILISTIC PRIMALITY TESTS /* delineated as follows by Menezes [66] */

For each positive integer n a set $W(n) \subset Z_n$ is such that the following properties hold;

(i) given $a \in Z_n$ can be checked in deterministic polynomial time whether $a \in W(n)$

(ii) if n is prime, then $W(n) = 0$ (the empty set)

(iii) if n is composite, then $\#W(n) \geq n/2$

Algorithm /* Fermat's primality test presented by Menezes [66] */

FERMAT(n, t)

INPUT: an odd integer $n \geq 3$ and security parameter $t \geq 1$

OUTPUT: an answer prime or composite to the question: "Is n prime?"

```
1.  for $i$ from 1 to t do the following:
1.1 choose a random integer $a, 2 \leq a \leq n-2$
1.2 compute $r = a^{(n - 1)} \bmod n$
1.3 if $r \neq 1$ then return "composite"
2.  Return "prime".
```

Algorithm / * a probabilistic primality test as presented in Menezes [66] */

SOLOVAY-STRASSEN(n, t)

INPUT: an odd integer $n \geq 3$ and security parameter $t \geq 1$

OUTPUT: an answer "prime" or "composite" to the question: *"Is n a prime?"*

```
1.  for $i$ from 1 to $t$ do the following:
    1.1 choose a random integer $a, 2 \leq a \leq n-2$
    1.2 compute $r = a^{((n - 1)/2)} \bmod n$
1.1 if $r \neq 1$ and $r \neq n - 1$ then return "composite"
1.2 compute the Jacobi symbol $s = (a/n)$
1.3 if $r \neq s \pmod n$ then return "composite"
2.  Return "prime"
```

Summary: If $gcd(a, n) = d$, then d is a divisor of $r = a\hat{}((n - 1)/2) \bmod n$.

Algorithm / * Miller-Rabin probabilistic primality test as presented in Menezes [66] */

```
MILLER-RABIN(n, t)
INPUT: an odd integer $n \geq 3$, and a security parameter $t \geq 1$.
OUTPUT: an answer "prime" or "composite" to the question: "Is n prime?"
1. Write $n - 1 = (2\hat{}s) * r$ such that $r$ is odd.
2. For $i$ from 1 to $t$ do the following:
2.1     Choose a random integer $a$, $2 \leq a \leq n - 2$
2.2     Compute $y = a\hat{}r \pmod n$
2.3     If $y \neq 1$ and $y \neq n - 1$ then do the following:
            $j \leftarrow 1$
            While $j \leq s - 1$ and $y \neq n - 1$ do the following:
                Compute $y \leftarrow y\hat{}2 \pmod n$
                If $y = 1$ then return ("composite")
                $j \leftarrow j + 1$
            If $y \neq n - 1$ then return ("composite")
3. Return ("prime").
```

Algorithm P /* Probabilistic Primality Test presented in Menezes [66], by Knuth [51] */

Given an odd integer n, this algorithm attempts to decide whether or not n is a prime.

Let $n = 1 + 2\hat{}k * q$, where q is odd.

P1. [Generate x] Let x be a random integer in the range $1 < x < n$.

P2. [Exponentiate.] Set $j \leftarrow 0$ and y $\leftarrow x\hat{}q \bmod n$. (As in our previous primality test, $x\hat{}q \bmod n$ should be calculated in $O(\log q)$ steps)

P3. [Done?] (Now $y = x\hat{}((2\hat{}j)*q) \bmod n$.) If $j = 0$ and $y = 1$, or if $y = n - 1$, terminate the algorithm and say " n is probably prime." If $j > 0$ and $y = 1$, go to step P5.

P4. [Increase j.] Increase j by 1. If $j < k$, set $y \leftarrow y\hat{}2 \bmod n$ and return to step P3.

P5. [Not prime.] Terminate the algorithm and say that "n is definitely not prime."

Analysis of Prime Testing Algorithm /* due to Miller under the extended Riemann hypothesis (ERH) [69] and probabilistically by Rabin [88] */

Let ***p*** be an odd number to test for primality; generate ***k*** $\approx$ **(1/2)** ***log*** **2*p*** trial witnesses.

MR Algorithm: /* the primality testing algorithm[63] with the probability of failure equal ***O(p*^(-1/4 +** ε**))** for any $\varepsilon > 0$ */

1. Let ***n*** **-1** = β***2^*v*,** where β is odd
2. Choose an x, $0 \le x < n$
3. Let $t_0 = x$^β ***(mod n)***; for ***i***=1,..,***v***, let t_i = t_i-i^2(***mod n***)
4. Let $t_0 \equiv 1$ apparently prime
5. Let $t_0 \ne 1$ for ***i***, $0 \le i < v$, $t_i = -1$ apparently prime.
6. Otherwise composite

Davenport Method /* [31] implies that. */

if ***gcd (x,n)*** **= 1**, then x^ϕ***(n)*** $\equiv$ **1 *(mod n)*,** where ϕ***(n)*** is minimal.

Lenstra Theorem /* presented by Lenstra[60] */

Let ***n*** = ***C******9^*m*** + ***D******3^*m*** + 1, where **0 < *c* < 3^*m*** and 1 $\le$ ***D*** $\le$ **3^*m*.**

Then ***n*** is a prime if and only if both conditions hold:

1) ***D*^2 - 4*C** is not a square
2) There exists ***l*** with ***l*^(*n* - 1)** $\equiv$ ***1(n)*** and ***(l*^((*n* - 1)/3) -1, *n*) = 1**

Condition 1) distinguishes primes satisfying 2) from products of such integers.

PrimeTest Miller [69] /* presented by Gupta [44] */

```
Input n, if n is a perfect power, say m^s, output 'composite' and HALT
REPEAT FOR EACH x ≤f(n) {
        (1) if x divides n, output 'composite' HALT
        (2) if x^(n-1) ≠ 1 (mod n), output 'composite' and HALT
        (3) if there is an i such that (n - 1)/2^i = m is integral,
        and 1 < gcd (x^m -1, n) < n, output 'composite' and HALT
}       output 'prime' and HALT      }
```

PrimeTest(Rabin) [89] /* presented by Gupta [44] */

```
Input $n$
REPEAT r times {
        (1) randomly pick an $x$ between 1 and $n$
        (2) if $x$^($n$-1) $\neq$ 1 (mod $n$), output 'composite' and HALT
        (3) if there is an I such that ($n$-1)/2^I = $m$ is integral,
            and 1 < gcd ($x$^$m$ -1, $n$) < $n$, output 'composite' and HALT
        }   output 'prime' and HALT
}
```

Prime Test Solovay-Strassen [104] /* presented by Gupta [44] */

```
Input $n$
REPEAT r times {
        (1) randomly pick an $x$ between 1 and $n$
        (2) if gcd ($x$, $n$) > 1, output 'composite' and HALT
        (3) if $x$^(($n$ - 1)/2) (mod $n$) $\neq$ ($x$ / $n$) output 'composite' and HALT
        }   output 'prime' and HALT
}.
```

Morain Theorem /* presents a converse of Fermat's little Theorem [72] */

If there exists an a prime to N such that a^(N - 1) $\equiv$ 1 *mod* N,

but a^(N - 1)/q $\neq$ *mod* N for every prime divisor q of N - 1,

then N is a prime

TRUE PRIMALITY TESTS

PRIMALITY TESTS USING FACTORIZATION of n - *1* /* Menezes [66] */

Let an integer $n \geq 3$, then n is a prime if and only if there exists an integer satisfying

(i) a^(n-1) $\equiv$ 1 (*mod* n); and

(ii) a^(n-1)/q $\neq$ 1 (*mod* n) for each prime divisor q of n - 1.

Jacobi symbol (and Legendre symbol) computation Algorithm / * by Menezes [66] */

JACOBI*(a, n)*

INPUT: an odd integer $n \geq 3$ and an integer a, $0 \leq a \leq n$.

OUTPUT: the Jacobi symbol *(a/n)* (and hence the Legendre symbol when n is a prime).

```
1.   If $a$=0 then return (0).
2.   If $a$=1 then return (1).
3.   Write $a = (2^e)*a_1$, where $a_1$ is odd.
4.   If $e$ is even then set $s \leftarrow 1$. Otherwise set $s \leftarrow 1$ if $n \equiv 1$ or 7 (mod 8),
     or set $s \leftarrow -1$ if $n \equiv 3$ or 5 (mod 8).
5.   If $n \equiv 3$ (mod 4) and $a_1 \equiv 3$ (mod 4) then set $s \leftarrow -s$.
6.   Set $n_1 \leftarrow n$ mod $a_1$.
7.   Return $(s*JACOBI(n_1,a_1))$.
```

Algorithm C /* Factoring by addition and subtraction presented in Menezes [66] */

Given an odd number N, algorithm determines the largest factor of N that is less than or equal to $\sqrt{N}$.

```
C1.  [Initialize.] Set $x \leftarrow \lfloor\sqrt{N}\rfloor + 1$, $y \leftarrow 1$, $r \leftarrow \lfloor\sqrt{N}\rfloor$^2 - $N$ (During algorithm $x$, $y$,
     $r$ )
     Correspond respectively to $2*x + 1$, $2*y + 1$, $x$^2 - $y$^2 - $N$ as we search for a
     solution;
     We will have $|r| < x$ and $y < x$.
C2.  [Done?] If $r$=0 the algorithm terminates; we have $N = ((x - y)/2)((x + y - 2)/2)$.
C3.  [Step x.] Set $r \leftarrow r + x$, and x $\leftarrow x + 2$.
C4.  [Step y.] Set $r \leftarrow r + y$, and y $\leftarrow y + 2$.
C4.  [Test $r$.] Return to step C3 if $r > 0$, otherwise go back to C2.
```

Pocklington's Theorem /* Algorithm presented in Menezes [66] */

The prime F factorization is $F = \prod_{(j=1,t)} (q_j \wedge e_j)$ if there exists an integer a that satisfies;

Let an integer $n \geq 3$ and $n = R{*}F + 1$ (where F divides n -1)

(i) $a \wedge (n-1) \equiv 1\ (mod\ n)$; and

(ii) $gcd\ ((a \wedge ((n-1)/q_j)\ -1, n) = 1$ for each j, $1 \leq j \leq t$,

then every prime divisor p of n is congruent to 1 modulo $F > \sqrt{n} - 1$, then n is a prime.

ALGORITHMS GENERATING PRIME NUMBERS

RANDOMIZED PRIMALITY TESTING AND PRIMES GENERATING ALGORITHMS

Algorithm Finding Square Roots Modulo *a* Prime *p* /* presented in Menezes [66] */

This algorithm is preferable when $p - 1 = (2 \wedge s){*}t$, when s is large.

INPUT: an odd prime p and a square $a \in Q_p$

OUTPUT: the two square roots of a modulo p

```
    Choose random  $b \in Z_p$ until $b \wedge 2 - 4a$ is a quadratic non-residue modulo $p$
    i.e. $(b \wedge 2 - 4a)/p = -1$
    Let $f$ be the polynomial $x \wedge 2 - bx + a$ in $Z_p[x]$
    Compute $r = x \wedge ((p+1)/2)\ mod\ f$ ($r$ is an integer)
    Return $(r, -r)$
```

Computing square roots in a finite field can be extended to find square roots in any finite field F_q of an odd order $q = p \wedge m$, where p is a prime and $m \geq 1$.

Each element $a \in F2 \wedge m$ has exactly one square root, specifically $a \wedge (2 \wedge (m - 1))$.

Algorithm Finding Square Roots modulo a prime p /* a randomized method where the quadratic non-residue is selected, presented in Menezes [66] */

INPUT: an odd prime p and an integer a, $1 \le a \le p - 1$

OUTPUT: two square roots of a modulo p, provided a is a quadratic residue modulo p.

```
1. Compute the Legendre symbol $(a / p)$.
If $(a / p) = - 1$ then return ($a$ does not have a square root modulo $p$) and terminate.
2. Select integers $b$, $1 \le b \le p - 1$, at random until one is found with $(b / p) = - 1$.
($b$ is quadratic non-residue modulo $p$.)
3. By repeated division by 2 write $p - 1 = (2$^$s) * t$, where $t$ is odd.
4. Compute $a$^$(-1)$ $mod\ p$ by the extended Euclidean algorithm.
5. Set $c \leftarrow b$^$t\ mod\ p$ and $r \leftarrow a$^$((t + 1)/2)\ mod\ p$
6. For $i$ from 1 to $s - 1$ do the following:
6.1.Compute $d = (r$^$2 * a$^$( -1))$^$(2$^$(s - i - 1))\ mod\ p.$
6.2.If $d \equiv -1\ (mod\ p)$ then set $r \leftarrow r * c\ mod\ p$.
6.3.Set $c \leftarrow c$^$2\ (mod\ p).$
7. Return $(r,-r)$
```

Algorithm L: /* due to Lehmer [58] presented by Eric Bach */

Let a to be a nonzero square modulo p

then the following randomized algorithm will compute

$b = \sqrt{a}\ (mod\ p)$. This algorithm requires $O(log\ p)$^3 steps and is successful with the 1/2 probability.

```
1) Find $x$ that $\Delta = x$^$2 - 4*a$ is not a square $mod\ p$
2) Let $u = (x + \sqrt{\Delta})/2$
3) In $F_p[\sqrt{\Delta}] = F_p$^$2$ compute $v = u$^$((p + 1)/2$
4) Return $b = v$ as a square root of $a\ mod\ p$
```

(b) the same bounds hold for Adleman-Manders-Miller [5] algorithm that computes the q-th root

modulo prime, when $q \mid (p - 1)$ and $k = 1/2\ log_q\ p$

A Binomial Congruence Algorithm /* a randomized method for solving congruences by Adleman-Manders, and Miller [5] */

x^q $\equiv$ ***a (mod p)*** where ***q*** is a prime divisor of ***p*** **- 1.**

If ***q*** = **2** it reduces to another square algorithm published by Shanks [86].

AMM Algorithm:[5]

1) Let ***p*** **- 1** = ***m*q^*2**, where $q \nmid m$
2) Set a_q = ***a^m***; a_m = ***a^(q^k)***
3) Choose ***q-th*** power nonresidue *x* and set ***g*** = ***x^m***
4) Set ***e*** = **0** and repeat for ***i*** = **0,..,*k*** **- 1**;

   Select e_i, $0 \leq e_i \leq q$ to make ***(g^(e +*** e_{iq}***^i)*** a_q***)^q^(k - i - 1)*** $\equiv$ **1** ***(p)***

   (if $e_0 \neq 0$, quit; the problem is unsolvable)

   Replace ***e*** by ***e*** + e_{iq}***^i***
5) Set b_q = ***g^(-***e_{iq}***)***; b_m = a_m***^(q ^((-i) (mod m)))***
6) Choose *A* and ***B*** to satisfy ***A*m*** + ***B*q^k*** $\equiv$ **1(*p*** **- 1)**
7) Let ***b*** = b_q***^A*bm^B***; return ***b*** as a ***q-th*** root of *x*.

Random Search for Probable Prime Algorithm /* Menezes [66] */

The Miller-Rabin Test

RANDOM-SEARCH(k, t)

INPUT: an integer ***k***, and a security parameter ***t***

OUTPUT: a random ***k-bit*** probable prime

3. Generate an odd ***k-bit*** integer at random
4. Use trivial division to determine whether ***n*** is divisible by any odd prime $\leq$***B***.

   If it is then go to step 1.
5. If ***MILLER-RABIN(n, t)*** outputs "prime" then return ***(n)***

   Otherwise, go to step 1.

DETERMINISTIC PRIMALITY TESTS AND PRIME NUMBER GENERATING ALGORITHMS

Algorithm Finding Square Roots modulo *a* prime *p* where $p \equiv 3$ (*mod* 4) /* deterministic algorithm for a special case presented in Menezes [66] */

INPUT: an odd prime p where $p \equiv 3$ (*mod* 4), and a square $a \in Q_p$

OUTPUT: the two square roots of a modulo p

```
1. Compute $r = a^{((p + 1)/4)} \bmod p$
2. Return $(r, -r)$
```

Algorithm Finding Square Roots mod *a* prime *p* /* Menezes [66]*/ where $p \equiv 5$ (*mod* 8)

INPUT: an odd prime p where $p \equiv 5$ (*mod* 8), and a square $a \in Q_p$

OUTPUT: the two square roots of a modulo p

```
1. Compute $d = a^{((p - 1)/4)} \bmod p$
2. If $d = 1$ then compute $r = a^{((p + 3)/8)} \bmod p$
3. If $d = p - 1$ then compute $r = 2a\,(4a)^{((p - 5)/8)} \bmod p$
4. Return $(r, -r)$
```

Adleman and Huang Theorem 3 /* results [3] that counterpart the Solovay-Strassen results considering the existence of a random polynomial time algorithm for the set of composites */

There exists a $d \in N$ the number of primes between $x^2 - x^{(1.5)}$ and x^2 that is greater than $x^{(1.5)}/((\log^d) x)$

Lemma 4: There exists a $c \in N$ such that for all sufficiently large primes p, $\#T(p) \geq p^{(10.5)}/((\log^c)p)$,

$$\text{where } T(p) = \left\{ <p, q_1, q_2, q_3>:\ \begin{array}{l} <p, q_1, q_2, q_3> \in U(p) \\ \& \\ q_3 \notin \varepsilon(p^8) \end{array} \right\}$$

Parametric Representation of Composite Twins Algortihm /* invented by B. S. Verkhovsky, Ph.D. [107] for Generation of Prime and Quasi-Prime Numbers */

Consider a set of *three* integer parameters t, u, and w.

Output $t, u, w, a, b, c, d, cd, ab, g, V$.

Let $x:=2tu+e$; $y:=2tw$; $z:=x-2t$; $R:=wy-u(x+e)$; $h:=R+x-t$.

Let $cd:=(x+y)(2h+x-y)$ and $ab:=(y+z)(2h+y-z)$.

If e^2=1, then $cd-ab=2$.

Let e=1 and e=-1;

For $-3 \le t \le 3$; $-3 \le u \le 3$; $-3 \le w \le 3$;

compute $x:=2tu+e$; $y:=2tw$; $z:=x-2t$;

$R:=wy-u(x+e)$; $h:=R+x-t$;

$a:=y+z$; $b:=2h+y-z$; $c:=x+y$; $d:=2h+x-y$;

$g:=cd-ab$; $V:=cd\bmod 6$.

If V=1 and g=2, then output $t, u, w, a, b, c, d, cd, ab, g, V$.

STRONG PRIMES

Gordon's Algorithm for Generating Strong Primes /* Menezes [66] */

SUMMARY: a strong prime p is generated

1. Generate two large random primes s and t of roughly equal bit-length.
2. Select an integer io. Find the first prime in the sequence $2*i*t + 1$, for $i = io, io + 1, io + 2, \ldots$

   Denote this prime by $r = 2*i*t + 1$
3. Compose $po = 2*(s\text{^}(r-2) \bmod r)*s - 1$
4. Select an integer jo. Find the first prime in the sequence $po + 2*j*r*s$ for $j = jo, jo + 1, jo + 2, \ldots$

   Denote this prime by $p = po + 2*j*r*s$
5. Return (p).

CONSTRUCTIVE TECHNIQUES FOR PROVABLE PRIMES.

Maurer's Algorithm for generating provable primes. /* Menezes [66] */

```
PROVABLE.PRIME(k)
INPUT: a positive integer k
OUTPUT: a k-bit prime number n
1.  (If k is small, then test random integers by trial division. A table of small primes may
    be precomputed for this purpose.)
    If k ≤ 20 then repeatedly do the following:
    1.1  Select a random k-bit integer n.
    1.2  Use trial division by all primes less than √n to determine whether n is a
         prime.
    1.3  If n is a prime then return (n).
2.  Set c ← 0.1 and m ← 20.
3.  (Trial division bound) Set B ← c*k^2
4.  (Generate r, the size of q relative to n). If k >2*m then repeatedly do the
    following:
    select a random number s in the interval [0,1], set r ← 2^(s-1), until (k - r*k) > m.
    Otherwise (i.e. k ≤ 2*m) set r ← 0.5.
5.  Compute q ← PROVABLE.PRIME(⌊r*k⌋ + 1).
6.  Set I ← ⌊2^(k-1)/(2*q)⌋
7.  success ← 0
8.  While (success = 0) do the following:
    8.1  (select a candidate integer n) Select a random integer R in the interval
         ⌈I + 1, 2*I⌉
         and set n ← 2*R*q + 1.
    8.2  Use trial division to determine whether n is divisible by prime number < B
         If it is not then do the following:
              Select a random integer a in the interval [2,n-2]
              Compute b ← a^(n-1) mod n
```

```
                    If b = 1 then do the following:
                              Compute b ← a^(2*R) mod n and d ← gcd(b-1,n)
                              If d=1 then success ← 1
9.      Return (n).
```

Heath-Brown Theorem [46]: /* concerning the density of primes in small intervals */

For integers ***a, b*** let ***#p[a,b]*** denote the number of primes satisfying $a \le x \le b$

Let ***i(a,b)*** ← 1 if ***#p[a,b]*** ≤ ***(b-a)/(2[log a])*** and **0** otherwise.

Let ∃ alpha such that for sufficiently large *x*

$$\sum_{x \le a \le 2x} \mathbf{i}\,(\mathbf{a_1\, a} + \sqrt{\mathbf{a}}) \le \mathbf{x{}^\wedge(5/6)}\ ((log{}^\wedge a)$$

Lehmer's Algorithm: /* to compute modular square root [58] */

Find *x* so that ***Δ = x^2-4a*** is not a square mod ***p***

Let ***u = (x + √Δ)/2***

In ***Fp[√Δ] = F[p^2]*** compute ***v = u^(p+1)/2***

Return ***b = v*** as a square root of ***a mod p.***

Gupta et al, /* a generalized formula for generating large prime numbers [44] */

```
GenPrime {
REPEAT {
    Pick a large number at random;
    Test whether it is a prime;              }
UNTIL a prime number of desired size is found  }
```

Brillhart, Tonascia, and Weinberger /* [18] a search for odd solutions of N */

The multiplicative hashing scheme presents for the congruence a^(N - 1) ≡ 1(mod N^2)

H(k) = ⌊M(((A/w)K) mod 1⌋ where ***M*** is a prime number such that

r^k ≡ ±a (mod M); for ver small ***k*** and ***a***

A is a constant. integer relatively prime with ***w***

w is a word size of the computer

Goldwasser-Kilian *GK(p)* [41] Algorithm computes a sequence $p = p_1, p_2, \ldots, p_t$ such that p_t *prime* ⇒ … ⇒ p_1 prime.

```
Algorithm /* presented by Goldwasser-Kilian [41] */
Prove (p)
    $p_0 = p$;        $i = 0$;
    lowerbound = 2^((lg p)^(c/(lg lg lg p)))
    certificate = 0
    while $p_i \geq$ lowerboud do
    repeat Randomly choose $A, B \in Z_{p_i}$
        Compute $N_{p_i}(A,B)$
    Until (4*A^3 + 27*B^2, p) = 1 and
        PP((N_pi(A,B))/2) = "probably prime"
        q = ((N_pi(A,B))/2)
        repeat Randomly choose $M \in E_{p_i}(A,B)$
        until $M \neq I$ and $q*M = I$
        $i = i + 1$
        $p_i = q$
        Append $(M, p_i, A, B)$ to certificate
    If $p_i$ is composite then run Prove(p) again
/* $p_i$ is small enough to be
tested deterministcally by [APR]*/
return (certificate)
end
```

Pintz, Steiger, and Szemeredi Algorithm [77]

1) Find *C, D* : *n* = *C*9^*m* + *D*3^*m* + 1, 0 < *C* < 3^*m*
2) If *D*^2 - 4*C* is a square, "*n is composite*"
3) Else test each *l*, 1 < *l* ≤ *c (log n)*^6 for *l*^(*n* - 1) ≡ *1*(*n*) and (*l*^((*n* - 1)/3) -1, *n*) = 1

   If yes, "$n \in Pm$" by certificate *l*.
4) Else "$n \notin Pm$"

Algorithm GK(p) /* presented by Kaltofen and Valente [49] an improved Las Vegas Primality test */

```
Input: $p$, a highly probable prime
Output: either prime or composite along with a certificate of correctness for the assertion.
Begin
    If $p < B$ then
        perform trial divisions to determine whether $p$ is prime
        and return list of trial-divisors
    else
        repeat
            let $a,b$ be randomly chosen elements of $R=Z/pZ$;
            let $q = |E_R(a,b)|/2;$
        until probable prime $(q);$
        repeat
            randomly generate $P \notin E_R(a,b)$
        until $q*P = I\,\infty;$
        return $((P,q,a,b)$ appended to $GK(q))$
end;
```

Lehmer's Algorithm: /* to compute modular square root [58] */

```
Find $x$ so that $\Delta = x$^2-4$a$ is not a square mod $p$
Let $u = (x + \sqrt{\Delta})/2$
In $F_p[\sqrt{\Delta}] = F[p$^2] compute $v = u$^($p$+1)/2
Return $b = v$ as a square root of $a$ mod $p$.
```

Algorithm Atkin(p) /* improved Las Vegas Primality test in Kaltofen - Valente [49] */

Procedure Atkin *(p)*

Input: ***p***, a highly-probable prime.

Output: either prime or composite along with a certificate of correctness for this assertion

```
Begin
    If $p < B$ then
        perform trial divisions to determine whether $p$ is prime
        and return a list of trial-divisors
    else
        repeat
            repeat
                find fundamental discriminant $-D \le -7$
                satisfying $(-D/p) = 1$;
                set $b$ to $\sqrt{-D}$ (mod p);
                adjust $b$ so that its parity is equal to that of $-D$
                reducedform := Reduce [p, b, (b^2 + D)/(4*p)]
            until reducedform = [1,1,(1-d)/4];
            (x y)^T := S(1 0)^T, where $S$ is the transformation matrix
                from [p,b, (b^2+D)/(4*p)] to [1,1,(1 - D)/4];
            {remark: now $\pi = x + y*((1 + \sqrt{-D})/2)$}
            t := 2*p*x + b*y;      m+ := p + 1 + t;      m- := p + 1 - t
        until m+ or m- = k*q, q> (p^(1/4)+1)^2, and probable-prime(q);
        r := root mod p of $H_D(x)$;    l := r(1728 - r)^(-1) (mod p);
        (a,b) := (3*l,2*l) (mod p);    E := $E_F(a,b)$;
        If $(k*q)P \ne I_\infty$ then
            c := randomly chosen quadratic non-residue mod p;
            (a,b) := (a*c^2, b*c^3);      E := $E_F(a,b)$
        EndIf;
        Randomly generate $P \in E$ until $k*P \ne I_\infty$ and $(k*p)*P = I_\infty$;
        Append (P,k,q,a,b) to Atkin(q)
end;
```

SIEVE ALGORITHMS FOR GENERATING PRIME NUMBERS

Pritchard's Algorithm /* presents the sieve of Eratosthenes' by Pritchard [86] */

```
Eratosthenes($n$) {
$a[1] := 0$
for $i := 2$ to $n$ do $a[i] := 1$
        $p := 2$
        while $p^2 \le n$ do {
                $j := 2*p$
                while $(j \le n)$ do {
                        $a[j] := 0$
                        $j := j+p$
                } repeat $p := p + 1$
                until $a[p] = 1$
        } return $(a)$
}
```

Sieve Algortihm /* The 6k+1 sequences invented by B. S. Verkhovsky, Ph.D. [107] */

Sequence $m = 6k + 1$ is generated for $k = 1, 3, 5, \ldots$ {for *odd* k's only};

1. If k **mod** $(6f + 1) = f$, then reject k;
2. If k **mod** $(6f - 1) = 5f - 1$, then reject k;
3. For every k, f is changing from **1** to **ceiling**[($k/6$) ^.5] ;
4. Generate m's on the [*smlst, lrgst*]

   where $m = 6k + 1$

Sieve Algortihm /* The SFA invented by B.S. Verkhovsky, Ph.D. [108] */

```
Let $r:=7, 5$;
$kmin:=ceiling(A/12)$;
$kmax:=floor(B/12)$;
for $k$ from $kmin$ to $kmax$ step 1 do
    for $e$ from 1 to $k/13$ do                              {The 1st Sieve};
        if $k \bmod(12e-r)=11e-r$
            then reject $k$;
            goto the next $k$;
        else goto the next $e$;
    for $f$ from 0 to $k/13$ do                              {The 2nd Sieve};
        if $k \bmod(12f+r)=f$
            then reject $k$;
            goto the next $k$;
        else goto the next $f$;
    for $g$ from 1 to $k/(24-r)$ do                          {The 3rd Sieve};
        if $k \bmod(12g-1)=(12-r)g-1$
            then reject $k$;
            goto the next $k$;
       else goto the next $g$;
   for $h$ from 1 to $k/(12+r)$ do                           {The 4th Sieve};
        if $k \bmod(12h+1)=hr$
            then reject $k$;
            goto the next $k$;
       else goto the next $h$;
output $m:=12k+r$;
```

Giblin Algorithm /* method of primes multiplication by composites elimination [39] */

$r < p \le (r + 2n - 1)$ primes p in some interval

$1 \le k \le n$ where a composite c is defined as $c = r + 2n - 1$

Xuedong Luo Algorithm 1: /* pertaining to the sieve of Eratosthenes' [110] */

```
Assume that $N = 2M$ is even and $j, k, p, q, S$ := 1, 1, 3, 4,  {3, 5,...,2M-1};
a)      if $S[j] = 0$ goto (c)
        Otherwise $k \leftarrow q$
b)      If $k \le M$
then $S[k] \leftarrow 0$,  $k \leftarrow k + p$
and repeat this step
c)      $j \leftarrow j + 1, p \leftarrow p + 2, q \leftarrow q + 2p - 2$
if  $q < M$, return to (a)
$S = |x| x$ is zero or $x$ is prime $\le N$
```

Xuedong Luo Algorithm 2:

In this algorithm_Xuedong Luo uses an array S initially set to [5,7,11,..,$3i + 2, 3(i + 1)+1$,..,N] where i is odd and all non-primes are set to 0 as they are sieved out. Written in a format of guarded commandsas per Dijkstra.[33]

```
Assume that $N$ has the form of $3M+2$, where $M$ is odd;
$i, q, S := 1, \sqrt{N/3}, \{5, 7, 11,...,N\}$;
do $i \le q \rightarrow$   put the position of square of ith element into $C_i$;
                 set zero to all multiples of ith element beginning from $C_i$;
$i := i + 1$;
od
$\{S = \{x \mid x$ is zero or prime $\le N\}\}$
"set... " is implemented:
j := Ci;
do $j \le M \rightarrow S[j] := 0$;
        if $j$ is odd $\rightarrow j := j + 2i + 1$
          $i, j$ is even  $\rightarrow j := j + 4i + 1$
          $i$ is odd  $\rightarrow j := j + 4i + 3$
        fi
od
```

Xuedong Luo Algorithm 3:

Xuedong Luo implied that algorithm 3 is similar to algorithm 2, except algorithm 3 was written in conventional form.

```
Assume $c, k, t, q, M, S$ := 0, 1, 2, $\sqrt{N}/3$, $N/3$, {5, 7, 11,...,N};
For $i$ := 1 to $q$ do begin
$k := 3 - k$; $c := c + 4k * i$; $j := c$;
$ij := 2i * (3 - k) + 1$; $t := t + 4k$;
while $j \le M$ do begin
        $S[j] := 0$; $j := j + ij$; $ij := t - ij$
end
end
```

Algorithm D /* Factoring with sieves presented in Menezes [66] */

Given an odd number *N*, this algorithm determines the largest factor of ***n*** less than or equal to $\sqrt{N}$.

D1. [Initialize.] Set $x \leftarrow [\sqrt{N}]$, and set $k_i \leftarrow (-x) \bmod m_i$ for $1 \le i \le r$.

(Troughout this algorithm the index variables $k_1,k_2,\dots,k_r$ will be set so that $k_i = (-x) \bmod m_i$.)

D2. [Sieve.] If $S[i,k_i] = 1$ for $1 \le i \le r$, go to step D4.

D3. [Step x.] Set $x \leftarrow x + 1$, and set $k_i \leftarrow (k_i - 1) \bmod m_i$ for $1 \le i \le r$.

Return to step D2.

D4. [Test *x*^2 - *N*.] Set $y \leftarrow \lfloor \sqrt{(x^2-N)} \rfloor$, or to $\lceil \sqrt{(x^2-N)} \rceil$. If *y*^2 = *x*^2 - *N*, then $(x - y)$ is the

desired factor, and the algorithm terminates. Otherwise return to step D3.

Pritchard Method /* improved multiplicative sieve [82, modified Mairson's method*/ to attain the $O[N/\log \log N]$ additive sieve.

The Pritchard's enhanced sieve is based on the property that;

$$(p_1*p_2...p_i, p_1*p_2...p_i+k) = 1$$

where $(k, p_j) = 1$; p_i denotes the ***ith*** prime and $1 \le j \le i$.

A Sublinear Additive Sieves Finding Prime Numbers Algorithm 1 /* Pritchard [84] */

RE: Mairson's Algorithm

```
{$N \geq 2$}
$i, p, S := 1, 2, \{2, \ldots, N\}$;
do    $p^\wedge 2 \leq N \rightarrow$
      Establish $C = C_i$;
      $S := S - C$;
      $i, p := i + 1, next\ (S, p)$
od
{$S$ = the set of primes $\leq N$}
```

Algorithm 2 /* P. Pritchard [84] */

```
{$N \geq 5$}
$i, p, length, S, P := 2, 3, 2, \{1\}, \{2\}$;
do    $p^\wedge 2 \leq N$ or $length < N \rightarrow$
      Extend $S$ up to min $(\{p*length, N\})$;
      Remove all multiples of $p$ from $S$;
      $P := P \cup \{p\}$;
      $i, p := i + 1, next\ (S, 1)$
od
{$S \cup P - \{1\}$ = the set of primes $\leq N$}
```

Algorithm /* Pritchard [86] employs Chinese Reminder Theorem in enhancement to prime numbers generators */

```
$k, m, u := 0, 1, 0$;
[invariant: $P$]
do $k \neq r \rightarrow k, m, u := k + 1, m_{k+1}\, m, \ldots$ od
{$P$ and $k = r$}
```

A Linear Sieve Algorithm for Finding Prime Numbers Algorithm 1

```
/* D.Gries and J.Misra  [43]*/
{$n \geq 4$}
$p, q, k, x, S := 2, 2, 1, 4, \{2,\ldots,n\}$;
do $x \leq n$                          → remove {$S, x$};
                                         $k, x := k + 1, p*x$
[] $x > n$ and $p*next\ (S, q) \leq n$   → $q := next\ (S, q)$;
                                         $k, x := 1, p*q$
[] $x > n$ and $p*next\ (S, q) > n$    and next $(S,p)$^2 $\leq n$
                                       → $p := next\ (S, p)$;
                                         $q, k, x := p, 1, p*p$
od
{$S = \{y \mid 2 \leq y \leq n$ and $y$ prime}
```

Algorithm 2 /* D.Gries and J.Misra [43] */

```
$p, S := 2, \{2,\ldots,n\}$;
while $p*p \leq n$ do begin
        $q := p$;
        while $p*q \leq n$ do begin
                $x := p*q$;
                while $x \leq n$ do begin
                remove $(S, x)$; $x := p*x$
                end;
        $q :=$ next $(S, q)$
        end;
$p :=$ next $(S, p)$
end
```

A Multiplicative Sieve Algorithm for Finding Prime Numbers Algorithm

```
/* H.G. Mairson  [62]*/
integer arrays RLINK, LLINK, DELETE
procedure SIEVE (n):
    comment create the doubly linked list
    (RLINK[i] $\leftarrow i + 1$) for $i \leftarrow$ 1 to $N$ - 1 step 1;
    (LLINK[i] $\leftarrow i - 1$) for $i \leftarrow$ 2 to $N$ step 1;
    comment execute the sieve
    PRIME $\leftarrow$ 2; FACTOR $\leftarrow$ 2;
    while PRIME $\leq \sqrt{N}$ do {
        POINTER $\leftarrow$ 0;
        while PRIME*FACTOR $\leq N$ do
            K $\leftarrow$ PRIME * FACTOR;
            POINTER $\leftarrow$ POINTER + 1;
            DELETE[POINTER] $\leftarrow$ K;
            FACTOR $\leftarrow$ RLINK[FACTOR]);
        CROSSOFF(DELETE[i]) for i $\leftarrow$ to POINTER step 1;
        PRIME $\leftarrow$ RLINK[PRIME];
        FACTOR $\leftarrow$ PRIME;
    comment output the primes
    P $\leftarrow$ RLINK[1];
    while P $\neq$ 0 do {
        output {P};
    P $\leftarrow$ RLINK[P]);
    end
    subprocedure CROSSOFF(A);
        RLINK[LLINK[A]] $\leftarrow$ RLINK[A];
        LLINK[RLINK[A]] $\leftarrow$ LLINK[A];
    END
```

APPENDIX B

SELECTED COMPUTER PROGRAMS

This appendix provides selected computer programs that provided results to the computer experiments tables.

PRIMALITY TESTING PROGRAMS

```
// PROGRAM IMPLEMENTS SOLOVAY-STRASSEN ALGORITHM [66] TO
// TEST PRIMALITY. A CONSTANT NUMBER n IS TESTED, BASE a IS
// A RANDOM NUMBER.
// THIS PROGRAM COMPUTES LONG NUMBERS UP TO 2,147,483,647
// WHERE JACOBI SYMBOL IS SIGNED NUMBER.
//
// Program written by Wieslawa E. Amber
// Advisor Professor Boris S. Verkhovsky, Ph.D.
//
//
long n=131303L;           // odd integer 1 <= n <= n-1
//
class Prime{
public:
     Prime(long=0);
     long a();
     long r();
     long agen();
     long ffunct();
     long jfunct();
     void swap(long &, long &);
private:
     long _a;            // random integer 2 <= a <= n-2
     long _r;            // rem r=a^(n-1)/2 mod n if r != 1
                         // then composite
};

// Constructor
// ---------------------------------------------------------
Prime::Prime(long a): _a(a){};

     long Prime::a() { return _a;}
     long Prime::r() { return _r;}
```

```
// Member function agen generates random number a
//-----------------------------------------------------------
long Prime::agen()
{
randomize();
int min=1;     long max=1000L;
long range=max-min+1;    // rand a ranges from min to max

     _a=rand()/100%range+min;      return _a;
}

// Member function ffunct computes a^(n-1)/2 mod n
//-----------------------------------------------------------
long Prime::ffunct()
{
long val, exp;           // local variables
long _r=1;               // initialize _r to 1
     exp=(n-1)/2;
for(long j=1; j<=exp; j++){
     val=_a*_r;          // val is a prod of a and rem r
     _r=val%n;         } // rem r obtained from val mod n
     return _r;
}

// Function computes jacobi symbol
//-----------------------------------------------------------
long Prime::jfunct()
{
signed long jac=1;
     while(_a!=0){
     if(_a==1)
          jac=1;
     if(_a>0 && _a%2==0){
          _a=_a/2;
     if((n%8==1 || n%8==7))
          jac=jac;
     if((n%8==3 || n%8==5))
          jac=-jac;
     }                   // call subfunction swap
     swap(_a,n);         // interchange denominator with
                         // nominator
     if(n%4==3 && _a%4==3)
          jac=-jac;
          _a=_a%n;
     }
     if(n==1)
          return jac;
     else
          return (0);
}

// Member function swap interchanges the denonimator with
dominator
//-----------------------------------------------------------
void Prime::swap(long &a, long &n)
{
long temp = a;
          a = n;
          n = temp;
}

main()
{
signed long s;           // denotes Jacobi symbol
long nprob;              // denotes number of probes
long cnt;                // denotes number of primes
long u,a;
float avg;
long np=n;               // save n as np
time_t first, second;
first = time(NULL);      // Gets system start time

cout<<" R E S U L T S "<<endl;
Prime prime;
for(int x=1; x<=5; x++){ // generate random numbers
     prime.agen();
     a=prime.agen();
     u=prime.ffunct();  } // save reminder as u
     if(u==1 && u==n-1){  // reminder to be equal 1 or n-1
          cnt=1;
          cout<<" n = "<<n<<"\tis a prime "<<endl;
     }
     else {              // Jacobi symbol results
     cout<<prime.jfunct()<<endl;
     s=prime.jfunct();   // save Jacobi symbol as s
     if(s%np==u){        // compare Jacobi symbol to rem
          cnt=1;
          cout<<" n = "<<np<<"\tis a prime "<<endl;}
     else {
          cnt=0;
          cout<<" n = "<<np<<"\tis a composite "<<endl; }
     }    cout<<endl;
cout<<np;
nprob=(np-1)*(np-1)/2;   // calculates number of probes
```

```
second = time(NULL);      // Gets system ending time again
difftime(second,first);   // Compute total CPU time
// Create an output table
//
return 0;
}
```

PRIME NUMBERS GENERATING PROGRAMS

RANDOM SEARCH FOR PROBABLE PRIMES

```
// PROGRAM IMPLEMENTS MILLER-RABIN [66] RANDOM SEARCH FOR A
// PRIME. NUMBER n AND BASE a ARE RANDOMLY GENERATED AND n
// IS PRE-TESTED  FOR PRIMALITY
//
// Program written by Wieslawa E. Amber
// Advisor Professor Boris S. Verkhovsky, Ph.D.
//
class Prime{
public:
     Prime(long=0, long=0, long=0, long=0);
     long a();
     long r();
     long y();
     long s();
     long agen();
     long rfunct();
     long ffunct();
     long sfunct();
private:
     long _a;               // random integer 2 <= a <= n-2
     long _r;               // reminder y != 1
     long _y;
     long _s;
};

long n;
const long L=39000L;
const long U=40000L;
void ngen();
long is_divisible();
char *read0lst="read0lst.dat";

// Constructor
// --------------------------------------------------------
Prime::Prime(long a, long y, long s, long r)
```

```
          : _a(a), _y(y), _s(s), _r(r){};

     long Prime::a()     { return _a;}
     long Prime::r()     { return _r;}
     long Prime::y()     { return _y;}
     long Prime::s()     { return _s;}


// Member function agen generates random number a
//-------------------------------------------------------------
long Prime::agen()
{
randomize();
int min=2;     long max=1000L;
long range=max-min+1;
     _a=rand()/100%range+min; // random a ranges from min
to max
     return _a;               // class object
}

// Member function computes exponents r
//-------------------------------------------------------------
long Prime::rfunct()
{
long diff;
diff=n-1;
_r=diff;
do{_r=_r/2; }
while(_r%2==0);
     return (_r);
}

// Member function ffunct computes y = (a^r) mod n
//-------------------------------------------------------------
long Prime::ffunct()
{
long val;           // local variable
long _y=1;          // initialize _y to 1
for(long j=1; j<=_r; j++){
     val=_a*_y;     // val is a prod of a and reminder r
     _y=val%n;    } // reminder r obtained from val mod n
     return _y;
}

// Member function computes exponents s
//-------------------------------------------------------------
long Prime::sfunct()
```

```
{
long _s=0;     long rdif;
rdif=n-1;
_r=rdif;
do{_r=_r/2;
     _s++;    }
while(_r%2==0);
     return (_s);  }
main()
{
ifstream r0lst(read0lst,ios::out);
long nprob;              // denotes number of probes
long cnt=0;              // denotes number of primes
long y;                  // denotes the reminder
long s;                  // denotes exponent s
float avg;               // average time per prime

time_t first, second;
first = time(NULL);      // Gets system start time

ngen();
cout<<" R E S U L T S "<<endl;
Prime prime;

while(r0lst>>n){
     prime.agen();       // generate random number
     prime.rfunct();
     prime.ffunct();       // compute y=(a^r)mod n
     y=prime.ffunct();
     prime.sfunct();
     s=prime.sfunct();
if(y!=1 && y!=n-1)      { // if mod rem is equal 1 or n-1
     int j=1;
     if(j<=s-1 && y!=n-1){// if reminder is not equal n-1
          y=pow(y,2);
          y=y%n;
          if(y==1)  {       // if reminder is not equal 1
               cout<<"";}          // then n is a composite
               j=j+1;           }
          if(y!=n-1 && y!=1){      // if reminder is not
equal n-1
               cout<<"";}     }    // then n is a composite
else{cout<<setw(6)<<n;             // otherwise n is a
prime number
     cnt++;
     if(cnt%10==0)
          cout<<endl;         }
     }
cout<<endl;
second = time(NULL);     // Gets system ending time
difftime(second,first);  // Compute total CPU time
nprob=(U-L)*s*s;         // calculates number of probes
                         // tested primes diff upper to
lower limit
                         // number of r iterations
// Create an output table
//
return 0;
}

// Function generates a range of n odd random numbers
//--------------------------------------------------------
void ngen()
{
ofstream r0lst(read0lst,ios::out);
int bool;
randomize();
long min=L;    long max=U;
long range=max-min+1;
for(int i=L; i<=U; i++){// random a ranges from min to max
     n=rand()/100%range+min;  bool=is_divisible();
     if(bool==1){
          r0lst<<n<<" ";}     }
}

// Member function pretests if number n has factors
//---------------------------------------------------------
long is_divisible()
{
double N=n, xmax;
     xmax=sqrt(N);
long xM=floor(xmax);
if(n==1) return 0;
if(n%2==0) return 0;
for(long x=3; x<=xM; x++){
     if(n%x==0)
          return 0;
     else
          return 1; }
}
```

EXTENDED FERMAT PROGRAM

```
// PROGRAM IMPLEMENTS FERMAT's ALGORITHM [60] TO TEST
// PRIMALITY AND GENERATES PRIME NUMBERS n IN THE SPECIFIED
// INTERVAL [L,U], WHERE BASE a IS A RANDOMLY GENERATED
// NUMBER.
//
// Program written by Wieslawa E. Amber
// Advisor Professor Boris S. Verkhovsky, Ph.D.
//
class Prime{
public:
     Prime(long=0, long=0);
     long a();
     long r();
     long agen();
     long ffunct();
     long is_prime();
private:
     long _a;            // random integer 2 <= a <= n-2
     long _r;            // reminder r=a^(n-1)/2 mod n
};                       // if r != 1 then composite

long n;
const long L=99000L, U=100000L;
void ngen();
char *read0lst="read0lst.dat";

// Constructor
// -----------------------------------------------------------
Prime::Prime(long a, long r): _a(a), _r(r){};

     long Prime::a() { return _a;}
     long Prime::r() { return _r;}

// Member function agen generates random number a
//------------------------------------------------------------
long Prime::agen()
{
randomize();
int min=1;     long max=U;
long range=max-min+1;    // random a ranges from min to max
     _a=rand()/100%range+min;
     return _a;
}

// Member function ffunct computes a^(n-1)/2 mod n
```

```
//----------------------------------------------------------
long Prime::ffunct()
{
long val;            // local variable
long _r=1;           // initialize _r to 1
for(long j=1; j<=n-1; j++){
     val=_a*_r;      // val is a prod of a and reminder r
     _r=val%n; }     // rem r obtained from val mod n
     return _r;
}

// Member function is_prime tests if the number n is a
prime
//----------------------------------------------------------
long Prime::is_prime()
{
double N=n,xmax;
     xmax=sqrt(N);
long xM=ceil(xmax);
if(n%2==0) return 0;
for(long x=3; x<xM; x+=2)
     if(n%x==0) return 0;
     if(n%x!=0) return 1;
}

main()
{
ifstream r0lst(read0lst,ios::in);
long nprob;          // denotes number of probes
long cnt=0;          // denotes number of primes
long u;
float avg;
long r;

time_t first, second;
first = time(NULL); // Gets system start time
ngen();
cout<<"              L I S T   O F   P R I M E S "<<endl;
Prime prime;
while(r0lst>>n){
     prime.agen();  // generate random numbers
     u=prime.ffunct();    // save reminder as u
     r=prime.is_prime(); // needs to be equal 1 or n-1
     if(r==1 && (u==1 || u==n-1)){
     cout<<setw(7)<<n;
          cnt++;
     if(cnt%10==0) cout<<endl;
     }
     else {cout<<"";}
     }

second = time(NULL);     // Gets system ending time again
difftime(second,first);  // Compute total CPU time
nprob=(U-L)*(U-L);       // calculates number of probes
// Create an output table
//
return 0;
}

// Function generates a range of n odd numbers
//---------------------------------------------------------
void ngen()
{
ofstream r0lst(read0lst,ios::out);

for(long i=L/2; i<=U/2; i++){
     n=i*2+1;
     r0lst<<n<<" ";
     }
}
```

GENERATING STRONG PRIMES

```
// PROGRAM IMPLEMENTS GORDON's ALGORITHM FOR GENERATING
// STRONG PRIMES
//
// Program written by Wieslawa E. Amber
// Advisor Professor Boris S. Verkhovsky, Ph.D.
//
long n,r,s,t,p,p0,nprime,nprob;
long U=10000000L;
long ngen(long);
int is_prime(long);
void tfunct(long);
void p0funct(long);
int p_prime(long);
void pfunct(long);
void test_p(long,long &);

char *read0lst="gordon0.dat";
char *read1lst="gordon1.dat";
char *read2lst="gordon2.dat";
char *read3lst="gordon3.dat";
char *read4lst="gordon4.dat";
main()
{
ifstream r0lst("gordon0.dat",ios::in);
time_t first, second;
first = time(NULL);      // Gets system start time
ngen(n);
tfunct(t);
p0funct(r);
pfunct(p);
test_p(p,nprime);
while(r0lst>>n){
double N=n,sqrtN;
           sqrtN=sqrt(N);
long sn=ceil(sqrtN);
      nprob=2*sn*sn+U;}
second = time(NULL);     // Gets system ending time again
difftime(second,first);  // Compute total CPU time
//
// print results
//
return 0;
}

// function generates random n numbers and pre-tests for
// possible divisors using is_prime sub-function
// -----------------------------------------------------------
long ngen(long n)
{
ofstream r0lst(read0lst,ios::out);
int bool; int cnt=0;
randomize();
int min=1;     long max=U;
long range=max-min+1;
while(cnt<2){       // random n ranges from min to max
     n=rand()/100%range+min;
     bool=is_prime(n);    // call subfunction is_prime
     if(bool==1){
          cnt++;
          r0lst<<n<<" ";}
     else
          cout<<"";  }
}

// sub-function pre-tests the randomly generated numbers n
// for possible divisors
// -----------------------------------------------------------
```

```
int is_prime(long n)
{
double N=n;
double sqrtN=sqrt(N);
long sqrtn=ceil(sqrtN);
if(n%2==0)     return 0;
for(long x=3; x<=sqrtn; x++)
     if(n%x==0)return 0;
     return 1;
}

// function computes r values
// --------------------------------------------------------
void tfunct(long t)
{
ofstream r1lst("gordon1.dat",ios::out);
ifstream r0lst("gordon0.dat",ios::in);
int i;
while(r0lst>>t>>s){
double T=t;
double sqrtT=sqrt(T);
long sqrtt=ceil(sqrtT);
for(i=1; i<=sqrtt; i++){
     r=2*i*t+1;
     r1lst<<r<<" ";        }
     }
}

// function computes p0 values
// --------------------------------------------------------
void p0funct(long r)
{
ofstream r2lst("gordon2.dat",ios::out);
ifstream r1lst("gordon1.dat",ios::in);
ifstream r0lst("gordon0.dat",ios::in);
int k;    long p0,val,u=1;
while(r0lst>>t>>s){
while(r1lst>>r){
for(k=1; k<=r-2; k++){
     val=s*u;
     u=val%r;                }
     p0=2*u*s-1;
     r2lst<<r<<" "<<p0<<" ";   }
     }
}

void pfunct(long p)
```

```
{
ofstream r3lst("gordon3.dat",ios::out);
ifstream r2lst("gordon2.dat",ios::in);
ifstream r0lst("gordon0.dat",ios::in);
int j;    long s,t,r,p0;
while(r0lst>>t>>s){
double S=s;
double sqrtS=sqrt(S);
long sqrts=ceil(sqrtS);
while(r2lst>>r>>p0){
     for(j=1; j<=sqrts; j++){
          p=p0+2*j*r*s;
          r3lst<<p<<" ";      }
          }
     }
}

// function computes probable primes p, pre-tests each
// candidate for possible divisors, if p does not have
// possible divisors, then is printed out as a strong prime
// ---------------------------------------------------------
void test_p(long p,long &nprime)
{
ofstream r4lst(read4lst,ios::out);
ifstream r3lst("gordon3.dat",ios::in);
int bool;
      nprime=0;
while(r3lst>>p){
bool=p_prime(p);    // call pre-test p_prime sub-function
if(bool==1){
     r4lst<<endl<<p;
     nprime++;}
else
     cout<<"";        }
}

// sub-function to pre-test the prime candidates for
// possible divisors
// ---------------------------------------------------------
int p_prime(long p)
{
double N=p,sqrtN;
          sqrtN=sqrt(N);
long sqrtn=ceil(sqrtN);
if(p%2==0)     return 0;
for(long x=3; x<=sqrtn; x++)
     if(p%x==0)return 0;
     return 1;
}
```

SIEVES

```
// PROGRAM IMPLEMENTS THE SIEVE OF ERATOSTHENS ALGORITHM
// Presented by the Menezes [66]
//
// Program written by Wieslawa E. Amber
// Advisor Professor Boris S. Verkhovsky, Ph.D.
//
typedef bool;
bool false=0;
bool true=1;
long N,L;
bool S[20000]={0};
long i,j,p,nprob;
long c1=0,c2=0,nprime=0;

main()
{
time_t first, second;
first = time(NULL);      // Gets system start time
cout<<" Please enter the lower L and the upper U limits of
the interval"
      <<endl<<" L: ";
cin>>L;
cout<<" U: ";
cin>>N;
cout<<endl<<"            L I S T   O F   P R I M E S "
      <<endl;
for(i=2; i<N; i++){           // assert all numbers in
     S[i]=true;               // the range 2 to N

for(j=2; j<N/2; j++)          // numbers that are multiples
          S[2*j]=false;       // of 2, to be removed
          c1++;
          p=3;
}
while(p<N){
for(long j=2; p*j<N; j++)     // numbers that are multiple
     S[p*j]=false;            // of n, to be removed
     do {++p;}                // increment p by 1 go to
     while(!S[p]);            // next p
   c2++;
     }
for(i=L; i<N; i++){
     if(S[i]){                // print entire set after
false had
          cout<<i<<" ";       // been removed
          nprime++;}
     if(i%100==0)
          cout<<endl;
          }
     cout<<endl;

nprob=c1+c2;
second = time(NULL);     // Gets system ending time again
// difftime(second,first);    // Compute total CPU time
//
// print results
//
return 0;
}

// PROGRAM IMPLEMENTS XUEDONG LUO's SIEVE ALGORITHM 1
//
// Program written by: Wieslawa E. Amber
// Advisor Professor Boris S. Verkhovsky, Ph.D.
//
typedef bool;
bool false=0;
bool true=1;
const long U=10000;           // denotes uper limit
const long L=9000;            // denotes lower limit
bool S[U]={0};
long j,k,q,p,M;
long nprob;

main()
{
time_t first, second;
first = time(NULL);           // Gets system start time

long M=U/2;                   // initial condition, N=U
long nprime=0;                // is even
for(j=2; j<U; j++){
     S[j]=true;               // assert all S[j] == true
     if(j==0){
          q=j; }

s2:if(S[j]==false)            // begin step a)
          {goto s1;}          // mark if S[j] == false
     else
```

```
          k=q;
for(k=2; k<=M; k++)
     S[2*k]=false;
     p=3;                     }

while(p<M){
for(k=2; p*k<U; k++)
     S[p*k]=false;            // begin step b)
     do{++p;}                 // increment p and repeat
while p*k<N
     while(!S[p]);  }         // mark if S[p] == false

s1:j=j+1;               // begin step c)
     p=p+2;
     q=q+2*p-2;
     if(q<M)
     {goto s2;}
for(j=L; j<U; j++){
     if(S[j]){          // print j only if S[j] unmarked
          cout<<setw(6)<<j;
          nprime++;}
     if(j%50==0)
          cout<<endl;
     }
nprob=U*(U-L);
second = time(NULL);     // Gets system ending time again
difftime(second,first);  // Compute total CPU time
cout<<endl<<endl;
//
// results
return 0;}
```

```
// PROGRAM IMPLEMENTS SUBLINEAR ADDITIVE SIEVE ALGORITHM 2
// FOR FINDING PRIME NUMBERS. Algorithm by P.Pritchard
//
// Program written by Wieslawa E. Amber
// Advisor Professor Boris S. Verkhovsky, Ph.D.
//
typedef bool;
bool false=0;
bool true=1;
const int U=30000;
const int L=29000;
long N=U;
bool prime[U]={0};
long i,j,p,nprob,nprime=0;
```

```
void funct1();
void funct2();
void funct3();
void funct4();

main()
{
time_t first, second;
first = time(NULL);       // Gets system start time

funct1();
funct2();
funct3();
funct4();

nprob=3*N;
second = time(NULL);      // Gets system ending time again
difftime(second,first);   // Compute total CPU time
//
// results
//
return 0;
}

// function asserts entire set of numbers from 2 to N as
// boolean true
// --------------------------------------------------------
void funct1()
{
for(i=2; i<N; i++){
     prime[i]=true; }
}

// function 2 tests if the numbers in the range from 2 to N
// have factor if yes, these numbers are multiples of 2,
// marked as boolean false, and nonprimes (composites) will
// be removed from the set
// --------------------------------------------------------
void funct2()
{
for(j=2; 2*j<N; j++){
     prime[2*j]=false;
     p=3;                    }
}
```

```
// function 3  tests if the numbers in the range from 2 to
// N have factors if yes, these numbers are multiples of
// 3 <= p <= N, marked as boolean false, and  nonprimes
// (composites) will be removed from the set
// -----------------------------------------------------------
void funct3()
{
while(p<N/2){
for(j=2; p*j<N; j++)
     prime[p*j]=false;
     do {++p;}
     while(!prime[p]);}
}
// function 4 prints all unmarked numbers in the interval
// i=min to N
// -----------------------------------------------------------
void funct4()
{
for(i=L; i<U; i++){
     if(prime[i]){
          cout<<i<<" ";
          nprime++;}
     if(i%80==0)
          cout<<endl;
          }
     cout<<endl;
}

// PROGRAM IMPLEMENTS THE SIEVE OF ERATOSTHENES FOR FINDING
// PRIMES
// as presented by P.Pritchard [86]
// Program written by Wieslawa E. Amber
//
typedef bool;
bool false=0;
bool true=1;
const long N=100;
const long L=90;
bool a[N]={0};
long i,j,p,M,nprime=0,nprob=0;

void sieve();
char *out0file="00000.txt";

main()
{
time_t first, second;
first = time(NULL);       // Get system start time

sieve();

double x = N, result;
     result = sqrt(x);
long rN=ceil(result);

second = time(NULL);      // Get system ending time again
difftime(second,first);   // Compute total CPU time
cout<<endl<<endl;
return 0; }

void sieve()
{
ofstream o0file(out0file, ios::out);
bool a[N]={0};
     a[1]=0;
for(i=2; i<N; i++){
     a[i]=1;
     p=2;
     while(p*p<N){
          j=2*p;
          while(j<=N){
               nprob++;
               a[j]=0;
               j=j+p; }
          do{p=p+1; }
          while(!a[p]);
     }
     if(a[i]==1 && i>=L){
          nprime++;
          o0file<<i<<" ";
          }
     }
}


// PROGRAM IMPLEMENTS THE SFA ALGORITMS USING 2 LINEAR
// EQUASIONS m=12*k+7 and m=12*k+5
// ALGORTITHM BY PROFESSOR BORIS S. VERKHOVSKY, PH.D.
//
// Program written by Wieslawa E. Amber
// Advisor Professor Boris S. Verkhovsky, Ph.D.
//
long k,A,B;
long N71,N51,N72,N52,N73,N53,N74,N54;
long NP71,NP51,NP72,NP52,NP73,NP53,NP74,NP54;
long m,prime,kmax;

void compute(long, long, long &);
void sieve71(long, long &, long &);
void sieve51(long, long &, long &);
void sieve72(long, long &, long &);
void sieve52(long, long &, long &);
void sieve73(long, long &, long &);
void sieve53(long, long &, long &);
void sieve74(long, long &, long &);
void sieve54(long, long &, long &);
void findm7(long,long);
void findm5(long,long);
void mergek();
void mergem();
void print(long,long);

main()
{
cout<<" PROGRAM IMPLEMENTS THE SFA ALGORITM"<<endl;
cout<<" Algorithm by Professor Boris S. Verkhovsky,
Ph.D."<<endl;
cout<<" Program written by Wieslawa E. Amber "<<endl<<endl;
cout<<" Please enter the interval [A,B]. "<<endl<<" A = ";
cin>>A;
cout<<" B = ";
cin>>B;
time_t first, second;
first = time(NULL);           /* Gets system time */

compute(A,B,kmax);
sieve71(k,N71,NP71);
sieve51(k,N51,NP51);
sieve72(k,N72,NP72);
sieve52(k,N52,NP52);
sieve73(k,N73,NP73);
sieve53(k,N53,NP53);
sieve74(k,N74,NP74);
sieve54(k,N54,NP54);
findm7(m,k);
findm5(m,k);
second = time(NULL);          /* Gets system time again */
difftime(second,first);
mergek();
mergem();
print(m,k);
```

```
cout<<endl<<endl;
return 0;
}

void compute(long A, long B, long &kmax)
{
ofstream p00sfa(prn00sfa,ios::out);
double a=A,kM;
kM=(a/12.0);
long kmin=floor(kM);
double b=B,kX;
kX=(b/12.0);
kmax=floor(kX);
for(k=kmin; k<=kmax; k++){
    p00sfa<<k<<" ";
    }
}

void sieve71(long k, long &N71, long &NP71)
{
ofstream p71sfa(prn71sfa,ios::out);
ifstream p00sfa("000sfa.dat",ios::in);
int r=7;  N71=0;    long kmax;
long emax,ek,e,u=0;
while(p00sfa>>k){
kmax=k;
if(k%r!=0 && k%(12+r)!=1){
    emax=k/12;
    emax=floor(emax);
for(e=1; e<emax; e++){
    u=e+1;
    ek=(12*e+7);          }
    if(k%(ek)!=e || k!=(ek*u)-e){
        p71sfa<<k<<" ";
        N71++;        }
        }
    else {}
        NP71= emax; }
}

void sieve51(long k, long &N51, long &NP51)
{
ofstream p51sfa(prn51sfa,ios::out);
ifstream p00sfa("000sfa.dat",ios::in);
int r=5;  N51=0;    long kmax;
long emax,ek,e,u=0;
while(p00sfa>>k){
```

```
kmax=k;
if(k%r!=0 && k%(12+r)!=1){
     emax=k/12;
     emax=floor(emax);
for(e=1; e<emax; e++){
     u=e+1;
     ek=(12*e+r);           }
     if(k%(ek)!=e || k!=(ek*u-e)){
          N51++;
          p51sfa<<k<<" "; }
          }
     else {}
          NP51=emax; }
}

void findm7(long m, long k)
{
ofstream p75sfa(prn75sfa,ios::out);
ifstream p74sfa("074sfa.dat",ios::in);
while(p74sfa>>k){
     m=12*k+7;
     p75sfa<<m<<" ";
     }
}

void findm5(long m, long k)
{
ofstream p55sfa(prn55sfa,ios::out);
ifstream p54sfa("054sfa.dat",ios::in);
while(p54sfa>>k){
     m=12*k+5;
     p55sfa<<m<<" ";
     }
}

void mergek()
{
ofstream p0ksfa(prn0ksfa,ios::out);
ifstream p74sfa("074sfa.dat",ios::in);
ifstream p54sfa("054sfa.dat",ios::in);
long k7,k5;
while((p74sfa>>k7) && (p54sfa>>k5)){
          if(k7<k5){
               p0ksfa<<k7<<" "<<k5<<" ";}
          else{
               p0ksfa<<k5<<" "<<k7<<endl;}
     }
```

```
}

void mergem()
{
ofstream p0msfa(prn0msfa,ios::out);
ifstream p75sfa("075sfa.dat",ios::in);
ifstream p55sfa("055sfa.dat",ios::in);
long m7,m5;
while((p75sfa>>m7) && (p55sfa>>m5)){
          if(m7<m5){
               p0msfa<<m7<<" "<<m5<<" ";}
          else{
               p0msfa<<m5<<" "<<m7<<endl;}
     }
}

void print(long m, long xk)
{
ofstream p07sfa(prn07sfa,ios::out);
ifstream p0msfa("00msfa.dat",ios::in);
ifstream p0ksfa("00ksfa.dat",ios::in);
cout<<endl<<"  List of primes generated by SFA Algorithm"
      <<endl;
int xcnt=0;
int mcnt=0;
s2:  cout<<endl<<setw(8)<<" k  ";
          while(p0ksfa>>xk){{
               xcnt++;
               cout<<setw(8)<<xk;
               }
          if((xcnt)%10==0){
               goto s1;     }
          }
s1:  cout<<endl<<setw(8)<<" m(k) ";
          while(p0msfa>>m){{
               cout<<setw(8)<<m;
               mcnt++;
               }
          if((mcnt)%10==0){
               cout<<endl;
               goto s2;     }
          }
}


// PROGRAM IMPLEMENTS SEQUENCES CONTAINING PRIMES 6k+1
// ALGORITHM
// ALGORITHM BY Professor Boris S. Verkhovsky, Ph.D.
//
// Program written by Wieslawa E. Amber
// Advisor Professor Boris S. Verkhovsky, Ph.D.
//
unsigned long m,k;
unsigned long A;
unsigned long B;
unsigned long prime,pcnt;
float avg;
void compute(unsigned long,unsigned long,unsigned long &);
long is_divisible(unsigned long);
void print(unsigned long,unsigned long &);

char *prn001st = "00prime.dat";
char *prn00six = "00sixk.dat";
char *prn01six = "01sixk.dat";

main()
{
cout<<" Please enter the interval limits [A,B]. "<<endl<<"
A = ";
cin>>A;
cout<<" B = ";
cin>>B;
cout<<endl<<endl<<"           Printing list of primes.
"<<endl;
time_t first, second;
first = time(NULL);           // Gets system time

compute(m,k,pcnt);
print(m,prime);

second = time(NULL);          // Gets system time again
difftime(second,first);       // Total CPU time


// Create an output table
//
 cout<<endl<<endl;
return 0;
}

void compute(unsigned long m,unsigned long k,unsigned long
&pcnt)
{
ofstream p00six(prn00six,ios::out);
int bool;
     pcnt=0;                     // number of probes
unsigned long f,sf,nfs,ns;
unsigned long max,kmax,kmin;
     kmin=floor(A/6);
     kmax=ceil(B/6);
for(k=kmin; k<=kmax; k+=2){
double K=k,aa;
     aa=sqrt(K/6);
     max=ceil(aa);
for(f=1; f<=max; f++){
     pcnt++;
     sf=(6*f+1);
     nfs=(6*f-1);
     ns=5*f-1;
     if((k%(sf)!=f) && (k%(nfs)!=(ns)))
          m=6*k+1;                       }
     if(m>A){
          bool=is_divisible(m);
          if(bool==1){
               p00six<<m<<" ";}
          else
               cout<<"";      }
          }
}

// Member function pretests if the number n has factors
//----------------------------------------------------------
long is_divisible(unsigned long m)
{
unsigned long xM,x;
double M=m, xmax;
     xmax=sqrt(M);
     xM=floor(xmax);
if(m%2==0) return 0;
for(x=3; x<=xM; x++)
     if(m%x==0)
          return 0;
     return 1;
}
```

APPENDIX C

This appendix groups, sorts, and compares the results from the selected computer experiments to indicate the best performing algorithm.

PRIMALITY TESTING ALGORITHMS

TABLE OF RESULTS 1
Fermat's Primality Test [66]

Tested prime	CPU time [s]	Number of probes	Number of primes	time/prime [s]/prime
31337	0.00021	7,928,008	1	0.00021
131303	0.01411	40,309,714	1	0.01411
1311307	3.05030	246,525,528	1	3.05030

TABLE OF RESULTS 2
Miller - Rabin Primality Test [66]

Tested prime	CPU time [s]	Number of probes	Number of primes	time/prime [s]/prime
31337	0.00001	3,008,256	1	0.00001
131303	0.00183	19,563,998	1	0.00183
1311307	1.00000	188,172,411	1	1.00000

TABLE OF RESULTS 3
Solovay - Strassen Primality Test [66]

Tested prime	CPU time [s]	Number of probes	Number of primes	time/prime [s]/prime
31337	0.00611	2,506,880	1	0.00611
131303	1.00230	3,939,060	1	1.00230
1311307	2.00230	118,017,540	1	2.00230

PRIME NUMBERS GENERATING ALGORITHMS

TABLE OF RESULTS 4
pertaining to Sieve Algorithm 1 by Xuedong Luo

interval [L,U]	CPU time	Number of probes	Number of primes	time/prime [s]/prime
[19000,20000]	27.003	20000000	104	0.259644
[9000,10000]	6.120	10000000	112	0.054643
[900,1000]	0.008	100000	14	0.000571

TABLE OF RESULTS 5
Pritchard's Sublinear Additive Sieve Algorithm for Finding Prime Numbers

interval [L,U]	CPU time	Number of probes	Number of primes	time/prime [s]/prime
[19000,20000]	2.00310	60000	104	0.019261
[9000,10000]	0.00820	30000	112	0.000073
[900,1000]	0.00001	3000	14	0.000001

TABLE OF RESULTS 6
pertaining to the Sieve Algorithm presented by Pritchard [86]

interval [L,U]	CPU time [s]	Number of probes	Number of primes	time/prime [s]/prime
[1000,2000]	1.11100	6490446	135	0.008230
[900,1000]	0.00280	1538248	14	0.000200

TABLE OF RESULTS 7
pertaining to D.Gries and J.Misra Sieve Algorithm 2

interval [L,U]	CPU time	Number of probes	Number of primes	time/prime [s]/prime
[900,1000]	2.0031	2142256	14	0.143079
[100,1000]	0.0026	2142256	143	0.000018

TABLE OF RESULTS 8
Gordon's Algorithm for Finding Strong Primes

interval [L,U]	CPU time	Number of probes	Number of primes	time/prime [s]/prime
[1,10000000]	0.00826	10000512	29	0.000285
[1,1000000]	0.00134	1000512	31	0.000043
[1,100000]	0.00061	100578	32	0.000019
[1,10000]	0.00012	10162	12	0.000010
[1,1000]	0.00001	1512	22	0.000001

The number of experiments pertaining to tables 5, 6, and 7 was limited by the 64K automatic data segmentation resulting from array type specified in the algorithm.

TABLE OF RESULTS 9
pertaining to the Sieve of Eratosthenes Algorithm

interval [L,U]	CPU time [s]	Number of probes	Number of primes	time/prime [s]/prime
[24000,25000]	131	27759	94	1.393617
[19000,20000]	88	22259	104	0.846154
[9000,10000]	36	11226	112	0.321429
[900,1000]	9	1165	14	0.642857

The number of experiments pertaining to table 9 was limited by the 64K automatic data segmentation resulting from array type specified in the algorithm.

TABLE OF RESULTS 10
Miller - Rabin Search for Random Primes

interval [L,U]	CPU time [s]	Number of probes	Number of primes	time/prime [s]/prime
[999000,1000000]	20.8123	9000	55	0.378405
[99000,100000]	3.2083	1000	57	0.056286
[99900,100000]	0.0894	400	4	0.022350
[9000,10000]	2.1156	1000	80	0.026445
[9900,10000]	0.0078	100	5	0.001560
[900,1000]	0.0001	100	6	0.000017

TABLE OF RESULTS 11
Extending Fermat's Probabilistic Primality Test to Generate Primes in the specified interval

interval [L,U]	CPU time [s]	Number of probes	Number of primes	time/prime [s]/prime
[999000,1000000]	194.236	1000000	65	2.988246
[99000,100000]	18.763	1000000	87	0.215667
[9000,10000]	2.034	1000000	112	0.018161
[900,1000]	0.001	10000	14	0.000071

TABLE OF RESULTS 12
Extending Fermat's Probabilistic Primality Test to Generate Random Prime

interval [L,U]	CPU time [s]	Number of probes	Number of primes	time/prime [s]/prime
[999000,1000000]	3.762	1000000	83	0.045325
[99000,100000]	3.056	1000000	92	0.033217
[9000,10000]	0.913	1000000	112	0.008152
[900,1000]	0.001	10000	21	0.000048

TABLE OF RESULTS 13
pertaining to SFA Algorithm

Interval [A,B]	CPU time [s]	Number of probes	Number of primes	time/prime [s]/prime
[999000,1000000]	1.1008	142897	84	0.013105
[999900,1000000]	0.0935	91358	10	0.009350
[99000,100000]	0.0096	19437	83	0.000116
[9900,10000]	0.0008	1936	10	0.000080
[900,1000]	0.0001	178	8	0.000013

TABLE OF RESULTS 14
Sequences Containing Primes Algorithm

Interval [A,B]	CPU time [s]	Number of probes	Number of primes	time/prime [s]/prime
[9999900,10000000]	0.01137	4752	3	0.003790
[999900,1000000]	0.00986	1503	1	0.009860
[999000,1000000]	0.00152	14028	14	0.000109
[99000,100000]	0.00087	4452	23	0.000038
[9000,10000]	0.00073	1409	28	0.000026
[900,1000]	0.00001	53	2	0.000005

TABLE OF RESULTS 15
APPLYING EXPONENTIAL FUNCTION ((a!)^2)*((-1)^b) IN PRIME NUMBERS GENERATION [105]

Interval [A,B]	CPU time [s]	Number of probes	Number of primes	time/prime [s]/prime
[1,1000]	0.078	125250	167	0.000467
[1,2000]	0.142	500500	302	0.000470
[1,10000]	8.043	12502500	1228	0.006550
[1,20000]	30.856	50005000	2261	0.013647

TABLE OF RESULTS 16
PARAMETRIC REPRESENTATION OF COMPOSITE TWINS AND GENERATION OF PRIME AND QUASI_PRIME NUMBERS [108]

	CPU time [s]	Number of probes	Number of variances of verified primes and quasi-primes	Time/prime [s]/prime
[1,200]	1	12221	84	0.0119048
[1,1000]	8	112211	3585	0.0022315
[1,10000]	74	1030301	4492	0.0164737

SUMMARY OF RESULTS

TABLE OF RESULTS 17
Comparison Exponential [105] vs. Fermat [66] Algorithms

	Interval [A,B]	CPU time [s]	Number of probes	Number of primes	time/prime [s]/prime
Exponential	[1,1000]	0.001	125250	167	0.00001
Fermat	[1,1000]	1.000	998001	167	0.00599
Exponential	[1,2000]	0.010	500500	302	0.00003
Fermat	[1,2000]	2.000	3996001	302	0.00662
Exponential	[1,10000]	8	12502500	1228	0.00651
Fermat	[1,10000]	18	99980001	1229	0.01465
Exponential	[1,20000]	31	50005000	2261	0.01371
Fermat	[1,20000]	54	399960001	2262	0.02387

TABLE OF RESULTS 18
Comparison Miller-Rabin, SFA, and Sequences Containing Primes Algorithms

	interval [L,U]	CPU time	Number of probes	Number of primes	time/prime [s]/prime
Miller-Rabin	[999000,1000000]	20.8123	9000	55	0.378405
SFA	[999000,1000000]	1.1008	142897	84	0.013105
6KSeq	[999000,1000000]	0.00152	14028	14	0.000109
Miller-Rabin	[999900,1000000]	0.0010	100	1	0.00100
SFA	[999900,1000000]	0.0935	91358	10	0.009350
6KSeq	[999900,1000000]	0.0098	1503	1	0.009860
Miller-Rabin	[99000,100000]	3.2083	1000	57	0.056286
SFA	[99000,100000]	0.0096	19437	83	0.000116
6KSeq	[99000,100000]	0.0008	4452	23	0.000038
Miller-Rabin	[9000,10000]	2.1156	1000	80	0.026445
SFA	[9000,10000]	0.0011	1936	90	0.000012
6KSeq	[9000,10000]	0.0007	1409	28	0.000025
Miller-Rabin	[900,1000]	0.0001	100	6	0.000017
SFA	[900,1000]	0.0001	178	8	0.000013
6KSeq	[900,1000]	0.00001	53	2	0.000005

REFERENCES

1. W. W. Adams, D. Shanks, "Strong primality tests that are not sufficient", *Math. Comp* 39, pp. 255-300, 1982.

2. L. M. Adleman, C. Pomerance, and R. S. Rumely, "On distinguishing prime numbers from composite numbers", *Annals. of Math.*, 117, pp. 173-206, 1983.

3. L. Adleman, M. Huang, "Recognizing primes in random polynomial time", In the *Nineteenth Annual ACM Conference on Theory of Computing*, pp. 462-469, 1987.

4. L. Adleman, M. Huang, "Recognizing primes in random polynomial time", *Technical Report*, University of Southern California, 1988.

5. L. Adleman, M. Huang, "Primality testing and abelian varieties over finite fields"; Lecture Notes of Mathematics, *Spring-Verlag*, vol.1512, New York, 1992.

6. L. Adleman, K. Manders, and G. Miller, "On taking roots in finite fields", In the *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, New York, *IEEE Press*, 1977.

7. F. Arnault, "Le Test de Primalite de Rabin-Miller, Un nombre compose qui le passe" Report 61, Universite de Poitiers Departement de Mathematiques, Nov.1991

8. A. O. L. Atkin, B.J. Birch, *Computers In Number Theory*; J. Brillhart, J. Tonascia, and P. Weinberger, "On The Fermat Quotient; The search for prime solutions, programming solutions", pp. 213-218. *Proceeding of the Science Research Council Atlas Symposium, No. 2* Oxford, August 1969, Academic Press Inc., London, New York, 1971, ISBN: 0 12 065750 3

9. A. O. L. Atkin, "Intelligent Primality Test Offer, Computational perspectives on number theory", D.A. Buell, and J.T. Teitelbaum, eds. *Proceedings of a Conference in Honor of A. O. L. Atkin, International Press*, pp.1-11, 1998

10. S. Atle, "On the normal density of primes in small intervals, and the difference between consecutive primes", *Arch. Math. Naturvid.* B 47, No.6, pp.1-19 1943.

11. E. Bach, "Realistic analysis of some randomized algorithms", *The Nineteenth Annual ACM Conference on Theory of Computing*, pp. 453-461, 1987.

12. M. B. Barban, J. V. Linnik, and N. G. Tshudakov, "On prime numbers in arithmetic progression with a prime-power", *Acta Arith.* vol. 9, pp. 375-390, 1964.

13. C. Bayes, R. Hudson, "The segmented sieve of Eratosthenes and primes in arithmetic progression", *BIT*, 17, pp. 121-127, 1977.

14. P. Beauchemin, G. Brassard, C. Crepeau, C. Goutier, and C. Pomerance, "The generation of random numbers that are probably prime"; *Cryptol Journal* vol. 1, no. 1, pp. 53-64, 1988.

15. L. Blum, M. Blum, M. Shub, "A simple secure pseudorandom number generator" *Proceedings of CRYPTO 82, Plenum*, 1982.

16. W. Bosma, "Primality testing using elliptic curves", Technical Report 85-12, Institute of Math, Universiteit van Amsterdam, The Netherlands, 1985.

17. W. Bosma, M. P. Van Der Hulst, "Faster primality testing", In *Proceedings of EUROCRYPT '89*, Lecture Notes in Computer Science, vol. 434, *Springer-Verlag*, New York, pp. 652-656, 1990.

18. J. Brillhart, D. H. Lehmer, J. L. Selfridge, "New primality criteria and factorizations" *Mathematics of Computation*, vol. 29, no. 1930, pp. 620-647, 1975.

19. D. M. Bressoud, *Factorizations And Primality Testing*, Springer-Verlag, New York, ISBN 3-540-97040-1, pp. 19, 1989.

20. D. M. Bressoud, S. Wagon, *A Course in Computational Number Theory*, Chapters 4, 8, 9, and Appendix B, Emeryville, CA, Key College Publishing in cooperation with Springer-Verlag, ISBN 1-930190-10-7, 2000.

21. R. D. Carmichael, "On composite numbers p which satisfy the Fermat congruence" *Am. Math. Mon.*, vol. 19, no.2, pp. 22-27, 1912

22. C. C. Chang, "The study of an ordered minimal perfect hashing scheme"; *Commun. ACM* vol. 27, no. 4, pp. 384-387, April 1984.

23. B. A. Chartres, "Algorithms", pp. 310-311; "Prime number generators 1 and 2", *Comm., ACM*, vol. 10, no. 9, pp. 569-570, 1967.

24. D. V. Chudnovsky, G. V. Chudnovsky, "Sequences of numbers generated by addition in formal groups and new primality and factorization tests", *Advances in Applied Mathematics*, vol. 7, pp. 385-434, 1986.

25. H. Cohen, H. W. Lenstra, Jr., "Implementation of a new primality test", *Mathematics of Computation*, vol. 48 no. 177, pp. 103-121, 1987.

26. H. Cohen, H.W. Lenstra, Jr., "Primality testing and Jacobi sums", *Mathematics of Computation*, vol. 42, no. 165, pp. 297-330, 1984.

27. T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction To Algorithms*; The MIT Press, Cambridge, Mass., 1990

28. J. H. Davenport, G. C. Smith, "Rabin's primality testing algorithm", *A Group Theory Theory View*, University of Bath, Technical Report 87-04, 1987.

29. I. Damgard, P. Landrock, "Improved bounds for the Rabin primality test", In *Proceedings of 3rd IMA conference on Coding and Cryptography, ed.* M. Ganley, OUP, 1991.

30. J. H. Davenport, G. C. Smith, "Rabin's primality testing algorithm", *A Group Theory View*, University of Bath, Technical Report 87-04, 1987.

31. J. H. Davenport, "Primality testing revisited", School of Mathematical Sciences, University of Bath BA 27 AY, England, *Proceedings of the Annual Conference on Computer Science In Communications of the ACM*, pp. 123-12, July 1992.

32. J. A. Davis, D. B. Holdridge, "An update on factorization at Sandia National Laboratories", Albuquerque, New Mexico 87185, In *Springer Verlag Heidelberg*, Online Publication, November 24, pp. 114, 2000.

33. E. W. Dijkstra, "Guarded commands, nondeterminancy and formal derivation of programs", *Commun. ACM* vol. 18, No. 8, pp. 453-457, Aug. 1975.

34. R. E. Dressler, T. S. Parker, "Primes with a prime subscript"; *ACM Journal* vol. 22, no. 3, July, pp. 380-381, 1975

35. R. E. Dressler, A. Makowski, T. S. Parker, "Sums of distinct primes form congruence classes modulo 12", *Math. Comp.* vol. 28 , pp. 651-652, 1974.

36. P. Erdos, J. Spencer, *Probabilistic Methods In Combinatorics*, Academic Press, New York, 1974.

37. P. Furer, "Deterministic and Las Vegas primality testing algorithms", *Proc. of ICALP*, 1985.

38. A. Galligo, S. Watt, "A numerical absolute primality test for bivariate polynomial", *Proceedings of the International Symposium on Symbolic and Algebraic Computation, ACM, Inc.* ISSAC, Maui Hawaii, USA, pp. 217-224, 1997.

39. P. Giblin, *Primes And Programming*; *An Introduction To Number Theory With Computing*, p. 51, Cambridge University Press, 40 West 20 Street, New York, NY 10011, 1993.

40. S. Goldwasser, J. Kilian, "Almost all primes can be quickly certified"; *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing,* Berkeley, CA, May, pp. 316-329, 1986.

41. S. Goldwasser, J. Kilian, "Primality testing using elliptic curves; Primality proving algorithm", *Journal of the ACM,* vol. 46, pp. 458-461, July 1999.

42. A. Granville, "Primality testing and Carmichael numbers", *Notices Amer. Math. .Soc.* vol. 39, pp. 696-700, 1992.

43. D. Gries,, J. A. Misra, "A linear sieve algorithm for finding prime numbers", *Commun. ACM* vol. 21, no. 12, pp. 999-1003, Dec. 1978.

44. R. Gupta, S. A. Smolka, S. Bhaskar, "On randomization in sequential and distributed algorithms"; Sect. 2.2, *ACM Computing Surveys*, vol. 26, no. 1, pp. 17-22, March 1994.

45. A. Haas, "The multiple prime random number generator", *ACM Transactions on Mathematical Software,* vol. 13, no. 4, pp. 368-381, 1987.

46. D. R. Heath-Brown, "The differences between consecutive primes", London, *Mathematics of Computation Journal* vol. 2 no. 18, pp.7-13, 1978.

47. J. Heintz, M. Sieveking, "Absolute primality is decidable in random polynomial time in the number of variables", *Springer-Verlag*, no. 11, in LNCS, pp. 16-28, 1981.

48. H. Iwaniec, M. Jutila, "Primes in Short Intervals", *Ark. Mat.* 17, no. 1, pp. 167-176, 1979.

49. E. Kaltofen, T. Valente, and N. Yui, "An improved Las Vegas primality test; a modification of Goldwasser-Kilian-Atkin primality test", *Proceedings of the ACM-SIGSAM, International Symposium on Symbolic and Algebraic Computation,* ISSAC '89, Portland OR, Gilt Gonnet, ed. ACM, New York, pp. 26-33, 1989.

50. J. Kilian, "Uses of randomness in algorithms and protocols", 1989 Kilian's Ph.D. dissertation selected as an ACM Distinguished Dissertation, The MIT Press, Cambridge, Mass. 1990.

51. D. E. Knuth, *The Art of Computer Programming*, vol. 2, Addison-Wesley, an Imprint of Addison Wesley Longman, Inc. Third Edition, ISBN 0-201-89684-2, 1998

52. S. Konyagin, C. Pomerance, "On primes recognizable in deterministic polynomial time", *In Mathematics of P. Erdos, R. Graham, and J. Nesetril, eds.*, Springer- Verlag, New York, pp. 177-198, 1997.

53. L. Kronsjo, *Computational Complexity of Sequential and Parallel Algorithms*; John Wiley and Sons, New York, Ch. 5, Sect. 5.3, 1985.

54. Y. S. Kuo, W. K. Chou, "Generating essential primes for a boolean function with multiple-valued inputs", *Proceedings of the 23rd ACM/IEEE Conference on Design Automation*, pp. 193-199, 1986.

55. G. C. Kurtz, D. Shanks, H. C. Williams, "Fast primality tests for numbers less than 50*10^9", *Math. Comp.* vol. 46, pp. 691-701, 1986.

56. D. Lehmann, "On primality tests, based on the Extended Riemann Hypothesis (ERH)" *SIAM Computing Journal,* vol.11, no.2, May 1982.

57. D. H. Lehmer, "Tests for primality by the converse of Fermat's theorem", *Bulletin American Math. Society*, vol. 33, no. 1, pp. 327-340, 1927.

58. D. H. Lehmer, "Computer technology applied to the theory of numbers", *MAA Studies in Mathematics*, vol. 6, 1969.

59. H. W. Lenstra, Jr., "Primality testing algorithms", Seminare Bourbaki 1980/81, *Springer-Verlag*, Berlin-Heidelberg, pp.243-257, 1981.

60. H. W. Lenstra, Jr., "Primality testing", *Math Centrum Tracts 154*, Computational Methods in Number Theory, Mathematisch Centrum, Amsterdam, pp. 55-77, 1982.

61. A. K. Lenstra, H. W. Lenstra Jr., "Algorithms and complexity, algorithms in number theory", J. van Leeuwen, ed., *Handbook of Theoretical Computer Science*, vol. A, pp. 673-715, 1990.

62. H. G. Mairson, "Some new upper bounds on the generation of prime numbers", *Commun. ACM,* vol. 20, no. 9, pp. 664-669, Sept. 1977.

63. C. P. Mawata, "A sparse distributed representation using prime numbers", *Proceedings of the Symposium on Symbolic and Algebraic Computation*, pp. 110-114, 1986.

64. P. C. McGeer, R. K. Brayton, "Efficient prime factorization of logic expressions", *26th ACM/IEEE Design Automation Conference*, pp. 221-224, 1989.

65. K. Mehlhorn, "On the program size of perfect and universal hash functions", In *Proceedings of the 23rd Annual IEEE Symposium on the Foundations of Computer Science, IEEE*, New York, pp. 170-175, 1982.

66. A. J. Menezes, *Handbook Of Applied Cryptography*; Boca Raton, CRC Press, 1997.

67. M. Mignotte, "Tests de primalite", *Theoret. Comput. Sci*, vol. 12, pp. 109-117, 1980.

68. P. Mihalescu, "Cyclotomy primality proving-recent developments", In *Proceedings of the 3rd International Algorithmic Number Theory Symposium (ANTS)*, Lecture Notes in Computer Science, vol. 877, Springer-Verlag, New York, pp. 95-110, 1994.

69. G. Miller, "Riemann's hypothesis and test for primality", In the *Proceedings of the7th Annual ACM Symposium on the Theory of Computing*, pp. 234-239, 1975. *Computer and System Science Journal* vol. 13, pp. 300-317, 1976.

70. L. Monier, "Evaluation and comparison of two efficient probabilistic primality testing algorithms", *Theoret. Comput. Sci.*, vol. 12, no. 1, pp. 97-108, 1980.

71. F. Morain, "Distributed primality proving and primality of (2^3539+1)/3", In *Advances in Cryptology* - EUROCRYPT '90, 1991, I. B. Damgard, ed., vol. 473, Springer-Verlag, pp. 110-123, *Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques*, Aarhus, Denmark, May 21-24, 1990.

72. F. Morain, "Elliptic curve primality proving and some titanic primes", In *Journees Arithmetiques*, 1989, vol. 198-200 of Astrisque, SMF, pp. 245-251, 1992.

73. F. Morain, "Easy numbers for the elliptic curve primality proving algorithm", and "Prime Values of Partition Numbers And Some New Large Primes", Apr. 1992. In the *ACM Proceedings*, vol. 490-2, pp. 263-268, 1992,

74. B. Z. Moroz, "The distribution of power residues and non-residues", *Vestnik*, Leningrad University 16, No. 19, pp. 164-169, 1961.

75. A. J. Pettofrezzo, D. R. Byrkit, *Elements Of Number Theory*, Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1970

76. J. Pintz, W. Steiger, and E. Szemeredi, "Two infinite sets of primes with fast primality tests and quick generation of large primes", *Proceedings of The Twentieth Annual ACM Symposium on Theory of Computing*, pp. 504-509, 1988, *Mathematics of Computation Journal* vol. 53, no.187, pp. 399-406, 1989,

77. J. M. Pollard, "Theorems on factorization and primality testing", In *Proceedings Cambridge Philos. Soc.* vol. 76, pp. 521-528, 1974.

78. C. Pomerance, "Very Short Primality Proofs", *Mathematics of Computation* vol. 48, no. 177, pp. 315-322, 1987.

79. C. Pomerance, J. L. Selfridge, and S. S. Wagstaff, "The pseudoprimes up to 25*10^9 In reference to Rabin's algorithm that begins with a choice of a random seed x, not congruent to 0 modulo N", *Math. Comp.* vol. 35, pp. 1003-1026, 1980.

80. V. R. Pratt, "Every prime has a succinct certificate", *SIAM Computing Journal*, vol. 4, no. 3, pp. 214-220, 1975.

81. P. Pritchard, "Variations on a scheme of Eratosthenes", Rept. No.8, Dept. Comp. Sci., Univ. of Queensland, Australia, 1979.

82. P. Pritchard, "On the prime example of programming, in language design and programming methodology", *Lecture Notes in Computer Science 79, Springer-Verlag*, Berlin, Heildeberg and New York, pp. 85-94, 1980.

83. P. Pritchard, "A sublinear additive sieve for finding prime numbers", *Commun. ACM*, vol. 24, no. 1, pp. 18-23, Jan. 1981.

84. P. Pritchard, "Explaining the wheel sieve. Fast compact prime number sieves", *Acta Informatica*, vol. 17, pp. 477-485, 1982.

85. P. Pritchard, "Some negative results concerning prime number generators, Programming Techniques And Data Structures", *Communications of ACM*, vol. 27, no. 1, pp. 53-57, 1984.

86. P. Pritchard, "Linear prime-number sieves: A family tree. Comparison of various sieves", *Science Computer Programming Journal*, vol. 9, pp. 17-35, 1987.

87. M. O. Rabin, *Probabilistic algorithms. In algorithms and complexity. New directions directions and recent results*, Academic Press, New York, pp. 21-39, 1976.

88. M. O. Rabin, "Probabilistic algorithms for testing primality", *Journal of Number Theory*, vol. 12, pp. 128-138, 1980.

89. R. A. Rankin, "The difference between consecutive prime numbers", *V. Proc.* Edinburgh *Math. Soc.*, vol. 2, no. 13, pp. 331-332, 1962.

90. H. E. Reichert, "Uber zerfallungen in ungleiche primzahlen", *Math.* Z.62, pp. 342-343 1949.

91. P. Ribenboim, "The book of prime number records", 2nd ed. Springer, 1989.

92. H. Riesel, "Prime numbers and computer methods for factorization", Boston, Mass., Birkhauser, 1985 (revised and corrected printing 1987)

93. K. H. Rosen, *Elementary Number Theory And Its Applications*, Addison-Wesley Publishing Company, Reading Massachusetts, ISBN 0-201-06561-4,1984.

94. R. Schoff, "Elliptic curves over finite fields and the computation of square roots mod P", *Math. Comp.* 44, pp. 483-494, 1985.

95. M. R. Schroeder, *Number Theory In Science And Communication With Applications In Cryptography, Physics, Biology, Digital Information And Computing;* Springer-Verlag, New York, 1984

96. A. Selberg, "On the normal density of primes in small intervals, and the difference between consecutive primes", *Archiv for Mathematik of Naturvidensakb*, B, XLVII, no. 6, pp. 483-494.

97. D. Shanks, "On Maximal gaps between successive primes", *Mathematics of Computation Journal*, vol. 18, pp. 646-651, 1964.

98. D. Shanks, "Class number, a theory of factorization and genera", Proc. Symp. In Pure Mathematics 20, 1969, *Amer. Math. Soc.*, Providence, RI, pp. 415-440 1971.

99. D. Shanks, "Five number-theoretic algorithms", *Proceedings of the Second* Manitoba *Conference on Numerical Mathematics*, pp. 51-70, 1972.

100. P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer", *SIAM J. Comput.*, vol. 26, no. 5, pp. 1484-1509, 1997, *SIAM* Review, vol. 41, no. 7, pp. 303-332, 1999.

101. R. C. Singleton, "Algorithms", pp. 356-357, "Prime number generation using the tree sort principle", *Comm. ACM* vol.12, no. 10, pp. 563-564, 1969.

102. M. Sipser, "Expanders, randomness, or the time versus space", *Proceedings of the Second* Manitoba *Conference on Numerical Mathematics*, pp. 51-70,1972.

103. R. Solovay, and V. Strassen, "A Fast Monte-Carlo Test for Primality"; *SIAM, Computing Journal* 6, pp. 84-85, 1977.

104. G. Szekeres, "Higher order pseudoprimes in primality testing, combinatorics", Paul Erdos, is eighty, *Bolyai Soc. Math. Stud.*, vol. 2, *Janos Boylai Math Soc.*, Budapest, pp. 451-458, 1996.

105. B. S. Verkhovsky, Ph.D., "Applying exponential function ((a!)^2)*((-1)^b) in Generating Primes", Manuscript and personal communication, Dec, 2000.

106. B. S. Verkhovsky, Ph.D., "Sequences containing primes algorithm", Manuscript and personal communication, Dec, 2000.

107. B. S. Verkhovsky, Ph.D., "Sieve algorithm for primes, the San-Francisco algorithm" Manuscript and personal communication, Dec. 2000.

108. B. S. Verkhovsky, Ph.D., " Parametric representation of composite twins and generation of prime and quasi-prime numbers ", Manuscript and personal communication, Dec, 2000.

109. A. Weil, "The field of definition of variety", *Am. Jour. Math.*78, pp. 509-524, 1956.

110. H. C. Williams, "Primality testing on a computer", *Ars Combinatorica*, vol. 5, pp. 127-185, 1980.

111. M. C. Wunderlich, "A performance analysis of a simple prime-testing algorithm" *Mathematics of Computation Journal*, vol. 40, no. 162 pp. 709-714, 1983.

112. M. C. Wunderlich, "Sieving procedures on a digital computer", Journal of the ACM, vol. 14 no. 1, pp. 10-19, January 1967.

113. L. Xuedong, "A practical sieve algorithm for finding prime numbers", In *Communications of the ACM*, vol. 32, no. 3, pp. 344-346, March 1989.