Investigation and Implementation of Some Probabilistic Algorithms

by

Ahmad Said Ghazal

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

June, 1988

Investigation and implementation of some probabilistic algorithms

Ghazal, Ahmad Said, M.S.

King Fahd University of Petroleum and Minerals (Saudi Arabia), 1988

INVESTIGATION AND IMPLEMENTATION OF SOME PROBABILISTIC ALGORITHMS

BY

AHMAD SAID GHAZAL

A Thesis Presented to the
FACULTY OF THE COLLEGE OF GRADUATE STUDIES
KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE
In
COMPUTER SCIENCE

JUNE 1988

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN 31261, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis, written by AHMAD SAID S. GHAZAL under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate studies, in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE in COMPUTER SCIENCE.

THESIS COMMITTEE

[illegible]
Thesis Advisor

[illegible]
Member

[illegible]
Member

[illegible]
Department Chairman

[illegible]
Dean, College of Graduate Studies

Aug. 6, 1988
Date

كلية الدراسات العليا
COLLEGE OF GRADUATE STUDIES
KING FAHD UNIVERSITY OF PETROLEUM

This thesis is dedicated to my parents

and to my all brothers and sisters

ACKNOWLEDGMENT

Acknowledgment is due to the King Fahd University of Petroleum and Minerals for providing the opportunity to carry out this research work.

I gratefully acknowledge the tremendous help, guidance and encouragement by Dr. Mohammed Ibrahim Al-Suwaiyel who served as my committee chairman. I express my thanks and appreciation to the other members of my thesis Committee, Dr. Muhammed Khayat and Dr. Peter Walker.

I would like to thank Dr. Yilimiz Akildiz for his help in in writing the thesis. I would also like to thank Mr. Muhammed Ammar Al-samman for his help throughout all the time this work was in progress. I would like to thank those who helped me to edit this thesis those are : Khalid Hameed and Ali Damati. Finally, I thank everybody who supported me by any means.

TABLE OF CONTENTS

LIST OF TABLES

THESIS ABSTRACT

NAME OF STUDENT: AHMAD SAID GHAZAL

TITLE OF STUDY : INVESTIGATION AND IMPLEMENTATION OF SOME PROBABILISTIC ALGORITHMS

MAJOR FIELD : COMPUTER SCIENCE AND ENGINEERING

DATE OF DEGREE : JUNE 1988

Probabilistic algorithms were shown to be of importance after some fast polynomial time probabilistic solutions to some "hard" problems were given.

In this thesis theoretical foundations of probabilistic algorithms are studied. Probabilistic solutions to some "hard" problems are studied and investigated. Also probabilistic algorithms for some "easy problems" are studied and presented.

The main concentration is on the probabilistic solutions to primality testing. The probability of a composite integer passing the Fermat test is derived. This includes an upper bound on the probability of a given integer being a Carmichael number. Three applications of this work are given as follows :

1.Using the pseduoprimlity test alone to test for primality.

2.Using the pseduoprimlity test with the Rabin test to test for Carmichael numbers.

3.Using the pseduoprimality test to generate keys for the Pohlig-Hellman encryption scheme.

MASTER OF SCIENCE DEGREE

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

Dhahran, Saudi Arabia

June 1988

خلاصة الرسالة

أسم الطالب: أحمد سعيد غزال

عنوان الدراسة: بحث و تنفيذ بعض الخوارزميات الإحتمالية

التخصص: علوم و هندسة الحاسب الألي

تاريخ الدرجة: حزيران ١٩٨٨ م

أدى ظهور حلول إحتمالية سريعة لبعض المسائل الصعبة إلى إبراز أهمية الخوارزميات الإحتمالية .

في هذه الرسالة تدرس الأسس النظرية للخوارزميات الإحتمالية و كذلك الخوارزميات الإحتمالية لبعض المسائل الصعبة كمسئلة الكشف عن الأرقام الأولية و بعض المسائل في الحقول النهائية ، كما نعطي خوارزميات إحتمالية لبعض المسائل السهلة كمسئلة أقرب زوج من النقاط و مسئلة مطابقة النمط.

وسيكون التركيز الرئيسي على الخوارزميات الإحتمالية لمسئلة الكشف عن الأرقام الأولية و منه يستنتج إحتمالية عدم الكشف عن الأرقام المركبة تحت إختبار فيرمات و يشتمل ذلك أيضا على إستنتاج حد أعلى لإحتمال كون رقم ما من أعداد "كارميكل" .

نتج عن هذا البحث نتائج رئيسية و هي :

١. إستعمال إختبار فيرمات فقط للكشف عن الأعداد الأولية .

٢. إستعمال إختبار فيرمات مع إختبار رابين للكشف عن أعداد "كارميكل" .

٣. إستعمال إختبار فيرمات لتوليد مفاتيح للتشفير بإستعمال طريقة بوهليق هلمان.

درجة الماجستير في العلوم

جامعة الملك فهد للبترول والمعادن

الظهران – المملكة العربية السعودية

حزيران ١٩٨٨ م

CHAPTER I

INTRODUCTION

Probabilistic algorithms have come up fairly recently in computer science and have been of practical value in important areas like cryptography.

Probabilistic algorithms offer a good promise of coping with the difficulty of solving some hard problems , especially those which are not proven to be NP-complete. Examples are primality testing and factoring polynomials over a finite field. There are also probabilistic solutions to polynomial problems that are faster than the deterministic ones , e.g the the probabilistic algorithms for the nearest pair problem and the pattern matching problem

However the speed gained by the use of probabilistic algorithms is at the expense of some "uncertainty" in the solution by either allowing wrong answers to occur , but with "low probability" , or by allowing the algorithm not to halt , i.e. no answer is given, but again with "low probability". Probabilistic algorithms should not be confused with probabilistic analysis of the complexity of some deterministic algorithms.

This thesis is concerned with :

1) The study of the theoretical foundations of probabilistic algorithms and

2) A study of various practical examples of probabilistic algorithms.

Chapter two covers definitions and the structure of probabilistic algorithms which are based on probabilistic turing machines (PTMs). It also covers the basic types of probabilistic algorithms, i.e the Monte Carlo and the Las Vegas type and gives subtypes of these two types. Finally chapter two gives the hierarchy of probabilistic algorithms.

Chapter three covers probabilistic algorithms for primality testing which belongs to RP (random polynomial time which is a mixture of Monte Carlo and Las Vegas types). Sections 3.3 and 3.4 explain the two methods for primality testing by Rabin [RAB80] and Solovay & Strassen [SOL77].

The best known deterministic algorithms for primality testing are exponential in terms of the the length of the tested number , see [ADL83] and [COH87] , while both probabilistic algorithms by Solovay and Rabin are polynomial in terms of the length of the tested number. Section 3.5 investigates the idea of using the pseudoprimality test alone for primality testing and contains the derivation of an upperbound on the probability of a composite integer passing this test as a prime. Section 3.5.3 presents an application using the pseduoprimality test with the Rabin test to test if given number is carmichael or not. Section 3.5.4 presents an application of using the pseduoprimality test in the generation of keys for an encryption scheme. The three approaches , i.e the Rabin , Solovay & Strassen and the Pseduoprimality test , were implemented using C and the source listing is given in Appendix A.

Chapter four completes of the overview of the types of probabilistic algorithms by giving more details on some of them. Two are of the Las Vegas type and one of the Monte Carlo type. Section 4.2 presents a probabilistic algorithm for the nearest pair problem . This problem is of the Las Vegas type and the best known deterministic solution for it is O(n*log n) by Shamos , Benteley and Yuval , [RAB76].

Rabin gave a probabilistic algorithm for this problem of complexity O(n) with "high probablity" , [RAB76]. However the answer is always correct. This algorithm was implemented in C and the listing is given in Appendix B.

Section 4.3 presents a probabilistic algorithm for pattern matching. The deterministic algorithm for solving this problem is in the worst case of O(m*n) where m is the length of the pattern and n is the length of the text. Karp and Rabin gave a Monte Carlo probabilistic algorithm for this problem which runs in the worst case in time O(m+n) [KAR86]. However the algorithm may give false matches but with "low probablity".

Section 4.4 presents probabilistic algorithms for finding an irreducible polynomial over a finite field and for root finding of a polynomial over a finite field. The probabilistic classes of these algorithms and their complexities are also given in 4.4. A deterministic solution to the problem of finding an irreducible polynomial over a field $GF(p^n)$ involves an exhaustive search through the field. Rabin [RAB80] gave a probabilistic solution to the problem which uses randomization in the search for these polynomials , and proved that the expected number of trials until an irreducible polynomial is found is n. A deterministic solution to the problem of finding a root of a polynomial over a field $GF(p^n)$ involves an exhaustive search through the field. Rabin [RAB80] gave a probabilistic solution to the problem which uses randomization in the search for these roots , and proved that the complexity of his algorithm is $O((n*L(n)^2 * (\log n + \log p))$ where $L(n) = \log n * \log \log n$.

CHAPTER II

THEORETICAL FOUNDATION OF PROBABILISTIC ALGORITHMS

2.1 Introduction

The class of probabilistic algorithms contains those algorithms in which the run time and the output of the algorithm are random variables and whose probability distribution may depend on the input of the algorithm [SCH85].

Probabilistic algorithms are formally modeled by non-deterministic Turing machines in which each non-deterministic choice is considered as a random experiment.

Before proceeding to explain the structure of those probabilistic algorithms we need to define some basic classes in complexity theory. The ***P*** class is defined to be the class of problems whose worst case complexities are of the form f(n) = O(p(n)) where p(n) is polynomial in n. Examples of these algorithms are sorting and searching problems. Non-polynomial time class is defined to be the class of problems whose worst case complexity can not be bound from above by a polynomial in the size of the input. Normally those are algorithms of exponential complexity, e.g. $O(2^n)$. Examples are the travelling salesman , satisfiability and the knapsack problems.

It is desirable to solve a given problem in polynomial time. When this is possible it is called "easy". However this is not always possible , and in this case the problem will be "hard".

Attention was directed to solve those hard problems using non-deterministic machines , i.e. machines that can "guess" or "generate" a solution and simultaneously verify a number of these solutions. Non-deterministic machines are not realistic , but they are of great theoretical importance.

The class of hard problems that can be solved in polynomial time on a nondeterministic machine is called ***NP***. It is clear that ***P*** is a subset of ***NP***.

Many researches have investigated these hard problems , and part of their effort was what Stephen Cook [COO71] showed that there is a problem in NP whose solution in polynomial time on a deterministic machine will imply that all problems in ***NP*** are also in ***P***, i.e. ***P=NP***. That problem was the satisfiability problem, ***SAT***.

Further classification of those hard problems can be done as follows :

- A problem is called ***NP-HARD*** if and only if ***SAT*** can be reduced to it. ***SAT*** can be reduced to a problem if there is a polynomial time transformation from ***SAT*** to that problem.
- A problem is called ***NP-COMPLETE*** if and only if it is ***NP-HARD*** and can be solved in polynomial time on a non-deterministic machine.

This classification implies that ***NP-COMPLETE*** is a subset of ***NP-HARD***. Furthermore it is known that there are ***NP-HARD*** problems that are not ***NP-COMPLETE*** such as the halting problem. In fact, the halting problem falls into a class of problems , known as "undecidable" problems , which are impossible to solve by any algorithm.

It is clear that ***NP-COMPLETE*** is a subset of ***NP*** But is ***P*** a subset of ***NP OR P=NP-COMPLETE*** ? This question has challenged many very capable people so far. At the present it is conjectured that ***P ≠ NP-complete***.

Assuming that ***P ≠ NP-complete,*** we define the NPI class as follows :

NPI = NP - (P U NP-COMPLETE).

This class denotes problems whose complexities are yet not proven to be in ***P*** or in ***NP-COMPLETE*** , i.e Open problems , [GAR79].

The classification of problems according to their complexities results in the classes ***P, NPI, NP-COMPLETE*** , ***NP-HARD*** , ***NP*** and the undecidable classes among many other classes.

Algorithms to solve
NP-HARD AND NP-COMPLETE problems will be costly to implement , and unfortunately many important practical problems fall into these classes.

The class of problems for which probabilistic solutions were successful so far is the NPI class. In this chapter we will present the structure of probabilistic algorithms , their types and their hierarchy.

2.2 Structure of Probabilistic Algorithms :

Probabilistic algorithms are defined in terms of languages accepted by a probabilistic Turing Machines (PTM). A PTM , as defined in [SCH85] , is a Turing machine with ditinguished states called coin tossing states. For each coin tossing state, the finite control unit specifies two possible next states. The computation of a probabilistic Turing machine is deterministic except that in coin-tossing states the machine tosses an unbiased coin to decide between the two possible next states. For simplicity and without loss of generality we assume that each nondeterministic branch in a PTM has two possible outcomes each assigne d probablity 1/2. A PTM has 3 types of Final states; 1-state (or accepting state) , 0-state (or rejecting state) and ?-state (or don't know state). Since each non-deterministic choice has probability 1/2, each nondeterministic path of length t has probability 2^{-t}.

2.3 Types of Probabilistic Algorithms :

Probabilistic algorithms in general are classified as :

- Las Vegas type : Probabilistic Algorithms that never give a wrong answer but may not give an answer at all (? final state) with "a low probability".

- Monte Carlo type : Probabilistic Algorithms that always give an answer but may produce wrong answers with "a low probability" , (in other words the machine may output 0 for some x in the language and may output 1 for some x not in the language). Before we proceed to list subtypes of the above two basic types we give some basic definitions :

1- ***Let*** Σ = basic alphabet = $\{0,1\}$.

2- ***Let*** Σ^* = set of all strings composed from Σ and $L \leq \Sigma^*$ denotes a language defined over Σ

3. $X \in L$, ***i.e.*** X is a sentence in L.

4- Let M(x) = outcome (type of final state) of a probabilistic Turing Machine M on input x , whose range is the set $\{0,1,?\}$

5- Let $X_A(x) = 0$ if x is not in the language and $X_A(x) = 1$ if x is in the language.

6- Let prob[M(x) = a] be the probability that M on input x halts in an a-state , a in {0,1,?} for more details see [SCH85] pages 30-31 and the references given there.

Now we list some special languages which fall in the Las Vegas class , or in the Monte Carlo class or in between these two classes

PP TYPE (PROBABILISTIC POLYNOMIAL TYPE) :

This is the class of languages $A \leq \Sigma^*$, for which there exists a probabilistic and polynomial time bounded Turing Machine M such that for each x in Σ^* prob[M(x) = $X_A(x)$] > 1/2. Clearly the Turing Machines accepting the above languages are of type Monte Carlo since they may "lie" with "small probability"

BPP TYPE (BOUNDED ERROR PROBABILISTIC POLYNOMIAL TIME) :

This is the class of languages $A \leq \Sigma^*$, for which there exists a probabilistic and polynomial time bounded Turing Machine M such that for each x in Σ^* prob[M(x) = $X_A(x)$] > 1/2+e where 0 < e < 1/2 . It is clear that BPP is a subset of PP, so BPP is also a Monte Carlo type.

R TYPE (RANDOM POLYNOMIAL TIME) :

This the class of languages $A \leq \Sigma^*$, for which there exists a probabilistic and polynomial time bounded Turing Machine M such that for each x in Σ^*

(1) $x \in A$ -> prob[M(x) = 1] > 1/2.

(2) $x \notin A$ -> prob[M(x) = 0] = 1.

This class is in a sense is a mixture of the two classes of "Las Vegas" and "Monte Carlo" types since the machine may lie on x in A and will not lie on x not in A. This class is called random since the algorithm depends on a random experiment for output. So if d is chosen between 0 and 1 such that prob[M(x) = 1] > d for x in A then prob [M(x) = 0] < d for x not in A. By iterating the random procedure k times the prob[M(x) = 0] < $(1-d)^k$. So by appropriate values for k it is possible to have prob[M(x) = 1] > 1/2 + e for any e , 0 < e < 1/2 , when x is in the language A [SCH85].

A good and important example of this class is the probabilistic algorithms for primality testing for which an abstract model is given below.

ABSTRACT MODEL FOR PROBABILISTIC PRIMALITY TESTING

The following is an abstract model outlining the basic steps in the probabilistic primality given by Monier in [MON80].

INPUT : n an odd integer

OUTPUT : n is prime or n is composite

NOTATIONS : T(a,n) is a predicate function of a and n. If n is prime then T(a,n) is false for all $a<n$. If n is composite then T(a,n) is true for some a's < n and false for the rest provided that the ratio of number of a's where T(a,n) is false to all a's is < 0.5. This ratio is denoted by α.

PROCEDURE :

```
For i = 1 to k do  (k can be determined from the
probability of error as will be seen in Chapter 2)
 Generate randomly a < n and test
 if T(a,n) then
 STOP and declare n to be composite
 endfor
 declare n to be prime with probability $\le 1-\alpha^{k}$
```

ZPP TYPE (ZERO ERROR POLYNOMIAL TIME) :

This is the class of languages $A < \Sigma^*$, for which there exists a probabilistic and polynomial time bounded Turing Machine M such that for each x in Σ^*

(1) $x \in A$ -> prob[M(x) = 1] > 1/2 & prob[M(x) = 0] = 0.

(2) $x \notin A$ -> prob[M(x) = 0] > 1/2 & prob[M(x) = 1] = 0.

This class belongs to the "Las Vegas"-type since the machine does not lie but may not halt. The ***ZPP*** class is by definition the intersection of ***R*** and ***CO-R*** where the class ***CO-R*** = { ***A*** $\leq \Sigma^*$ such that the complement of A in R}.

From the previous defintions the folloing hierachy of complexity classes is obtained :

P ≤ ZPP = R U Co-R ≤ BPP ≤ PP

The question of wheather ***BPP≤NP or NP≤BPP*** is still an open problem. For basic references on these classes see [GIL77] , [SCH85] and [ZAC82] .

CHAPTER III
PROBABILISTIC ALGORITHMS FOR PRIMALITY TESTING

3.1 Introduction

The problem of testing an integer for primality has been a source of interest and fascination to mathematicians since Euclid. In the past two decades it generated even more interest because of its applications in cryptography.

Probabilistic and non-probabilistic algorithms were developed for testing primality. Non-probabilistic algorithms give a definite answer whether the integer being tested is composite or prime. However the cost i.e computation time , of these algorithms becomes very high as the size of the number grows.

To be more precise the complexity of these non-probabilistic algorithms discovered so far is not strictly polynomial unless extra assumptions are made as in the case of Miller's algorithm , see [MIL76]. The slowest algorithm for testing primality is the one based on trial divisions which is exponential in terms of the length of the tested number and the fastest one in literature today is the Pomerance Adelman-Rumely algorithm [ADL83] and the Cohen-Lenstra version of it , [COH87] , which takes about 10 minutes to test the primality of a 200 digit number. The computational complexity of this fast algorithm can be expressed as a function of the size of the integer to be tested, say n, as:

$d(n)^{(c*d(d(d(n))))}$, *where*

d(n) is the number of digits in n, and c is a constant.

Here the algorithm is exponential in terms of the number of bits of the number of bits of the number of bits of the tested number.

Gary Miller also gave a non-probabilistic algorithm which runs in polynomial time $(O(log^5(n))$ but the algorithm requires the Extended Riemann Hypothesis to be true. For more details see [MIL76] .

Probabilistic algorithms for primality tests work fast but may, on very rare occasisons, declare a composite number to be prime. The probability of such error could be reduced as much as less than d where $0 < d < 1$ by iterating the procedure as often as needed.

Two of the first and most well known probabilistic methods are the Solovay-Strassen method [SOL77] and the

Rabin method [RAB76]. These methods are presented in the next two subsections and an implementation of them in C is given in Appendix A.

Those algorithms are of type R (random polynomial time) as explained in Chapter 2. However, some classify these algorithms as of the Monte Carlo type since they always given an answer, but may lie some of the time.

Most of the probabilistic primality test algorithms are based on Fermat's theorem which states that $b^{(n-1)} \equiv 1 (mod\ n)$ if n is prime or b is relatively prime to n. The test based on this theorem is called the pseudoprimality test. These algorithms go a step beyond the use of this test to establish the primality or compositeness of an integer, n, by subjecting n to more tests such as the quadratic residue test in the case of the Solovay-Strassen algorithm and the Miller test in the case of the Rabin's algorithm.

This chapter investigates the issue of utilizing the pseudoprimality test alone as a test for primality and studies the probability of composite integers passing this test. It is well known that many composite integers can pass this test, most notably of these are the Carmichael numbers.

An algorithm is developed to test primality using the pseudoprimality test alone and an upper bound on the probability of error, i.e. of an odd composite integer passing the test, is derived and given in section 3.5 .

3.2 Mathmatical Preliminaries:

Number Theory serves as the basis for studying and developing algorithms for primality. We start with one of the basic definitions in number theory which is Congruence.

a is said to be congruent to b modulo n if n|(a-b) (| means divide) and it is denoted by $a \equiv b$ (mod n). Now we list some congruence rules.

if $a \equiv b$ (mod n) then $a \; op \; c \equiv b \; op \; c \; (mod \; n)$ where op is

+ , - , * .

if a * $c \equiv b * c$ (mod n) and d = gcd(c,n) then

$a \equiv b \; (mod \; n/d)$

The basic theory used for developing probabilistic algorithms for primality testing is Fermat's theorem stated as:

***FERMAT THEOREM*:**

If n is prime and a is a positive integer such that gcd(a,n) = 1, then: $a^{(n-1)}$ mod n $\equiv$ 1. This is one form of Fermat Theorem , and there are other equivalent forms to this form , see [HAR79].

EULER'S GENERALIZATION OF FERMAT THEOREM:

Euler extended Fermat's theorem as follows:

Let $\varphi(n)$ be defined as the number of positive integers less than n and relatively prime to n.

If n is a positive integer and a is an integer such that gcd(a,n) = 1 then $a^{\varphi(n)} \equiv 1 (mod\ n)$.

THE EXPONENT OF AN INTEGER MODULO N:

If gcd(a,n) = 1 then there exists δ such that a^{δ} mod n ≡ 1 furthermore there is no $\beta<\delta$ such that $a^{\beta} \equiv 1$ mod n and, $\delta | \lambda(n)$ and $\delta | \varphi(n)$, [CAR15] , where $\lambda(n)$ is the minimum universal exponent defined below.

UNIVERSAL EXPONENT:

U is called a universal exponent for n if for all a relatively prime to n a^{U} mod n = 1, and the minimum universal exponent is defined as the smallest integer satisfying the property of the universal exponent [CAR15].

Let $n=p_1^{e_1} * p_2^{e_2} \ldots * p_m^{e_m}$ be the prime power factorization of an odd integer n, and let $\lambda(n)$ be the minimum universal exponent, then $\lambda(n)$ can be computed as follows, [ROS84] :

$$\lambda(n) = LCM[\varphi(p_1^{e_1}), \varphi(p_2^{e_2}), \ldots\ldots\ldots, \varphi(p_m^{e_m})].\ i.e.,$$

$$LCM[p_1^{e_1-1} * (p_1-1), p_2^{e_2-1} * (p_2-1), \ldots\ldots\ldots\ldots p_m^{e_m-1} * (p_m-1)]$$

except when 8 | n then $\lambda(n) = LCM[2^{(a-2)}, \lambda(n/2^{a})]$ *where* 2^{a} is the largest power of 2 that divides n.

$\lambda(n)$ is sometimes referred to in the literature as the Carmichael function, e.g. [REI87] .

It is known , e.g. [POM81] , that n is a Carmichael number if and only if $\lambda(n) | (n-1)$

***PSEUDOPRIMES*:**

n is called is pseudoprime to the base b if n is positive integer and $b^{(n-1)} \equiv 1 \bmod n$

***CARMICHAEL NUMBERS*:**

A composite integer n which satisfies $b^{(n-1)} \equiv 1 \pmod{n}$ for all positive integers b relatively prime to n is called a Carmichael number.

***QUADRATIC RESIDUE* :**

a is called a quadratic residue of n if gcd(a,n) = 1 and the congruence $x^2 \equiv a \ (mod\ n)$) has a solution , otherwise a is called a non quadratic residue of n.

LEGENDRE SYMBOL :

When n is an odd prime the legendre symbol (a|n)

= +1 if a is a quadratic residue of n.

= -1 if a is a non quadratic residue of n.

Euler critera can be used to test if a given integer is a quadratic residue of n or not without trying to solve the congruence $x^2 \equiv a \ (mod\ n)$). Euler's critera states that if gcd(a,n) = 1 and n is an odd prime then $(a|n) \equiv a^{\frac{(n-1)}{2}} \ (mod\ n) \equiv \pm 1$ (mod n). Also Euler's critera can be used as a compositeness test , i.e. if (a|n) $\not\equiv \pm 1$ (mod n) then n is composite .

If p and q are primes then (p|q) = (q|p) $(-1)^{\frac{1}{2}(p-1)(q-1)}$ This is known as the law of Quadratic Reciprocity , see [BAK84]. Euler's critera helps to find which a's are quadratic residue mod n while the quadratic reciprocity law helps to find p's such that a is a quadratic residue. The legendre symbol is defined only when n is prime , the general form of the legendre symbol is called the jacobi symbol $\frac{a}{n}$. , see [BAK84].

JACOBI SYMBOL J(A,B): $(\frac{a}{n}) = \prod (a|p_i)^{\alpha_i}$, if $n = \prod p_i^{\alpha_i}$,. The jacobi symbol will not help to decide whether a is a quadratic residue mod n or not , however as it will be shown in section 3.3 the jacobi symbol and the quadratic residue together can be used to test probabilistically if a given integer n is prime or not.

```
J(a,b) is defined recursively as
  if a = 1 then J:=1
  else if a mod 2 = 0 then begin
  if (b*b-1)/8 mod 2 = 0
  then j := j(1/2,b)
  else j := -j(1/2,b) end
  else if (a-1) * (b-1)/4 mod 2 = 0
  then j:=j (b mod a)
  else j := -j(b mod a,a) see [DOR84] for this defintion.
```

3.3 Primality Testing Using The Solovay & Strassen Method

3.3.1 Introduction

Solovay & Strassen in 1977 , [SOL77] , developed a probabilistic algorithm for primality testing using the quadratic residue as a witness for compositeness.

The idea of a witness of a composite number is based on the concept that there is a test based on a random number less than n , the integer tested for primality , such that if n is prime then it will pass the test with all the numbers less than n , while if n is composite then it will pass the tests only for some portion of the numbers less than n. The number for which the test fails is the witness for n's compositeness. The ratio of the numbers which pass the test to all the numbers less than n is called the probability of error.

Repetition of the witness test of k independent trials reduces the probability of error to be $<pe^k$ (where pe is the probability of error).

It is worth noting here that the polynomial time deterministic algorithm by Miller assumes the truth of the extended Riemann Hypothesis because the truth of the hypothesis will imply that the number of witnesses needed to establish the primality of N $<2(\ln N)^{2}$. [WAG86]

Let $L_s(n)=\{1 \le a < n$ such that $a^{(n-1)/2} \equiv J(a,n) \bmod n\}$, where n is composite. The elements of $L_s(n)$ represent false witnesses, or liars as called by Monier [MON80] . The size of $L_s(n)$, or number or liars , when n is composite was shown to be :

$|L_s(n)| \le \varphi(n)/2.$, see [MON80] .

Solovay & Strassen used a witness based on the quadratic residue and they proved an upper bound on the probability of error for each witness test to *be* ≤ 0.5.

The cost of the algorithm was upper bounded by 6 * log n.

This algorithm will always terminate but may give wrong answers as pointed above, and therefore it belongs to the Monte Carlo class.

3.3.2 The Algorithm:

INPUT : n an odd integer, E an upper limit on the error probability.

OUTPUT :

n is composite or n is prime with pr(error) $\le E$.

NOTATIONS : J(a,b) = Jacobi symbol.

PROCEDURE PRIME_S:

```
    1.  k = - ⌈$\log_2 E$⌉

    2.  For i:= 1 to k do

          M = random(n).

          if (gcd(M,n) $\neq$1) or ($M^{(n-1)/2} \not\equiv J(M,n) \bmod n$) then

          Stop and Output n is composite.

        ENDIF

      ENDFOR

  3.  Output n is prime with pr(error) $\le E$.

END PROCEDURE PRIME_S.
```

3.4 Rabin Probabilistic Algorithm for Primality Testing

3.4.1 (Original Version)

***INTRODUCTION*:**

Michael Rabin in 1976 [RAB76] presented a probabilistic algorithm for primality testing which has the same concept as the solovay and strassen algorithm in the sense that primes will never be declared to be composite but composites may be declared as primes with some probability of error.

Rabin used Miller's Test as a witness for compositeness. Miller's test is based on the following concepts:

Let n be a positive integer with $n-1 = 2^s * t$ where s is a nonnegative integer and t is an odd positive integer. We say that n passes Miller's test for the base b if either :

1- $b^t \equiv 1 \pmod n$ or

2- $b^{2^{j*t}} \equiv -1 \pmod n$ for some j with $0 \le j \le s-1$.

Let $L_m(n) = \{a \text{ whereas } a^t \equiv 1 (mod\ n)$ or $a^{t*2^j} \equiv 1\ (mod\ n)\}$ *where* $0 \le j < s$ when n is composite. The elements of $L_m(n)$

represent false witnesses, or liars. The size of $L_m(n)$, or number of liars, when n is composite has been shown to be :

$$|L_m(n)| = (1+2^{(r*v)}-1/(2^{(r-1})) \prod_{i=1}^{r} (gcd(t,p_i))$$

where $n = p_1^{\alpha_1} * p_2^{\alpha_2} \ldots . p_r^{\alpha_r}$ and $v = \min(\alpha_i)$ for $1 \le i \le r$. [MON80].

Rabin gave a better algorithm than SOLOVAY's in terms of probability of error since the probability of error for each witness test is < 1/4 , whereas the Solovay & Strassen error is$\le$1/2. On the other hand the cost of Rabin's algorithm in the worst case = 2 * log n +s * log n which is higher than Solovay's when s > 4.

This algorithm will always terminate but may give wrong answers as pointed above, and therefore it belongs to the Monte Carlo class.

THE ALGORITHM:

> ***INPUT*** : n an odd integer, E an upper bound on the probability of error.
>
> ***OUTPUT*** : : n is composite or n is prime with pr(error) $\le E$.

***PROCEDURE PRIME_R*:**

```
1.  k = - $\frac{\lceil \log_2 E \rceil}{2}$

2.  Generate randomly $b_1, b_2, b_3, \ldots b_k$ where each $b_i$ in
    [0..n-1] .

3.  Compute M, L such that $n-1 = 2^L * M$ where M is
    odd.

4.  For i : = 1 to k do
    d = $b_i^M$ mod $n$.

    For J := 1 to L do
         Compute and Store $b_i^{(2^J * M)}$.
    endfor
    For J := L down to 1 do
         test if gcd(rem $(b_i^{2^J * M} - 1), n) \neq 1$
         then exit and output n is composite.

    endfor
endfor
5.  Output n is prime with $pr(error) \leq E$
```

```
END PROCEDURE PRIME_R
```

3.4.2 An Improvement of the Rabin Algorithm:

An improvement given by Knuth, [KNU81] is based on the fact that if n = 1 + 2^k*q is prime and $x^q*t \neq 1 (mod\ n)$ then the sequence x*t mod n , x^2*t mod n ,................ $x^{(2^s)}*t$ mod n will end with 1, and the value just preceding the first appearance of 1 will be n-1. So once $b_i^{(2^j)}*t=n-1$ (mod n) there is no need to continue the test of the remaining values of j.

Knuth also proved that the probability of error for composite numbers of each witness test is *still* ≤ 0.25.

***THE ALGORITHM*:**

RABIN PROBABILISTIC ALGORITHM FOR

PRIMALITY TESTING

(Improved Version)

INPUT : n an odd integer, E an upper bound on the probability of error.

OUTPUT : : n is composite or n is prime with pr(error) < = *E*.

***PROCEDURE*:**

```
1. k = - upper $\frac{\lceil \log_2 E \rceil}{2}$

2. For i = 1 to k do

3. Generate x randomly in [0...n-1] .

4. Set j = 0 and y = $x^q$ mod $n$.

5. If j = 0 and y = 1 or if y = n-1 say that n is
   probably prime and go to the next i. If j > 0 and y
   = 1 declare n is composite and halt the algorithm.

6. Increase j by 1.  If j < k set y = $y^2$ mod n and
   goto step # 5, else declare n to be composite and
   halt the algorithm.

   ENDFOR

7. N is prime with probability of $error \le E$.
```

A comparison of the Rabin and the Solovay-Strassen methods done by Monier [MON80] , shows that Rabin test is more efficient than Solovay-Strassen's test.

The comparison was based on the ratio of the cost and the logarithm of its probability of error, say u. $u=kt/-\log(\alpha^k)$, so $u=t/-\log(\alpha)$ where k is number of iterations of the loop, t is the cost of each loop and $\alpha = 2$ for Solovay's and = 4 for Rabin's. Furthermore Monier has shown that any liar to Rabin's test is also a liar to the Solovay-Strassen test.

Some mathmaticians refer to the Solovay-Strassen's and Rabin's Algorithms as "Compositeness testing Algorithms" because they establish or prove the compositeness of an integer with certainty and don't prove the primality of an integer.

3.5 Primality Testing Utilizing the Pseudoprimality Test Alone

3.5.1 Introduction:

Most modern primality tests have arisen from the Fermat's theorem. This Theorem is the basis of the pseudoprimality test as mentioned earlier, but the main drawback of this test compared to Rabin's or Solovay's tests is that the Carmichael numbers will always pass it , and the pseudoprimes may pass it with probability similar to Solovay's probability of error as will be shown later.

In this section we will investigate the question of how useful is the pseudoprimality test alone in testing primality by finding an upper bound on the probability of a composite integer passing this test. It is also worth noting that $a \in \{2,5,7,13\}$ is enough to witness the compositeness of any $\textit{integer} \leq 25 \times 10^9$ [BEA88].

3.5.2 An Algorithm for Testing Primality Using the Pseduoprimality Test

The algorithm presented below is based on the pseduoprimality test alone and the analysis will lead to the upper bound sought.

***THE ALGORITHM*:**

```
INPUT : n an odd integer ≥ 5.
OUTPUT : n is composite or {n is prime with pr(error)}

PROCEDURE PTEST:
For i := 1 to k do
  M = random(n)
  if (gcd(M,n) ≠1) or ($M^n$ mod $n \neq M$)
  then output "n is composite" and STOP
ENDFOR
  output "n is prime"
END PROCEDURE.
```

It is clear that the procedure above will terminate since the loop will be executed exactly k times. However, the answer that n is prime may be wrong as shown in lemma 3 below , and therefore this algorithm belongs to the Monte Carlo class.

Let n be an odd integer and define the following sets :

Gr = { a such that gcd(a,n) = 1},

Gl = { a such that gcd(a,n) = 1 and $a^n \bmod n = a$},

S1 = { a such that gcd (a,n-1) = 1 and a < n-1}

Now for any a_i in S1 there is a unique integer a_j also in S1 such that $a_i * a_j$ mod n-1 = 1. For more details see [ROS84] pp 102-104.

Let (E,D) be a pair of elements in S1 much that $E*D \equiv 1 (\mathit{mod}\ n-1)$

and let

Gm = {a such that gcd(a,n) = 1 and $a^{(E*D)} \bmod n \equiv 1$}.

$\lambda(n)$= minimum universal exponent for n,

= modulo n multiplication.

***LEMMA 1*:**

(Gm,#) is a subgroup of (Gr, #)

***PROOF*:**

It is known that Gr is a group, and it is clear that Gm is a subset of Gr.

1. Closure of (Gm,#)

   If a_1 , a_2 in Gm then we need to prove that $a_3 = a_1 \# a_2$ is also in Gm.

   if a_1 in Gm then $gcd(a_1,n)=1$ and $a_1^{(E*D)} \bmod n \equiv a_1$ *also* if a_2 in Gm then $gcd(a_2 ,n) = 1$ and $a_2^{(E*D)} \bmod n \equiv a_2$.

   Let $a_3 = a_1 \# a_2, gcd(a_3,n) = gcd(a_1 * a_2 \bmod n, n) = 1$.

   Also $a_3^{(E*D)} \bmod n \equiv (a_1 * a_2 \bmod n)^{(E*D)} \bmod n$

   $= (a_1^{(E*D)} \bmod n) * (a_2^{(E*D)} \bmod n) \bmod n = a_1 * a_2 \bmod n = a_3$.

2. Associativity follows from Gr being a group

3. Identity = e = 1 in Gm because $1^{ed} \bmod n = 1 \bmod n = 1$.

4. Unique inverse exists in Gr as shown above, but we need to show that if a is in Gm then it has a unique inverse in Gm. i.e. if a * b mod n ≡ 1 has a unique solution in Gr and a is in Gm we need to show that b is in Gm.

   To show this note that gcd(b,n) = 1 since b is in Gr and since a#b = 1 then $(a\#b)^{(E*D)}$ mod n = 1, which implies that $(a^{(E*D)} \bmod n)$ # $(b^{(E*D)} \bmod n)$ mod n = 1. But $a^{(E*D)} \bmod n = a$ since a is in Gm, then a# $(b^{(E*D)} \bmod n) = 1$.

   Multiply both sides by b to get $b\#a\#(b^{(E*D)} \bmod n = b$, but since b#a = 1 then $b^{(E*D)} \bmod n = b$. Therefore, b is in Gm and hence Gm is a subgroup. QED.

Note that Gm may be a non proper subgroup of Gr as well as being a proper subgroup. For example let n = 9, E = 3 and D = 3 then Gr = {1,2,4,5,7,8}, Gm (3,3) = {1,8}, here Gm is a proper subgroup of Gr. If E is changed to 5 and D to 5 then Gm (5,5) = Gr and here Gm is a non-proper subgroup of Gr.

***LEMMA 2*:**

(Gl,#) is a subgroup of (Gm,#).

***PROOF*:**

(Gm,#) is a group since it is a subgroup from Lemma 1.

To prove that (Gl,#) is a subgroup of (Gm,#) note that Gl is a subset of Gm since a in $Gl \rightarrow a^{n} \bmod n \equiv a \rightarrow a^{(n-1)} \bmod n = 1$

$\rightarrow a^{t*(n-1)} \bmod n \equiv 1 \rightarrow a^{t*(n-1)+1} \bmod n \equiv a$

$\rightarrow a^{(E*D)} \bmod n \equiv a.$

$\rightarrow$a in Gm $\rightarrow$ Gl is a subset of Gm.

To prove that Gl is a group:

1. Closure of (Gl,#)

If a_1, a_2 in Gl then we need to prove that $a_3 = a_1 \# a_2$ also in Gl.

If a_1 in Gl then *gcd*(a_1 ,n) = 1 and $a_1^{n} \bmod n = a_1$ *also*

If a_2 in Gl then $gcd(a_2\ ,n) = 1$ and $a_2^{\ n} \bmod n = a_2$.

Let $a_3 = a_1\#a_2$, $gcd(a_3,n) = gcd(a_1 * a_2 \bmod n, n) = 1$,

Also $a_3^{\ n} \bmod n = (a_1 * a_2 \bmod n)^n \bmod n$

$= (a_1^{\ n} \bmod n) * (a_2^n \bmod n) \bmod n = a_1 * a_2 \bmod n \equiv a_3$.

2. Associativity follows from Gm.

3. Identity = e = 1 in Gl because $1^n \bmod n = 1 \bmod n = 1$.

4. Unique inverse exists in Gm as shown in Lemma 1, but we need to show that if a is in Gl then it has a unique inverse in Gl. i.e. if a * b mod n = 1 has a unique solution in Gm and a is in Gl we need to show that b is in Gl.

To show this note that gcd (b,n) = 1 since b is in Gm.

Now since a#b = 1 then $(a\#b)^n$ mod n = 1, which implies that $(a^n \bmod n)\#(b^n \bmod n) \bmod n = 1$.

But a^n mod n = a since a is Gl, then $a\#(b^n \bmod n) = 1$.

Now buy multiplying both sides by b we get:

$b\#a\#(b^n \bmod n)$ = b, but since b#a = 1 then $b^n \bmod n = b$.

Therefore b is in Gl and hence Gl is a subgroup. QED.

Note that Gl may be a non proper subgroup of Gm as well as being a proper subgroup. For example let n = 9, E = 5, D = 5, then Gm (5,5) = (1,2,4,5,7,8) and Gl = {1,8} here Gl is a proper subgroup of Gm. If E is changed to 3 and D to 3 then Gm = {1,8} = Gl, and in this case Gl is a non proper subgroup of Gm.

The order of Gl is equal to the number of solutions of $a^{(n-1)}$ mod n for a relatively prime to n, and this can be shown to be:

$= \prod gcd(n-1,p-1)$ where p divides n. [ERD86].

Of course the exact computation of |Gl| requires one to know the prime factorization of n.

Bond in [BON84] discussed Fermat test , Euler test and Miller test. He also presented how to calculate the probability of error in each test by defining the sets B(n) = number of bases for n , E(n) = number of Euler bases for n and S(n) = number of strong bases for n. He gave a formula for B(n) :

Let $n=p_1^{e_1} * p_2^{e_2} \dots * p_m^{e_m}$

$=2^r d+1$ where d is odd

$p_i=2^{r_i} d_i+1$ *where* d_i is odd and $S_0=\prod_{i=1}^{m} gcd(d_i,d)$ Then B(n) = $\prod_{i=1}^{k} 2^{min(r,r_i)}$. Again this requires the prime factorization of n.

From Lemmas 1 and 2 Gl is a subgroup of Gr, so in summary we can say that $Gl \le Gm \le Gr$.

Since Gl is a susbgroup of Gr then |Gl| divides |Gr| by Lagranges theorem or |Gl| divides $\varphi(n)$. This result is also mentioned by *Erdős* and Pomerance see [ERD86] .

Now if for some n we have a situation where (Gr=Gl) then such n is either prime or a Carmichael number by virtue of Fermat's theorem, the properties of the Carmichael numbers and the construction of Gr and Gl.

Since (Gr=Gl) implies that (Gm=Gr) for at least one pair (E,D) in Sl then the probability of $(Gl=Gr) \le$ probability of (Gm=Gr) for a pair (E,D) in S1.

***LEMMA 3*:**

1. If the input n is prime then procedure PTEST will never output "n is composite".

2. If n is composite then PTEST may declare it to be prime.

***PROOF*:**

1. If n is prime then for all $M < n$ gcd $(M,n) = 1$ and

   M^n mod = M as guaranteed by Fermat Theorem.

2. As a counter-example consider $n = 561 = 3 * 11 * 17$, by brute force computation of Gr and Gl we can find that Gr = Gl and hence PTEST will declare 561 to be prime.

The preceding analysis shows that procedure PTEST will not declare a prime to be composite, but may declare a composite to be prime. This situation leads one to ask " ***IF N IS COMPOSITE WHAT IS THE PROBABILITY THAT PROCEDURE PTEST WILL DECLARE IT AS PRIME?*** " this probability will be referred to as the probability of error.

Since for some n the resulting Gl is a subgroup of Gr then |Gl| divides |Gr| and one of the following two cases will be true:

Case I : $Gr \neq Gl$ which will imply n is composite and $|Gl| \leq 0.5*|Gr|$

or

Case II: $Gr = Gl$ and this is the case when n is either a prime or a Carmichael number.

***LEMMA 4*:**

Gr = Gm if and only if E * D mod $\lambda(n)=1$, *where* $\lambda(n)$ is the minimum universal exponent of n.

***PROOF*:**

1. E*D mod $\lambda(n)=1$ implies Gr = Gl:

   If E * D mod $\lambda(n)=1$ then E * D = k1* $\lambda(n)$ * a mod n

   $= (a^{\lambda(n)^{k1}} * a)$ mod n $= 1 * a$ mod n =

   since $a^{\lambda(n)}$ mod n = 1 for all a relatively prime to n which implies that (Gr = Gl).

2. Gr = Gl implies E*D mod $\lambda(n)=1$: (proof by contradiction.)

   Assume that (Gr = Gl) and E*D mod $\lambda(n)\neq 1$

   then E * D = $k1*\lambda(n)+r(1<r<\lambda(n))$.

   therefore, a^{E*D} mod n = $a^{\lambda(n)^{k1}} * a^{r}$ mod n

   $= a^{r}$ mod n = 1 (*since Gr = Gl*)

   and since r < $\lambda(n)$ *then* $\lambda(n)$ is not the minimum universal exponent of n, which leads to a contradiction!

***LEMMA* 5**: Let pd be the probability of (Gr = Gl) then

$$pd \leq \frac{1}{\varphi(\lambda(n))}$$

PROOF:

Let S2 = {a such that gcd (a, $\lambda(n)$)=1 and $a < \lambda(n)$}

The probability of (*Gr*=*Gl*) ≤ ***probability*** of (Gr = Gm) for a pair (E,D) in S1 since (Gr = Gl) implies that (Gr = Gm) but (Gr = Gm) does not necessarily imply that (Gr = Gl) (see the examples of Lemmas 1 and 2).

(Gr = Gm) if and only if E*D mod $\lambda(n)=1$ (From Lemma 4) and hence prob (Gr = Gm) = prob (E*D mod $\lambda(n)=1$) where E, D as used in the construction of Gm.

Now if one picks a random E from S1, this E defines a pair (E,D) where D is also in S1 such that E and D are mutual inverses modulo n-1 and then to have E*D mod $\lambda(n)=1$ three events have to happen:

1. gcd(E, $\lambda(n)$)=1 (otherwise E will not have an inverse mod $\lambda(n)$ which implies E*D mod $\lambda(n) \neq 1$).

Note that gcd(E, $\lambda(n)$)=1 implies that E mod $\lambda(n)$ is in S2.

2. gcd(D, $\lambda(n)$)=1 (for the same reason as 1)

   Note that gcd(D, $\lambda(n)$)=1 implies that D mod $\lambda(n)$ is in S2.

   Note that (1) & (2), i.e. E mod $\lambda(n)$ in S2 and D mod $\lambda(n)$ in S2 are not sufficient to say that they are inverses modulo $\lambda(n)$. *For* example if n = 15, E = 9 and D = 11. Then n-1 = 14 and $\lambda(n)$=4 Now gcd (E, $\lambda(n)$)=1 and E mod $\lambda(n)$=1 is in S2, the same for D, gcd (D, $\lambda(n)$)=1 and D mod $\lambda(n)$=3 is in S2. However (E mod $\lambda(n)$) * (D mod $\lambda(n)$) mod $\lambda(n)$ = 3 mod 4 = 3 $\neq$1.

   Therefore, the following extra condition is needed :

3. Given (1) there exist an inverse of E and call it D' and , given (2), D mod $\lambda(n)$ is in S2 then the condition says that

$$D' \equiv D \bmod \lambda(n).$$

The probability of these three events happening simultaneously whether they are dependent or not is obviously is less than or equal to the probability of the third event. Therefore :

$pd \le prob(D'=D \bmod \lambda(n)) = \frac{1}{\varphi(\lambda(n))}$ since there are $\varphi(\lambda(n))$ possibilities for such D'. Now this is the probability that a number is prime or Carmichael which implies that

$$pd \le \frac{1}{\varphi(\lambda(n))}.$$

The estimation above will not require the prime factorization of n to be known if one uses the lower bound on the Euler totient function $\varphi(x)$ and find a lower bound on $\lambda(n)$. And this will be in contrast with the estimates given by *Erdös* and by Bond stated earlier in page 42.

***THEOREM*:**

Procedure PTEST will not declare a prime number to be composite but may declare a composite number to be prime with ***probability*** $\leq$ ***pd***+(1-***pd***)*0.5^{k}.

***PROOF*:** From Lemma 3 the first part of the statement is true and if the number is composite then PTEST may declare it to be prime because either

1. (Gr = Gl) with ***probability*** $\leq$ ***pd*** where pd is the probability calculated in Lemma 5 , or
2. (Gr $\neq$ Gl) with probability (1-pd). This is the case when pseudoprimes pass the test.

Now when (Gl $\neq$ Gr) for each trial of PTEST the probability that it will not be declared as composite *is*$\leq$0.5 as seen before, (Since $|Gl| \leq 0.5*|Gr|$), and hence the probability that it will not be declared as composite for k independent trials for (*Gr*$\neq$*Gl*) is equal to (1-***pd***)*0.5^{k}.

Therefore the overall probability of error = pd+(1-pd)* 0.5^{k}.

***COST OF PROCEDURE PTEST*:**

It is clear that the cost of the procedure is =

cost of computing gcd (M,n)

+ cost of computing (M^n mod n) = 1.5 log n + 2.5 log n = 4 log n. For estimates of these costs see [KNU81] .

The cost of the procedure is less than that of the Solovay's and Rabin's although it is of the same order (i.e. Solovay upper bounded the cost by 6 log n , see [SOL77] , and Rabin did so by 2 * log n + 1 * log n see [RAB80] , and this obviously is due to the fact that PTEST performs only one test, i.e. the psuedoprimality test.

***COROLLARY 1*:**

As n grows larger the probability that it is a Carmichael number grows smaller and in the limit (*as* $n \to \infty$) the probability goes to zero.

***PROOF*:**

Let $n = p_1^{e_1} * p_2^{e_2} \ldots\ldots p_m^{e_m}$.

As n goes to ∞ ,$\lambda(n)=$ $p_1^{e_1-1}*(p_1-1)$, $p_2^{e_2-1}*(p_2-1)$, $\ldots p_m^{m-1}*(p_m-1)$ will also go to ∞. This is true since

$\lambda(n)\geq p_j^{e_j-1}*(p_j-1)$ *where* $p_j^{e_j-1}*(p_j-1)$ is the maximum of those $p_i^{e_i-1}*(p_i-1)$ for i = 1 to m. Moreover if n goes to ∞ either one of $p_i^{e_i}$ is infinite or number of those $p_i^{e_i}$ is infinite , but in this case also one of those p_i should be infinite since *every* p_i *is distinct*. Consequenctly one of those $p_i^{e_i-1}*(p_i-1)$ will also go to ∞ and hence $\lambda(n)$ *will go* ∞. $\varphi(\lambda(n))$ will go to zero since $\varphi(\lambda(n)) > \frac{\lambda(n)}{4*log(\lambda(n))}$ see [BAK84] , then finally $\frac{1}{\varphi(\lambda(n))}$ or pd will go to zero as n goes to ∞,*QED*.

3.5.3 An Algorithm for Testing If A Given Integer Is Carmichael or Not

```
PROCEDURE CARMICHAEL_TEST:
     FOR I : = 1 TO K DO

       M = random (n)

       If ($M^n$ mod n ≠ M) then

           output "n is not Carmichael and STOP.

     ENDFOR

     FOR I := 1 TO K DO

       M = random(n)

       If Rabin test on (n,M) output "composite" then

           output "n is probably Carmichael" and STOP

       END

     ENDFOR

           output "n is not Carmichael"

END PROCEDURE.
```

The probability of wrong decision (as will be proved in the next *corollary* $\leq 0.5^{k}$ *where* k could be computed as = - $\lceil \log_2 E \rceil$ where E is the probablity of error) .

***COROLLARY 2*:**

1. An odd composite integer declared in the above Procedure as Carmichael is a Carmichael number with *probability*$\geq (1-0.5^{k})$.

2. An odd composite integer declared in the above Procedure as not Carmichael is not Carmichael with *probability*$\geq (1-0.25^{k})$.

***PROOF*:**

1. The procedure will declare an integer to be Carmichael if the integer passes k pseduoprimality tests and does not pass one Rabin test and in this case n is definitely composite. Now either n is a Carmichael number or not, the probability that it is not a Carmichael , i.e. $Gr \neq Gl$, (and still declared by the above procedure as Carmicheal) *is*$\leq 0.5^{k}$ so the probability that n is a *Carmichael* $\geq 1-0.5^{k}$.

2. If the above Procedure declares that n is not a Carmichael number then this was based on either :

   a- The test M^n mod n $\neq$ M in the pseudoprimality test, and in this case the decision is definite by the virtue of the definition of the Carmichael numbers , or

   b- The integer n passed k iterations of pseudoprimality test and k iterations of Rabin test. In this case the probability that n is *composite* $\leq 0.25^k$ (Note here that we do not need to multiply both probabilities since the pseudoprimality test is part of the Rabin test). Therfore the overall probability that n is a Carmichael number *is* $\geq 1-0.25^k$

This algorithm is of type Monte Carlo , (Specifically the BPP class) since it may lie but always gives an answer.

The three approaches Rabin, Solovay and Strassen and the pseudoprimality test alone were implemented in C. The implementation is based on long digit operations, so all variables are defined as a consecutive sequence of words. All necessary operations were implemented for those variables (addition, multiplication and modular multiplication, fast exponentiation, greatest common divisor and many others). Those routines serve as a library of all long integers operations.

A test was done on random data for different ranges of integers using the three tests , Rabin , Solovay and the Psedouprimality test. Table 1 summarizes the results.

Length of n in Words (16 bit)	Number of Tests	Percentage of agreement of pseduoprimality to the other two
1	7000	99.971
3	50	100.00
5	20	100.00
7	10	100.00
10	10	100.00
13	10	100.00
17	10	100.00
20	10	100.00

Table 1 : Empirical results of testing random integers under the three methods and the percentage of agreement of the pseduoprimality test with the other two.

3.5.4 A Proposed Algorithm for Generating Keys for the *POHLIG_HELLMAN SCHEME*

***INTRODUCTION*:**

Pohlig and Hellman [POH78] proposed an encryption scheme based on computing exponentials over a finite field. The scheme enciphers a message block M in (0..n-1) by computing the exponential

$$C = M^e \bmod n \; \ldots \qquad (1),$$

where e and n form the key to the enciphering transformation. M is restored by the same operation but using a different exponent , d , for the key:

$$M = C^d \bmod n \; \ldots \qquad (2).$$

The above procedure is based on Euler's generalization of Fermat Theorem which says that $M^{\varphi(n)} \bmod n = 1$. This property implies that if e and d satisfy the relation e * d mod $\varphi(n) = 1$, then (1) and (2) are inverses [DORO82] . The security of the system is dependent on choosing the modulus n to be a large prime.

Key generation is done usually by producing large primes using one of the methods in sections 3.2 or 3.3.

Procedure KEYS as given below can be used to generate n, e and d and insures that for all M in (0...n-1), gcd (M,n) = 1 and $(M^e \bmod n)^d \bmod n = M$.

The following is a procedure to generate the keys using PTEST.

***PROCEDURE KEYS*:**

```
1.  n = random ; n is generated to be an odd integer
    and as large as required.

2.  F = n - 1

3.  FIND e s.t.  GCD (E,F) = 1; e can be as long as
    required.

4.  FIND d s.t. E.D MOD n-1 = 1

5.  For i : = 1 to k do

       M = random (n)

       if (gcd (M,n) ≠1) or ($M^{(e*d)} \bmod n$)≠$M$

  then reject n and goto step # 1
```

```
        endif

        endfor

 6.  Output n as the modulus, e as the enciphering key and d
     as the deciphering key.
End procedure.
```

***LEMMA 6*:**

1. If the input n is prime then Procedure KEYS will not reject n as the modulus.

***PROOF*:**

If n is prime then for all M < n gcd(M,n) = 1, and also $\varphi(n) = n-1$, and gcd (e, $\varphi(n)$) = 1, where $\varphi(n)$ is the Euler totient function of n.

In this case e * d mod $\varphi(n)$ = 1 will have a unique solution for d. Therefore $(M^n \bmod n)^d$ mod n = M for all M < n as guaranteed by the Euler Theorem.

Hence if n is prime then procedure PTEST above will not reject n. QED.

***LEMMA 7*:**

If n is composite then the algorithm may accept n as the modulus (in this case the keys will work only for blocks which are relatively prime to the modulus).

***PROOF*:**

A counter-example: Let n = 15 (a composite number). Then n-1 = 14. Let e = 13, i.e. gcd(e,F) = 1, then we can find d = 13 such that e * d (MOD F) = 1. Now by choosing any M that is relatively prime to n we will find out that: $((M^{E})\ \mathrm{mod}\ n)^{D})$ mod n) = M (Exhaustively). Therefore, Procedure PTEST will not reject n as the modulus although n is composite, QED.

***LEMMA 8*:**

The probability that the above procedure will produce wrong keys is when $Gr \neq Gl$ and this probability $is \leq \frac{1}{2^{k}}$.

***PROOF*:**

If n is prime then from Lemma 1 Gr = Gl. If n is composite and Gr = Gl as the example in Lemma 2 then those keys are guaranteed to encrypt and decrypt correctly all messages reltively prime to n.

If n is composite and $Gr \neq Gl$ then the probability that PTEST will not reject n in each trial of step 5 *is* $\leq 1/2$ so the probability that PTEST will not reject n for k trials *is*$\leq \frac{1}{2^k}$.

CHAPTER IV

OTHER PROBABILISTIC ALGORITHMS

4.1 INTRODUCTION :

In chapter three all the probabilistic algorithms for primality testing were of the Monte Carlo type. To complete the overview of the probabilistic algorithms types and examples this chapter covers probabilistic algorithms for three more problems. Two of these algorithms are of the Las Vegas type and one of the Monte Carlo type.

4.2 NEAREST PAIR ALGORITHM

The Nearest Pair problem in its general form is that given $x_1 , x_2 , \ldots . x_n$ to be n points in k_dimensional space R^k , one is asked to find a pair (x_i , x_j) such that d (x_i , x_j) is the minimum among all n pairs , where d(x,y) is a "distance measure" between point x and point y.

A brute force method, i.e. exhaustive, will give the answer by evaluating n(n-1)/2 distance computations and n(n-1)/2 - 1 comparisons.

Shamos & Benteley and later Yuval found two probabilistic algorithms requiring O(n*log n) distance computations. Both methods are recursive and involve considerable overhead in auxiliary memory, [RAB76].

The major steps in the algorithm need list processing , namely we are given n integers a_1 , a_2 , ... a_n and want to find the indices i,j for which $a_i = a_j$. If hashing is possible then this can be done in O(n). The algorithm is not recursive and easily programmable, [RAB76].

The algorithm is probabilistic in the sense that the running time is O(n) with high probability. However the output of the algorithm when it halts is definite , therefore the algorithm can be classified as of the Las Vegas type.

Here we will briefly outline how Rabin proved that the expected running time is O(n) starting with listing some necessary definitions given in [RAB76] :

1. Let D be a decomposition on S such that

$S = S_1 U S_2 U \ldots\ldots S_k$, $c(S_i)=n_i$, *let* $N(D) = \frac{\sum_{i=1}^{k} n_i(n_i-1)}{2}$ be a measure of D.

2.*If T , a subset of S ,* is a choice of m elements from S then we call T is a success on D if at least two elements were chosen from the same S_i

3.*If* D_1 is a partition S = H_1 U .... U H_l then we say that D dominates D_1 if for every m the probability of success on D with a choice m *elements* ≥ the probability of success on D_1 with a choice of m elements.

Rabin first proved that there exists $\lambda>0$ such that for every partition D of any finite set S there exists another partition D' of the same set such that $\lambda N(D) \le N(D')$ and D dominates D' and all sets (parts) in D' with exception of singletons. Then he proved that If $S_1 = \{x_{i_1} \ldots\ldots\ldots\ldots x_{i_m}\}$, $m=n^{2/3}$. is chosen at random from S and a lattice Γ with mesh size $\sigma(S_1)$ *is formed* then the probability of $(N(\Gamma) \le c_1 n) > (1-2e^{-\sqrt{2}\lambda n^{1/6}})$ *where* λ is that λ in the previous definition.

The algorithm was studied and implemented using C (Appendix B). The algorithm was tested on a different sets of random points and the time of Rabin method and the brute method was computed and compared. Table 2 on the next page gives the results for a timing experiment. It is clear that for the random cases tested the probabilistic solution is faster the deterministic solution and the difference increases as the number of points increases , which shows the difference in the order of execution time.

Number of random points	Rabin time in sec.	Brute method time in sec.
10	1.31	0.87
50	6.70	15.04
100	17.52	59.59
200	53.33	238.81

Table 2 : comparison of run time of the probabilistic algorithm & the deterministic algorithm on a sample of random points.

THE ALGORITHM :

INPUT :

S a set of n points in k dimensional space.

OUTPUT : x_i , x_j such that $d(x_i , x_j) = \min d(x_p , x_q)$ *where* $1 \le p < q \le n$. and $d(x_p , x_q)$ = the Euclidean Space distance measure.

PROCEDURE :

1. Let $m_1 = \lfloor n^{2/3} \rfloor$, $m_2 = \lfloor m_1{}^{2/3} \rfloor$ and S1 is generated from m1 random elements from S , while S2 is is generated from m2 random elements from S1.

2. Calculate $\delta(S2)$ = minimum $d(x_p , x_q)$ where x_p , x_q *are in* $S2$ (this minimum is calculated exhaustively).

3. Call Procedure Near(S1, $\delta(S2)$, m_1) and will receive back as an output $\delta(S1)$

4. Call Procedure Near(S, $\delta(S_1)$,n) and will receive back as an output $\delta(S)$ which will be the minimum sought.

Procedure Near(S, δ ,n) :

1. Construct a square lattice with mesh size = δ Every time let the origin to be in different position in the square middle , corner , middle of upper side and middle of left side. All other possiblities are symmertical to the prevoius four possiblities. (that is why we have four possiblities)

```
2. For i = 1 to 4 do
    a. Construct a square lattice $L_i$ with mesh size = $2*\delta$ and compute also $k_i$ = number of squares of $k_i$
    b. For j = 1 to $k_i$ do
        $S_{i,j}$ = $S$ intersected with square j of $L_i$ and compute
        $S_{i,j}$ = $S$ intersected with square j of $L_i$ and compute d = min $d(x_p , x_q)$ where $d(x_p , x_q)$ in $S_{i,j}$ then update minimum and p and q.
    ENDFOR
ENDFOR
3. RETURN minimum , p and q.
```

4.3 PATTERN MATCHING PROBABILISTIC ALGORITHM

The problem of pattern matching is that given a string of n bits called the pattern and a much longer string with m bits called the text . One is asked to determine if the pattern occurs as a block within the text.

A brute force solution to the problem is to compare every n consecutive bits of the text with the pattern. In the worst case the cost , i.e the execution time of this method , is of the order of the product of m and n , this method is not practical unless n is small.

Rabin and Karp [KAR86] gave a probabilistic algorithm to this problem which is of O(m+n) instead of O(m*n). The algorithm is of type Monte Carlo since it will always give an answer but it may lie some of the time.

The idea of the algorithm is that we have a number of fingerprinting functions defined on the text and on the pattern. A random fingerprinting function is chosen, call it f_i , f_i(*pattern*) and f_i(*block of n bits of the text*), are evaluated , then f_i(*pattern*) and f_i(*block of n bits of the text*), are compared. If not equal then no match otherwise there is a probable match. If the previous step is repeated k times each one resulting in a probable match then the overall probability of error = $pf_1 * pf_2 * \ldots\ldots\ldots pf_k$ where pf_i is probability of collision (two different strings with the same value of f_i) of two strings using f_i as a fingerprinting function. For a summary of the method see [KAR86].

```
THE ALGORITHM :
INPUT  : PATTERN OF LENGTH N , TEXT OF LENGTH M
OUTPUT : DECIDE IF THE PATTERN OCCURS IN THE TEXT OR
NOT
PROCEDURE :
  WHILE (NO MATCH AND NOT END OF THE STRING) DO
        1. Find next block of n bits in the text.
        2. FOR I = 1 TO K DO
             a. Choose randomly a fingerprinting function.
             b. Compute it for the pattern and for the
                block and compare them,
                if not equal exit the loop in step # 2
                and go on to the next block.
           ENDFOR
        3. A match found with probability of error equal
           to the product of the probability of error in
```

```
            each fingerprinting function.
            Exit the algorithm.
  ENDWHILE
      There is no match in the text.
END (ALGORITHM)
```

4.4 PROBABILISTIC ALGORITHMS IN FINITE FIELDS

The problem of finding an irreducible polynomial of degree n over a finite field and the problem of finding a root of a polynomial over a finite field are of great importance to algebraic coding theory , algebraic symbol manipulation and to number theory.

Solving these problems by exhaustive search is not feasible for large values of the size of the field.

Berlekamp solved the root finding problem for $f \in GF(p^n)$, deg(f) = m , by reducing it to the factorization problem of another polynomial $F(x) \in Z_p[x]$ ($Z_p = GF(p)$) , is the field of residues mod p),where deg(F) = mn. The problem of factoring $F(x) \in Z_p[x]$, is solved by reducing it to finding the roots in Z_p of another polynomial $G(x) \in Z_p[x]$. Thus all problems are reduced to that of root finding in $Z_p[x]$. For root finding in a large Z_p, a case in which search is not feasible, Berlekamp proposes a probabilistic algorithm involving a random choice of $d \in Z_p$. [BER70].

Rabin gave probabilistic algorithms for these problems , they are probabilistic in the sense that they use randomization in the search for for the solution. If they declare a solution it is definitely correct but the algorithm may not halt with very low probability , and therfore it is of the Vegas type.

4.4.1 RABIN PROBABILISTIC ALGORITHM FOR FINDING AN IRREDUCIBLE POLYNOMIAL OVER A FINITE FIELD :

INPUT : p the characteristic of the field Z_p

OUTPUT : $f(x)=a_0 + a_1 * x + a_2 * x^2 + \ldots a_m * x^m$ irreducible over Z_p.

PROCEDURE :

```
    REPEAT
            Pick a random polynomial over f(x) $\in Zp$
            , test it for irreduciblity by testing if
            $f(x) \mid (x^{p^n} - x)$
              and
            $(f(x), x^{p^{m_i}} - x) = 1$ where $1 \le i \le k$
```

```
        then stop and output f(x).
***UNTIL FOUND***
```

4.4.2 RABIN PROBABILISTIC ALGORITHM FOR FINDING A ROOT OF A POLYNOMIAL OVER A FINITE FIELD :

INPUT : $f(x) = a_0 + a_1 * x + a_2 * x^2 + \ldots . a_m * x^m$ in E(x) , GE = GF (p^n) , p and n.

OUTPUT : α *such that* $f(\alpha) = 0$ *(if it exist)*.

PROCEDURE :

```
1. Find $f_1(x) = (f(x), x^{(p-1)}-1)$.
2. If $f_1(x) = 1$ then stop and output no roots for f(x) in E.
3. i = 1
4. Choose randomly $\delta$ in E and compute $f_d(x) = (f_i(x), (x+\delta)^{(d-1)})$.
5. If $0 <$ deg $f_d$
   $<$ deg $f_i$ le (1/2)*deg $f_i$ then
   If deg fd $\le (1/2)$*deg $f_i$ then
        $f_{i+1}(x) = f_d(x)$
        else
        $f_{i+1}(x) = f_i / f_d(x)$
         endif
         i = i + 1
```

```
If deg $f_i(x)>1$ then
goto step 4
 else
output $\alpha$ such that $f_i(x) = x-\alpha$
  else
     goto step 4
 endif
```

Rabin proved that the expected number of iterations until δ found is less than 2 , and since in step 4 f_1 is at least halved then at most $\log_2$ m is nedded to find a factor $x-\alpha_i$ of f(x),i.e. a root.

The number of field operations $E=GF(p^n)$ required to find $f_1(x)$ and $f_2(x)$ is O(n.mL(m)log p) (where L(a) = $\log a \times \log\log a$). Since $deg\ f_{i+1} \leq deg\ f_i$, i.e. the number of field operations to find f_3 is at most half the number of field operations to find f_2 and similarly for f_4 etc. So the total number of field operations used for finding a root of f(x) is still O(n.mL(m)log p)

The expected number of operations in Z_p *is* $O(n^2.mL(m)L(n)\log p)$ since each operation in Z_p requires O(nL(n)) field operations with residues modulo p.

CHAPTER V

CONCLUSION AND FUTURE WORK

CONCLUSION :

In this thesis a number of probabilistic algorithms were studied and connected to various types of probabilistic algorithms. However the main part of the work was devoted for utilizing pseduprimality test (or Fermat test) in primality testing.

The estimation of the upper bound of the probability of a number being a Carmichael number showed that the pseudoprimality test will improve as n grows larger. Three practical applications resulted which are :

The first application is utilizing the pseudoprimality test alone as a primality test is a feasible proposition as supported by the upperbound on the probability for a composite number passing the test and as the impirical tests showed.

The Second is using the pseudoprimality test combined with Rabin test to produce a Monte Carlo algorithm to test wheather a given integer is a Carmichael number or not. The results proved to be correct with a high probability $(1- 0.5^{k})$ and with low cost $(4+l)*\log_2 n$

The third is key generation for the Pohlig-Hellman encryption scheme and again the probability that the keys are correct is as high as $(1-0.5^{k})$ and the cost is $6*\log_2 n$

The other algorithms studied (nearest pair , pattern matching and problems in finite fields) served as a completion of the total view of probabilistic algorithms and explained the various types of the probabilistic algorithms.

The probabilistic algorithms for primality testing for Rabin , Solovay & Strassen and PTEST were implemented in C. The implementation resulted in a library of 15 operations for long integers (including addition , multiplication & modular multiplication , fast exponentiation .. etc). The library forms a basis for applications requiring long integer arithmetic such as the R.S.A algorithm for encryption & decryption.

FUTURE WORK :

This thesis covered the theortical foundations of probabilistic algorithms , surveyed the classififcation of probabilistic languages, and investigated the utilization of the pseudoprimality test in a probabilistic algorithm for primality testing.

The investigation of the pseudoprimality test resulted in deriving an upperbound for the probability of composite integers passing that test, this probability was rferred to as pd.

Possible future work along the lines of analysis for pd include the the following points:

1. Derivation of a tighter upperbound for pd by giving a more refined analysis of the probabilities of the three events involved as mentioned in section 3.4. The analysis will involve conditional probabilities of interdependent events , and will also involve deriving a tight bound for $\lambda(n)$. So far there is no lower bound for $\lambda(n)$, however there is an upper bound for it , $\lambda(n) \leq n/2$.

2. Utilizing the relation derived between $\lambda(n)$ and pd to produce bounds on C(x) (C(x) is the counting function of the Carmichael numbers less than x see [POM81]), and then

comparing these bounds with the ones given in the literature , e.g. see [POM81].

The general area of probabilistic algorithms for primality testing is very active and certainly there is a need for new algorithms that are faster and more reliable unless a polynomial time deterministic algorithm is found.

Finally the successful probabilistic algorithms in the literature today are problems that are either in P or in NPI. It remains to be seen if there are probabilistic algorithms for Np-Complete or Np-hard problems

REFERENCES:

1. [*ADL*83] Aderman, Pomerance and Rumely, "On Distinguishing Prime Numbers from Composite Numbers", Annals of Mathematics, Vol. 117, 1983, pp. 173-206

2. [*BEA*88] Beauchemin , Pierre and Brassard , G. "The Generation of Random Numbers That are Probably Prime", Journal of Cryptology Vol. 1 1988 , pp. 53-64.

3. [*CAR*15] Carmicheal R.D., "The Theory of Numbers", Dover Publications, 1959.

4. [*COH*87] Cohen and Lenstra, Implementation of a new Primality Test, Mathematics of Computation, Vol. No. 177, 1987, pp. 103-121.

5. [*DEN*82] Denning, D.E.R., "Cryptography and Data Security", Addison-Wesley, Reading, Massachussets, 1982.

6. [*ERD*86] ***Erdos*** and Pomerance, "On the Number of False Witness for a Composite Number", Mathematicians of Computation, Vol. 46, No. 173, January 1986, pp. 259-279.

7. [*GAR*79] Garey , M. & Johnson , D. , Computers and Intractibility , W.H. Freeman Press , 1979.

8. [*GIL*77] Gill, J.,"Compuational Complexity of Probabilistic Turing Machines", SIAM J. Comput. , 6(1977) , pp. 675-695. pp. 259-279.

9. [*HAR*79] Hardy , G. & Wright , E. , An Introduction to The Theory of Numbers , Clarendon Press , Oxford , 1979.

10. [*KAR*86] Karp , R. , "Compinatorics Complexity , And Randomness" , Comm. Assoc. Mach. 29(1986) , pp. 98-109.

11. [*MON*80] Monier , L. , "Evaluation and Comparison of Two Efficent Probabilistic Primality Testing Algorithms" , Theortical Computer Science , Vol. 11 , pp. 97-108 , 1980.

12. [*KNU*81] Knuth, D., "The Art of Computer Programming, Seminumerical Algorithms, Vol. 2, Addison-Wesley, Reading, Mass., 1981.

13. [*POH*78] Pohlig, S. and Helmman, M.,"An Improved Algorithm for Computing Logarithms over GF(p) and its Cryptographic Significance", IEE Trans. on INformation Theory, Vol. IT-24(1), Jan. 1978, pp. 106-110.

14. [*POM*81] Pomerance, C., "On the Distribution of Pseudoprimes", Math. Comp., Vol. 37, 1981, pp. 587-593.

15. [*RAB*76] Rabin, M.O., "Probabilistic Algorithms in Algorithms and Complexity - Recent Results and New Directions" (J.F. Traub, Ed.), Academic Press, New York, 1976, pp. 21-40.

16.*a* [*RAB*80] Rabin, M.O., "Probabilistic Algorithms for Primality Testing", J. of Number Theory, No. 12, 1980, pp. 128-138.

16.*b* [*RAB*80] Rabin, M.O., "Probabilistic Algorithms in Finite Fields", SIAM J. COMPUT. Vol. 9 , No. 2 , May 1980 , pp. 273-280.

17. [*ROS*84] Rosen, Kenneth H., "Elementary Number Theory and Its Applications", Addison-Wesley, Reading Massachussets, 1984.

18. [*SCH*85] ***Schöning*** , U., Complexity and Structure , Lecture Notes in Computer Science , Vol. 211 , Springer-Verlag , Berlin , 1986.

19. [*SOL*77] Solovay, R. and Strassen, V., "A Fast Monte Carlo Method for Primality", SIAM Journal for Computing, Vol. 6, 1977.

20. [*WAG*86] Stan Wagon , The Mathmatical Intelligence , Vol.8 , No.3 , 1986. Primality", SIAM Journal for

Computing, Vol. 6, 1977.

21. [*ZAC*82] Zachos , S. , "Robustness of Probabilistic Computational Complexity Classes under Defintional Perturbations", Inf. and Cont. 54(1982) , pp. 143-154. Primality", SIAM Journal for Computing, Vol. 6, 1977.

APPENDIX A

THE FOLLOWING IS A LIST OF ALL THE ROUTINES USED IN IMPLEMETING the probabilistic primality testing programs

Note :

All operands in theses procedures are represented by a pointer to integer (in this case the operand will all the consecuctive words pointed by the pointer for a specific length)

Some abberivations :

UI	= Unsigned Integer
PUI	= Pointer to Unsigned Integer
PC	= Pointer to Character
PI	= Pointer to integer
PF	= Pointer to float
gcd	= Greatest Common divisor
LOP	= Length of pointer contents
CPB(x)	= Contents Pointed to By x
Procedure Name	= PN
Purpose	= PU
Return Value	= RV
Number of parameters	= NP

PN	PU	RV	NP
random()	Generate a pseduo random number	UI between 0 and FFFF	0
inc(x,length)	Increments x by one on a LOP(x) = length	UI = modified LOP(x)	2
clear_data(x, lower,upper)	Set CPB(x+lower) to CPB(x+upper-1) to zero	-	3
out(x,name, ength)	Display on the screen the contents of LOP(x) = length with a message in name	-	3
notequal(x,y ,length)	Check if CPB(x) equal or not to CPB(y) of LOP = length	UI = 1 to mean not equal or 0 to mean equal	3
copy(x,y, length)	Copy CPB(x) to CPM(y) of LOP = length	-	3
jacobi(x,y, length)	Compute jacobi symbol of	J(x,y) = -+ 1	3

gcd(u,v,counter)	Copmute gcd of CPB(u) , CPB(v) of LOP = counter	-	3
rem(x,y,counter)	Compute the quotient and the remainder of dividing top(x) by CPB(y)	-	3
getbit(l,b, x)	Return the lth bit of CPB(x) in a word having it as its bth bit	UI = word of all zeros exept the bth bit = lth bit of CPB(x)	3
count(x,counter)	Return number of bits of CPB(x) of maximun LOP = counter	UI = # of bits of CPB(x)	2
l_words(x, counter)	Return number of words of CPB(x) of maximun LOP = counter	UI = # of words of CPB(x)	2
subtract(a, d,l)	Subtract CPB(d) form CPB(a) of LOP = l	-	3

greater(x,y, 1)	Check if CPB(x) > CPB(Y) or < or =	UI = 0 if x < y 1 if x > y 2 if x = y	3
shiftl(w,1, append)	Shift CPB(w) to the left one bit and bit 0 = append	-	3
notone(x,1)	Check if CPB(x) = 1 or not	UI = 1 if x <> 1 0 if x = 1	2
fastexp(m,e, t,counter1, counter2, blocksize)	Compute $CPB(m)^{CPB(e)}$ mod CPB(t) where LOP(m) = blocksize , LOP(e) = counter1 , LOP(t) = counter2	-	6
mult(n,x,t, counter2)	Compute CPB(n) * CPB(x) mod CPB(t) on LOP = counter2	-	4
notzero(x,1)	Return weather CPB(x) = 0 or not	UI = 1 if x <> 0 0 if x = 0	2

add(a,d,1)	Compute CPB(a) + CPB(d) on LOP = 1	-	3
multil(q,s, cc)	Compute CPB(q) * CPB(s) on LOP = cc	-	3
shifr(w,cc)	Shift one bit to left CPB(w) on lOP = cc	-	2
shiftr(w,cc)	Shift one bit to rigth CPB(w) on LOP = cc	-	2
inv(ai,ni, res,cc).	Compute res such that ai * res mod ni = 1 on LOP = cc	-	4

THE FOLLOWING IS A LIST OF ALL THE PARAMETERS FOR ALL PROCEDURES

NOTE :

------

1. In the coulmn Old Value destroyed is applied only for contents of PUI type
2. Warning 1 means that the current length of PUI type may increase by 1
3. Warning 2 means that the current length of PUI type may be doubled
4. ? means garbage.

Procedure Name	Parameter Name	Parameter Type	Old value Destroyed (contents)	New Value if destroyed
inc	x	PUI	Y	CPB(x) incremented Warning 1
=	length	UI	-	-
clear_data	x	PUI	Y	CPB(x+i) = 0 for i = lower to upper
=	lower	UI	-	-
=	upper	=	-	-
out	x	PUI	N	-
=	name	PC	-	-
=	length	UI	-	-

notequal	x	PUI	N	-
=	y	=	N	-
=	length	UI	-	-
copy	x	PUI	N	-
=	y	PUI	Y	CPB(y) = CPB(x)
=	length	UI	-	-
jacobi	x	PUI	Y	?
=	y	=	Y	?
jacobi	length	UI	-	-
gcd	u	PUI	Y	CPB(u)=gcd(CPB(u),CPB(v))
=	v	=	Y	?
=	cc	UI	-	-

rem	x	PUI	Y	CPB(x) = CPB(x)/CPB(y)
=	y	=	Y	CPB(y) = rem(CPB(x)/CPB(y)
=	counter	UI	-	-
getbit	1	=	-	-
=	b	=	-	-
=	x	PUI	N	-
count	x	PUI	N	-
=	counter	UI	-	-
1_words	x	PUI	N	-
=	counter	UI	-	-
subtract	a	PUI	Y	CPB(a) = CPB(a) - CPB(d)
=	d	=	N	-
=	1	UI	-	-

greater	x	PUI	N	-
=	y	=	N	-
=	1	UI	-	-
shiftl	w	PUI	Y	CPB(w) is shifted one bit to the left bit 0 = append warning 1
=	1	UI	-	-
=	append	UI	-	-
notone	x	PUI	N	-
=	1	UI	-	-

fastexp	m	PUI	Y	CPB(m) = $CPB(m)^{CPB(e)}$ mod CPB(t)
=	e	=	N	-
=	t	=	N	-
=	counter1	UI	-	-
=	counter2	=	-	-
=	blocksize	=	-	-
mult	n	PUI	Y	CPB(n) = CPB(n)*CPB(x) mod CPB(t)
=	x	=	N	-
=	t	=	N	-
=	counter2	UI	-	-
notzero	x	PUI	N	-
=	1	UI	-	-

add	a	PUI	Y	CPB(a) = CPB(a)+CPB(d)
=	d	=	N	-
=	l	UI	-	-
multil	q	PUI	Y	CPB(q) = CPB(q)*CPB(s) warning 2
=	s	=	N	-
=	cc	UI	-	-
shifr	w	PUI	Y	CPB(w) is shifted one bit to left warning 1
=	cc	UI	-	-
shiftr	w	PUI	Y	CPB(w) is shifted one bit to right
=	cc	UI	-	-

inv	ai	PUI	N	-
=	ni	=	N	-
=	res	=	Y	CPB(res) = inv(CPB(ai),CPB(ni))
=	cc	UI	-	-

LISTING

```
void fastexp(unsigned int *m,unsigned int *e,unsigned int *t,unsigned
             counter1,unsigned int counter2,unsigned int blocksize);
void mult(unsigned int *n,unsigned int *x
          ,unsigned int *t,unsigned int counter2);
unsigned int  far notzero(unsigned int *x,unsigned int l);
void add(unsigned int *a,unsigned int *d,unsigned int l);
/*---------------------------------------------------------*/
/* this routine receives m e t and computes m**e mod t */
/* and return the result in m                            */
/* ---------------------------------------------------------  */
/* The algorithm works in the folowing sense    :
    start q =1.
    check if e is odd or even then halve n
    if n is even q = q*q mod t
    else  q = q*q mod t.
          and q = q*m mod t.                                */
/* ---------------------------------------------------------  */
void fastexp(unsigned int *m,unsigned int *e,unsigned int *t,unsigned
             counter1,unsigned int counter2,unsigned int blocksize)
{
   unsigned int c1,c2,d1;
   unsigned int q(100);
   unsigned int i,j,k;
   clear_data(q,0,100);
   *(q)  =0x01;
   for (i=0;i<counter1;++i)
     {
       d1 = *(e+counter1-i-1);
        for (j=0;j<16;++j)
         {
          mult(q,q,t,2*counter2);
           c1 = d1;
           c1 >>= (15-j);
           c1 &=0x01;
           if (c1 == 1)
             {
               mult(q,m,t,counter2+blocksize);
             }
         }
     }
  for (i=0;i < counter2 ; ++i)
    *(m+i) = *(q+i);
}
/* ------------------------------------------------------  */
void mult(unsigned int *n,unsigned int *x,unsigned int *t
         ,unsigned int counter2)
{
    unsigned int n1(100),y(100),z(100),c,flag,save(100);
    unsigned int j;
/* A1. (intialize)  n1 = n , Y=0 , z = x */
    clear_data(y,0,100);
    copy(t,save,100);
    copy(x,z,100);
    copy(n,n1,100);
    flag = 0x01;
/* ----------------------------------------------------------  */
/* A2. (Halve n1.)    */
  while (flag == 1)
   {
        c = n1(0);
        shiftr(n1,counter2);
        c &=0x0001;
/* ----------------------------------------------------------  */
/* A3. add y to z    */
         if (c == 1)
             {
                add(y,z,counter2+1);
                rem(y,t,counter2+1);
                for (j=0;j<2*counter2;++j)
                  t(j) = save(j);
/* ----------------------------------------------------------  */
/* A4. If n1=0 terminate    */
                if (notzero(n1,counter2) == 0) flag = 0;
            }
/* -------------------------------------------------------  */
/* A5. z = (z + z) mod t     */
         add(z,z,counter2+1);
         rem(z,t,counter2+1);
         for (j=0;j<2*counter2;++j)
           t(j) = save(j);
    }
/* -------------------------------------------------------  */
    for (j=0;j<counter2;++j)
    *(n+j) = y(j);
}
/* -------------------------------------------------------  */
/* this routine receives x (pointer to unsigned int) and decide if
/* x = 0 or not                                             */
/* -------------------------------------------------------  */
unsigned int far notzero(unsigned int *x,unsigned int l)
 {
    int j;
    for (j=0;j < l;++j)
     if (*(x+j) != 0 ) return(1);
    return(0);
```

```
}
/* ------------------------------------- */
/*       long digit addition             */
/*       THE RESULT is storde in a       */
/* ------------------------------------- */
/* This algorithm works in the following sense */
/* take a , d byte by byte add them if the result < any of them */
/* Then carry = 1 else carry = 0                                */
/* ------------------------------------- */
void add(unsigned int *a,unsigned int *d,unsigned int l)
{
  unsigned int result(100),carry;
  int i;
  carry = 0x00;
  for (i=0;i<l+1;++i)
    {
      result(i) = *(a+i) + carry;
      if (result(i) < *(a+i))
          carry  = 0x01;
     else carry  = 0x00;
      result(i) = result(i) + *(d+i);
      if (carry == 0)
            if (result(i) < *(d+i))
                 carry  = 0x01;
    }
  for (i=0;i<2*l;++i)
    *(a+i) = result(i);
}
/* ------------------------------------ */
void rem(unsigned int *x,unsigned int *y,unsigned int counter);
unsigned int far getbit(unsigned int l,unsigned int b,
          unsigned int *x);
unsigned int far count(unsigned int *x,unsigned int counter);
unsigned int far l_words(unsigned int *x,unsigned int counter);
void subtract(unsigned int *a,unsigned int *d,unsigned int l);
unsigned int far greater(unsigned int *x,unsigned int *y,
           unsigned int l);
void shiftl(unsigned int *w,unsigned int l,unsigned int append);
unsigned int far notone(unsigned int *x,unsigned int l);
/* ------------------------------------------------------- */
void rem(unsigned int *x,unsigned int *y,unsigned int counter)
{
   unsigned int i,flag,l1,l2,j,sub(100),q(100),l3,l4;
   unsigned int save(100),test,last,byte,bit,res,append;
   if (notzero(x,counter) == 0)
    return;
   clear_data(sub,0,100);
   clear_data(q,0,100);
  l1=count(x,counter);
  l2=count(y,counter);
  if (l2 > l1)
   {
     for(i=0;i<counter;++i)
      y(i) = 0;
     return;
   }
  if ((l2+1) % 16 == 0)
    l3=(l2+1)/16;
  else
    l3=(l2+1)/16+1;
  l4   = l1-l2;
  byte = 0;
/* ----------------------------------------  */
/* the next loop for form the intial 12 bits  */
/* ----------------------------------------  */
  for (i=0;i < l2-1;++i)
   {
     byte = i/16;
     bit  = i % 16;
     res = getbit(i+l4+1,bit,x);
     sub(byte)  = res;
   }
```

```
    for (i=0;i<=14;++i)
     {
       res = getbit(14-i,0,x);
       shiftl(sub,12+1,res);
       if (greater(sub,y,13) != 0)
         {
           subtract(sub,y,13);
           append = 0x0001;
         }
       else
           append = 0x0000;
       shiftl(q,i+1,append);
     }
     for (i=0;i<counter;++i)
     {
       *(x+i) = sub(i);
       *(y+i) = q(i);
     }
}
/* ---------------------------------------------------------------*/
/* this procedure will find the lth bit of x and reformat it to
   the bth bit of one output word called res
-----------------------------------------------------------------*/
unsigned int far getbit(unsigned int l,unsigned int b
                             ,unsigned int *x)
{
  unsigned int byte,bit,res;
   byte = l/16;
   bit  = l % 16;
   res  = x(byte);
   res  >>= bit;
   res  &=0x0001;
   res  <<= b;
   return(res);
}
/* -----------------------------------------------------------*/
unsigned int far count(unsigned int *x,unsigned int counter)
{
   unsigned int flag,l1,test,last,j;
   l1=counter;
   flag =1;
   while (flag == 1)
    {
      if (*(x+l1-1) == 0) --l1;
      else flag =0;
    }
   test = *(x+l1-1);
   l1  *= 16;
   flag =1;
   while (flag != 0)
    {
      last = test;
      last >>= (16-flag);
      last &=0x0001;
      if (last == 0)
       {
         --l1;
         ++flag;
       }
      else
         flag =0;
    }
   return(l1);
}
/* ---------------------------------------------------------  */
/* this routine receives x (pointer to int) and decide if    */
/* x = 0 or not                                               */
/* ---------------------------------------------------------  */
void subtract(unsigned int *a,unsigned int *d,unsigned int l)
 {
  unsigned int result(100),borrow;
  unsigned int i;
 borrow = 0x00;
 for (i=0;i<l;++i)
   {
      result(i) = *(a+i) - borrow;
      if (result(i) > *(a+i))
          borrow  = 0x01;
     else borrow  = 0x00;
     *(a+i) = result(i);
      result(i) = result(i) - *(d+i);
      if (borrow == 0)
            if (result(i) > *(a+i))
                   borrow  = 0x01;
   }
  for (i=0;i<l;++i)
    *(a+i) = result(i);
}
/* ------------------------------------- */
unsigned int far greater(unsigned int *x,unsigned int *y,
      unsigned int l)
{
  unsigned int i;
  unsigned int x1,y1;
  for (i=0;i<l;++i)
   {
    x1 = *(x+l-i-1);
    y1 = *(y+l-i-1);
    if (x1 > y1) {
                        return(1);
                 }
    else if (x1 < y1) {
                         return(0);
                        }
  }
  return(2);
}
/* -----------------------------------------------------------  */
/* this routine shifts w to the right one bit                    */
/* -----------------------------------------------------------  */
void shiftl(unsigned int *w,unsigned int l,unsigned int append)
{
  unsigned int c,e;
  unsigned int i,j;
  if ((l%16) == 0)
   j = l/16;
  else
   j = (l/16) + 1;
  c = append;
  for (i=0;i<j;++i)
```

```
      {
        e = *(w+i);
        e &= 0x8000;
        *(w+i) <<= 1;
        *(w+i)  = c;
        c = e;
        c >>=15;
      }
}
/*---------------------------------------------------------------*/
unsigned int far notone(unsigned int *x,unsigned int l)
 {
    int j;
    if (*(x) != 1) return(1);
    for (j=1;j < l;++j)
     if (*(x+j) != 0 ) return(1);
    return(0);
 }
 /* ---------------------------------------------------------- */
unsigned int far l_words(unsigned int *x,unsigned int counter)
  {
   unsigned int l;
   l = count(x,counter);
   if (l % 16 == 0)
```

```
    return(l/16);
   else
    return(l/16+1);
 }
/* ------------------------------------------------------------ */

/* -----------------------------------------------------------*/
/* Last date of modification started in  march 5 1988         */
/* Subject of this procedures : implementing jacobi  algorithm */
/* -----------------------------------------------------------*/
void multil(unsigned int *q,unsigned int *s,unsigned int cc);
void shifr(unsigned int *w,unsigned int cc);
void shiftr(unsigned int *w,unsigned int cc);
void inv(unsigned int *ai,unsigned int *ni,unsigned int *res,
         unsigned int cc);
/* ----------------------------------------------------------
   type ggg defines an array of maximum of 100 elements each is
   a pounsigned inter to unsigned intacters (bytes)
   ---------------------------------------------------------  */
/*-------------------------------------------------*/
void inv(unsigned int *n2s,unsigned int *nls,unsigned int *b,
         unsigned int cc)
 {
  unsigned int bs(100),q(100),r(100),t2(100),save(100),nl(100)
```

```
             ,n2(100);
int b_sign,bs_sign,t2_sign;
clear_data(b,0,100);
clear_data(bs,0,100);
clear_data(t2,0,100);
clear_data(save,0,100);
copy(n1s,n1,100);
copy(n2s,n2,100);
b(0) = 0x0001;
b_sign = 1;
bs_sign = 1;
t2_sign = 1;
while (1)
 {
   copy(n1,r,100);
   copy(n2,q,100);
   rem(r,q,cc);
   if (notzero(r,cc) == 0)
    break;
  else
    {
      copy(b,t2,cc);
      t2_sign= b_sign;
      multi1(q,b,cc);
 if (b_sign != bs_sign)
  {
   b_sign = bs_sign;
   add(q,bs,cc);
   copy(q,b,cc);
 }
else
 {
   if (greater(bs,q,cc) == 1)
     {
       b_sign = bs_sign;
       subtract(bs,q,cc);
       copy(bs,b,cc);
     }
   else
     {
       b_sign *=-1;
       subtract(q,bs,cc);
       copy(q,b,cc);
      }
 }
   copy(t2,bs,cc);
   bs_sign = t2_sign;
   copy(n2,n1,cc);
```

```
          copy(r,n2,cc);
       }
   }
   copy(n1s,n1,cc);
   if (b_sign == -1)
   {
    subtract(n1,b,cc);
    copy(n1,b,cc);
   }
}
/* ----------------------------------------------------*/
/*       long digit multiiplication        */
/*       THE RESULT is stored in q        */
/* ------------------------------------- */
/* The algorithm works by repeted rotation and addition   */
/* ------------------------------------- */
void multi1(unsigned int *q,unsigned int *s,unsigned int cc)
{
  unsigned int w(100),w1(100),res;
  unsigned int c(100),e,c1,e1,c2,e2;
  unsigned int i,j,k;
  for (i=0;i < cc;++i)
       c(i) = *(q+i);
    for (i=0 ;i < cc ; ++i)
```

```
      *(w+i) =  *(s+i);
    for (i=cc;i < 2*cc ; ++i)
     {
       *(w+i) &=0x00;
       *(q+i) &=0x00;
     }
    for (i=0;i < 2*cc ; ++i)
       *(w1+i) &=0x00;
  for (i=0;i < cc;++i)
     {
       for (j=0;j<16;++j)
         {
           e = c(i);
           if ((e & 0x01) == 1) add(w1,w,cc);
           c(i) >>= 1;
           shifr(w,cc);
         }
     }
    for (i=0 ;i < 2*cc ;++i)
      *(q+i) = *(w1+i);
}
/* ------------------------------------- */
void shifr(unsigned int *w,unsigned int cc)
{
  unsigned int c,e,i;
  c = 0x00;
  for (i=0;i<2*cc;++i)
      {
        e = *(w+i);
        e = e & 0x8000;
        *(w+i) <<= 1;
        *(w+i)  = c;
        c = e;
        c >>=15;
      }
}
/* ------------------------------------*/
void shiftr(unsigned int *w,unsigned int cc)
{
   unsigned int c,e,i;
   c = 0x0000;
   for (i=0;i<cc;++i)
       {
         e = *(w+cc-i-1);
         e &=0x0001;
         *(w+cc-i-1) >>= 1;
         *(w+cc-i-1)  = c;
         c = e;
        c <<=15;
      }
}
/* ------------------------------------------ */
unsigned int far random();
unsigned int far inc(unsigned int *x,unsigned int l);
void clear_data(unsigned int *x,unsigned int lower,
                unsigned int upper);
void out(unsigned int *x,char *name,unsigned int length);
unsigned int far notequal(unsigned int *x,unsigned int *y,
                          unsigned int length);
void copy(unsigned int *a,unsigned int *b,unsigned int counter);
int far jacobi(unsigned int *a,unsigned int *b,unsigned int counter);
void far gcd(unsigned int *ag,unsigned int *ng,unsigned int cc);
#define multiplier  25173
#define modulus     65536
#define increment   13849
#define intial_seed 17
unsigned int far random()
{
  static unsigned int seed = intial_seed;
  seed = (multiplier * seed + increment) % modulus;
  return(seed);
}
```

```
/* --------------------------------------------- */
unsigned int far inc(unsigned int *x,unsigned int l)
{
  unsigned int i;
  i=0;
  ++x(i);
  while (x(i) == 0)
   {
    ++i;
    ++x(i);
   }
  if (i > l-1)
   return(l+1);
  else
   return(l);
}
/* ------------------------------------------------------ */
void clear_data(unsigned int *x,unsigned int lower,
                unsigned int upper)
{
 unsigned int i;
 for(i=lower;i<upper;++i)
   *(x+i) = 0;
}
```

```
/*-----------------------------------------------*/
void out(unsigned int *x,char *name,unsigned int length)
{
 unsigned int i;
 printf("%s",name);
 for(i=0;i<length;++i)
  printf("%x ",x(length-i-1));
 printf("\n");
}
/*---------------------------------------------------*/
unsigned int far notequal(unsigned int *x,unsigned int *y,
                          unsigned int length)
{
  unsigned int i;
  for(i=0;i<length;++i)
   {
     if (x(i) != y(i))
       return(1);
   }
  return(0);
}
/*---------------------------------------------------*/
/* source      : a
   destination : b
```

```
-----------------------------------------------------*/
void copy(unsigned int *a,unsigned int *b,unsigned int counter)
{
 unsigned int i;
 for(i=0;i<counter;++i)
   *(b+i) = *(a+i);
}
/*---------------------------------------------------*/
int far jacobi(unsigned int *a,unsigned int *b
           ,unsigned int counter)
{
 int j;
 unsigned int one(100),parm(100),counter1,counter2;
 one(0) = 1;
 clear_data(one,1,100);
 j = 1;
 counter1 = l_words(a,counter);
 counter2 = l_words(b,counter);
 if (counter1 > counter2)
   counter = counter1;
 else
   counter = counter2;
 while (notone(a,counter) == 1)
  {
```

```
clear_data(parm,0,100);
counter1 = l_words(a,counter);
counter2 = l_words(b,counter);
if (counter1 > counter2)
  counter = counter1;
else
  counter = counter2;
if ((a(0) & 1) == 0)
 {
  shiftr(a,counter);
  parm(0) = b(0);
  parm(0) = parm(0)*parm(0);
  --parm(0);
  if ((parm(0) & 8) != 0)
   j*= -1;
 }
else
 {
   copy(a,parm,counter);
   --parm(0);
   --b(0);
   parm(0) = b(0)*parm(0);
   if ((parm(0) & 4) != 0)
     j*=-1;
```

```
      ++b(0);
      copy(a,parm,counter);
      rem(b,a,counter);
      copy(b,a,counter);
      copy(parm,b,counter);
    }
  }
 return(j);
}
/*------------------------------------------------------------*/
void  far gcd(unsigned int *u,unsigned int *v,unsigned int cc)
{
 unsigned int i,j,save(100);
 unsigned int save1(100),save2(100);
 clear_data(u,cc,100);
 clear_data(v,cc,100);
 copy(v,save2,cc);
 while (notzero(v,cc) == 1)
  {
    copy(v,save,cc);
    rem(u,v,10);
    copy(u,v,cc);
    copy(save,u,cc);
  }
```

```
   copy(save2,v,cc);
}
```

APPENDIX B

THE FOLLOWING IS A LIST OF ALL THE ROUTINES USED IN IMPLEMETING THE NEARSET PAIR PROBABILISTIC ALGORITHM BY RABIN

Some abberivations :

```
Procedure Name        = PN
Purpose               = PU
Return Value          = RV
Number of parameters = NP
dist type = struct
             {
               float x,y;
               int index;
             }
 dist_int type = struct
                 {
                  int x,y;
                  int index;
                 }
```

PN	PR	RV	NP
near_pair(ss ,s,p,q,sigma ,counter)	Calculate the minimum of an arrary of points using Rabin algorith	-	6
minimum(s,p, q,min, ,counter)	Calculate the minimum of an arrary of points exahaustively	-	5
compute(s, s_li,x_shift, y_shift,si, counter)	Calculate the points in each square	-	6
sort(s_li, flag,lower, upper)	Sort an array of points using linear sort	-	4

THE FOLLOWING IS A LIST OF ALL THE PARAMETERS FOR ALL PROCEDURES OF THE NEARES PAIR PROBABILISTIC ALGORITHM

NOTE :

------

1. In the coulmn Old Value destroyed is applied for only contents
   of PUI type
2. ? means garbage.

Procedure Name	Parameter Name	Parameter Type	Old value Destroyed (contents)	New Value (if destroyed)
near_pair	ss	array of dist	N	-
=	s	=	N	-
=	p	PI	Y	Index of the first point in the pair

=	q	PI	Y	Index of the second point in the pair
=	sigma	PF	=	Pointer to minimum
=	counter	INTEGER	-	-
minimum	s	array of dist	N	-
=	p	PI	Y	Index of the first point. in the pair
=	q	PI	Y	Index of the second point in the pair
=	min	PF	=	Pointer to minimum
=	counter	INTEGER	-	-

compute	s	array of dist	N	-
=	s_li	array of dist int	Y	Each s_li(i).x = (s(i).x+min* x_shift)/(2*min) Each s_li(i).y = (s(i).y+min* y_shift)/(2*min)
=	x_shift	PF	N	-
=	y_shift	PF	N	-
=	si	PF	N	-
=	counter	INTEGER	-	-
sort	s_li	array of dist_int	Y	The elements are sorted according to one field which is denoted by flag
=	flag	INTEGER	-	-

=	lower	=	-	-
=	upper	=	-	-

LISTING

```
#include <stdio.h>
#include <math.h>
#include <dos.h>
#define multiplier  17
#define modulus     300
#define increment   61
#define intial_seed 7
typedef struct
 {
   float x,y;
   int   index;
 } dist;
typedef struct
 {
   int x,y;
   int   index;
 } dist_int;
void near_pair(dist *ss,dist *s,int *p,int *q,float *sigma,
int counter);
void minimum(dist *s,int *p,int *q,float *min,int counter);
int  random();
void compute(dist *s,dist_int *s_li,float x_shift,float y_shift,
float sigma, int counter);
```

```
void sort(dist_int *s_li,int flag,int lower,int upper);
int random()
{
  static int seed = intial_seed;
  seed = (multiplier * seed + increment) % modulus;
  return(seed);
}
/* -------------------------------------------- */
main()
{
 dist s(200),s1(200),s2(200);
 char in_name(25);
 int counter,i,p,q,m1,m2,j;
 float min;
 FILE *in_file,*fopen();
 struct time now;
 printf ("\n Enter name of file that contains the data : \n");
 scanf ("%24s",in_name);
 in_file = fopen (in_name,"r");
 if (in_file==NULL)
   printf ("could not open %s for reading.\n:",in_name);
 else
  {
   fscanf(in_file,"%d",&counter);
  for (i=0;i<counter;++i)
   {
    fscanf(in_file,"%f",&s(i).x);
    fscanf(in_file,"%f",&s(i).y);
    s(i).index = i;
   }
  m1 = pow(counter,0.6666);
  m2 = pow(counter,0.4444);
  for (i=0;i<m1;++i)
    s1(i) = s(i);
  for (i=0;i<m2;++i)
    s2(i) = s1(i);
gettime(&now);
printf("\n Rabin algorithm started at %02d:%02d:%02d:%02d\n",
           now.ti_hour,now.ti_min,now.ti_sec,now.ti_hund
  minimum(s2,&p,&q,&min,m2);
  near_pair(s,s1,&p,&q,&min,m1);
  near_pair(s,s,&p,&q,&min,counter);
  printf("\n rabin minimum = %f at %d , %d \n",min,p,q);
  gettime(&now);
printf("\n Rabin algorithm ended at %02d:%02d:%02d:%02d\n",
           now.ti_hour,now.ti_min,now.ti_sec,now.ti_hund
  minimum(s,&p,&q,&min,counter);
  printf("\n minimum brute force = %f at %d , %d \n",min,p,q);
```

```
   gettime(&now);
   printf("\n Brute force method ended at %02d:%02d:%02d:%02d\n",
              now.ti_hour,now.ti_min,now.ti_sec,now.ti_hund);
  }
}
/*---------------------------------------------------*/
void minimum(dist *s,int *p,int *q,float *min,int counter)
{
 float value;
 int i,j;
 *p = s(0).index;
 *q = s(counter-1).index;
 j = counter-1;
 *min=sqrt(pow(s(0).x-s(j).x,2.0)+pow(s(0).y-s(j).y,2.0));
 for (i=0;i<counter-1;++i)
   {
    for (j=i+1;j<counter;++j)
     {
      value=sqrt(pow(s(i).x-s(j).x,2.0)+pow(s(i).y-s(j).y,2.0));
      if (value < *min)
       {
        *min = value;
        *p = s(i).index;
        *q = s(j).index;
        }
      }
    }
  }
/* ---------------------------------------------------*/
void near_pair(dist *ss,dist *s,int *p,int *q,float *sigma,
               int counter)
{
  int i,lower,j,p1,q1,k,xx,yy;
  dist_int s_li(200);
  dist s_i_j(200);
  float min;
for (xx=0;xx<2;++xx)
{
for (yy=0;yy<2;++yy)
 {
  compute(s,s_li,xx,yy,*sigma,counter);
  sort(s_li,1,0,counter-1);
  i = 0;
  while (i < counter)
   {
    lower = i;
    while (s_li(i).x == s_li(i+1).x)
        ++i;
    if (lower == i)
        ++i;
    else
        sort(s_li,0,lower,i);
   }
  i = 0;
  while (i < counter)
   {
    lower = i;
    while ((i<counter)&&((s_li(i).x==s_li(i+1).x)
                         &&(s_li(i).y==s_li(i+1).y)))
     {
       k = s_li(i).index;
       s_i_j(i-lower) = ss(k);
       k = s_li(i+1).index;
       s_i_j(i-lower+1) = ss(k);
       ++i;
     }
    if (lower == i)
        ++i;
    else
      {
        minimum(s_i_j,&p1,&q1,&min,i-lower+1);
        if (min  < *sigma)
```

```
          {
            *sigma = min;
            *p = p1;
            *q = q1;
          }
      }
   }
 }
 }
}
/* ------------------------------------------------------*/
void compute(dist *s,dist_int *s_li,float x_shift,float y_shift,
             float sigma,int counter)
{
 int i;
 for (i=0;i<counter;++i)
   {
     s_li(i).x = (s(i).x + sigma*x_shift)/(2.*sigma);
     s_li(i).y = (s(i).y + sigma*y_shift)/(2.*sigma);
     s_li(i).index = s(i).index;
   }
}
/*-------------------------------------------------------*/
void sort(dist_int *s_li,int flag ,int lower,int upper)
```

```
{
 int i,j,s1,s2;
 dist_int temp;
 for (i=lower;i<upper;++i)
  {
    for (j=lower;j<upper;++j)
     {
     if  (flag == 1)
      {
       s1 = s_li(j).x;
       s2 = s_li(j+1).x;
      }
     else
      {
       s1 = s_li(j).y;
       s2 = s_li(j+1).y;
      }
     if (s1 > s2)
        {
         temp = s_li(j);
         s_li(j) = s_li(j+1);
         s_li(j+1) = temp;
        }
  }
```

```
    }
  }
```