Uluslararası İleri Doğa Bilimleri ve Mühendislik Araştırmaları Dergisi
Sayı 8, S. 182-195, 2, 2024
© Telif hakkı IJANSER'e aittir
Araştırma Makalesi

https://as-proceeding.com/index.php/ijanser
ISSN: 2980-0811

International Journal of Advanced Natural Sciences and Engineering Researches
Volume 8, pp. 182-195, 2, 2024
Copyright © 2024 IJANSER
Research Article

A review of primarily tests and algorithms: Engaging students to code for mathematics

Robert Kosova*[1], Fatjona Bushi[2], Rinela Kapçiu[3], Fabiana Cullhaj[4], Anna Maria Kosova[5]

[1] Department of Mathematics, University "A. Moisiu" Durrës. Albania, e-mail robertkosova@uamd.edu.al*
[2] Department of Computer Science. University "A. Moisiu" Durrës. Albania, email fatjonabushi@gmail.com
[3] Department of Computer Science. University "A. Moisiu" Durrës. Albania, rinelakapciu@uamd.edu.al
[4] Department of Mathematics, University "A. Moisiu" Durrës. Albania, e-mail, fabianacullhaj@uamd.edu.al
[2] Department of Computer Science. University "A. Moisiu" Durrës. Albania, email annamariakosova@studnetsuamd.edu.al

(Received: 29 February 2024, Accepted: 08 March 2024)

(4th International Conference on Innovative Academic Studies ICIAS 2024, March 12-13, 2024)

ATIF/REFERENCE: Kosova, R., Bushi, F., Kapçiu, R. Cullhaj, F. & Kosova, A. M. (2024). A review of primarily tests and algorithms: Engaging students to code for mathematics. *International Journal of Advanced Natural Sciences and Engineering Researches, 8(2), 182-195.*

Abstract – The concept of prime numbers has intrigued mathematicians for centuries. The attempt to understand prime numbers dates back to ancient times, with great mathematicians like Euclid, who, among many other topics, explored integers and prime numbers' properties. Determining whether a number is prime or composite lies at the heart of many mathematical problems, leading to the development of primality tests. Since the first primality algorithm, the Sieve of Eratosthenes, the need to verify very large prime numbers has driven the development of many efficient tests and algorithms. This article presents an overview of some of the most important primality algorithms, as well as corresponding Python programs developed by computer science and mathematics students. The cooperation between theoretical mathematics and programming has become the premise for progress in the development of tests and helps students gain a better understanding of the theoretical problems while also encouraging and guiding them in their future research and new discoveries.

Keywords – Prime, Primality, Algorithm, Test, AKS, Ferma- Euler, Divisor, Euclid, Sieve, Eratosthenes.

I. INTRODUCTION

The development of algorithms and computer programming in the 1960s with the creation of the first programming languages has contributed a great deal to the study of many mathematical problems. Many math conjectures were verified for very large numbers which was impossible before, properties and features of integers were studied, patterns, formulas, and relations were discovered more easily, which helped to understand concepts and problems that required a lot of time with the old tools [1].

Some of the unsolved problems of mathematics, such as the Collatz conjecture, the twin prime conjecture, and the Mersenne primes, have been easily tested for very large numbers and remained conjectures yet; some others have been proved wrong and rejected by simple counterexamples found by computer programs; the most famous among them is the Euler conjecture on sums of like powers [2].

Among many problems in mathematics and number theory, the primality test of a natural number is an important mathematical and algorithmic problem. In addition to being a fundamental mathematical question, the problem of how to determine whether a given number is prime has tremendous practical importance [3]. Whenever someone uses the RSA public-key cryptosystem, they must generate a private key consisting of two large prime numbers and a public key consisting of their product [4].

The most commonly used primality tests are classified essentially into two types, which are deterministic and probabilistic [5]. Deterministic methods provide absolute certainty about whether a number is prime or not. Some of the most popular tests are the Lukas-Lehmer test, the Trial Division test, the Elliptic Curve Primality Test, the AKS Primality Test, etc. The probabilistic tests include the Fermat primality test, the Miller-Rabin test, the Euler test, the Solovay-Strassen test, the Frobenius primality test, etc.

Probabilistic methods can find a potential prime number, meaning the number is highly probable to be prime. In the mathematics programs of Albanian universities, the properties of integers, including prime numbers, are included in topics of number theory or discrete mathematics [6].

Many important theorems are presented and proved, such as the infinity of prime numbers with Euclid's proof, divisors' properties, the fundamental theorem of arithmetic, integer properties such as perfect numbers, perfect squares, and Pythagorean triples, modular arithmetic, etc., as well as important theorems related to prime numbers such as Willson's theorem, Ferma's or Mersenne numbers, etc. [7].

What is missing in such important topics are the corresponding algorithms and the developed programs to verify these properties, conjectures, and tests. These important topics are studied only as part of theoretical mathematics, with many theorems and proofs, while the related algorithms are missing.

A brilliant example of the cooperation of theoretical math and algorithms is the AKS primarily test [8]. Beyond that, it is unnecessary to emphasize the benefits of combining computer programming with theoretical mathematics, especially number theory, for math students [9].

Integrating coding into math topics fosters students' creative thinking and problem-solving skills. When students are tasked with coding solutions to primary tests, they are not merely memorizing formulas or algorithms; rather, they are actively engaging with mathematical concepts and applying logical reasoning to solve problems [10].

Starting in high school, the learning of several programming languages, such as C++, Java, and Java Script, helps them to develop programs, see the results of their work immediately and be encouraged by their achievements. This makes their performance in high school and university easier [11].

Coding empowers students to explore alternative approaches to problem-solving and encourages them to think outside the box. As students experiment with different algorithms and coding techniques, they develop a deeper understanding of mathematical concepts and gain insights into the various mathematical principles [12].

Additionally, integrating coding into math topics and problems helps bridge the gap between theoretical knowledge and practical application. By coding solutions to test questions, students gain a hands-on understanding of how mathematical concepts can be applied in real-world scenarios. This experiential learning approach not only reinforces their understanding of mathematical principles but also equips them with transferable skills that are highly sought after in today's digital age, such as computational thinking, data analysis, and algorithmic reasoning [13].

Furthermore, the process of writing codes promotes collaboration and teamwork among students. Coding projects often require students to work together, share ideas, and troubleshoot problems collaboratively. Through peer-to-peer interaction and discussion, students not only reinforce their own learning but also benefit from diverse perspectives and approaches to problem-solving, fostering a sense of community and collective achievement [14].

II. MATERIALS AND METHODS

Determining whether a large number is prime can be computationally intensive, especially as the number grows larger. For that reason, many algorithms are created to determine which number is prime in case if deterministic tests or probably prime in case of probabilistic test. Some of the most methods to test whether a given integer is prime are:

Deterministic tests:

The Sieve of Eratosthenes is named after the Greek mathematician and astronomer who developed this algorithm around 200 BCE. It is the oldest algorithm for finding all prime numbers up to a given number. The Sieve of Eratosthenes works by iteratively marking the multiples of each prime starting from 2 and gradually moving to larger primes. The unmarked numbers remaining after this process are prime numbers [15].

The Algorithm:

1. Create a list of consecutive integers from 2 through the given number n.
2. Start with the first unmarked number (which is 2) and mark all of its multiples up to the number n.
3. Find the next unmarked number; this is the next prime. Mark all of its multiples up to the given number n.
4. Repeat step 3 until you have done all numbers up to the square root of the given number n.
5. The remaining unmarked numbers less or equal to n are prime.

The trial division method is the first and one of the fundamental techniques to determine the primality of a number. Dating back to ancient times, this method remains a basic yet crucial tool in number theory and cryptography, allowing for the identification of prime numbers through a systematic approach of division and analysis. The algorithm for the trial division method can be outlined as follows:

1. Given a number n, check if it's less than 2. If so, it's not prime.
2. Iterate through all integers d from 2 to the square root of n.
3. For each d, check if it divides n evenly.
4. If a divisor is found, the number is declared composite.
5. If no divisors are found up to the square root of n, the number is declared prime.

Despite its simplicity, the trial division method faces significant inefficiency when dealing with extremely large numbers. As the candidate number grows, the number of divisions required increases exponentially, making it impractical for testing large primes, especially in cryptographic applications where such numbers are essential for security [16].

The Willson's Theorem test:

Willson is a deterministic method to check if a positive integer is prime or composite.

The result was known to Leibniz, but it was only published in 1770 by Waring, who named it after his former student John Wilson who had discovered it. The theorem was proved in 1771 by French mathematician Joseph-Louis Lagrange. Wilson's theorem is both necessary and sufficient condition for primality [17].

The Wilson's theorem states that any prime p divides $(p-1)! + 1$.

Theorem. An integer $p \geq 2$ is prime if and only if $(p-1)! \equiv -1(mod\ p)$.

Proof 1. If p is prime then p divides $(p-1)! + 1$.

Let p a prime number. Each of the integers $1,2,3,\dots p-1$ has an inverse modulo p.

Consider the first and last reminder of $(modp)$, which are 1 and $(p-1)$ we have:

$1.(p-1) \equiv (p-1)(mod\, p) \equiv -1(modp)$.

We can partition the set $S = \{2, \dots, p-2\}$ into pairs $\{a, b\}$ such that $a.b \equiv 1(modp)$.

Then the product of these pairs is $1.2.3\dots(p-1) \equiv 1.(-1)(modp) \equiv -1(modp)$.

For example, for $p = 7$, we have $(7-1)! \equiv 1.2.3.4.5.6 \equiv (2.4).(3.5).(1.6) \equiv 1.1.(-1)(mod7) \equiv -1(mod7)$

Proof 2. If p divides $(p+1)! + 1$ then p is prime.

Suppose that p is composite. Then p has a factor $1 < d \leq p-1$.

Then d divides $(p-1)!$, so d does not divide $(p-1)! + 1$.

Therefore p does not divide $(p-1)! + 1$.

For a composite number it is true that:

A positive integer n is composite number if and only if $(n-1)! \equiv 0(modn)$, except for $n = 4$ for which we have $(4-1)! \equiv 3! \equiv 2(mod4)$.

Wilson's theorem provides an effective deterministic way to check if a given natural number is prime or composite. However, while this method is effective for small integers it is limited for large integers as it involves computing very large factorials, so it is thus hard to compute them for large $n!$ even using fast computers.

The AKS Primality Test is a deterministic algorithm used to determine if a given number is prime or composite. It was developed by three Indian computer scientists—Manindra Agrawal, Neeraj Kayal, and Nitin Saxena—in 2002.

The test's significance lies in its ability to determine primality in polynomial time, challenging the common belief that primality testing required exponential time [18- 19].

The algorithm runs in O((log n)^12) time, making it theoretically efficient for large numbers. However, in practice, it's slower than probabilistic tests like the Miller-Rabin test for moderately large numbers due to its high constant factors.

Theorem. Let have an integer a, and a positive integer$n$; $a \in Z, n \in N, n \geq 2$ and $gcd(a, n) = 1$. Then n is prime if and only if $(X + a)^n = X^n + a(modn)$.

Proof. For $0 < i < n$, the coefficient of x^i is

$((X+a)^n - (X^n + a))$ is $\binom{n}{i} a^{n-i}$.

Suppose n is prime. Then is $\binom{n}{i} \equiv 0(mod\, n)$ and hence all the coefficients are zero.

Suppose n is composite. Consider a prime q that is a factor of n and let $q^k // n$. Then q^k does not divide $\binom{n}{q}$ and is coprime to a^{n-q} and hence the coefficient of $X^q$is not zero $(mod\, n)$.

Thus $((X+a)^n - (X^n + a))$ is not identically zero over Z_n.

The Algorithm:

```
1. Input integer $n > 1$.
2. If $n = a^b$ for $a \in N$ and $b > 1$, output composite.
3. Find the smallest $r$ such $o_r(n) > log^2 n$.
4. If $1 < (a, n) < n$ for some $a < r$, output composite.
5. For $a = 1$ to $\lfloor \sqrt{\emptyset(r)} \log n \rfloor$ do
       If $(X + a)^n \neq (X^n + a)$, output composite
6. Output prime;
```

Probability tests:

The Fermat primality test, named after French mathematician Pierre de Fermat, is a probabilistic algorithm which is used to evaluate whether a given number is likely prime or absolutely composite.

Fermat's test offers a fast approach but does not guarantee accuracy for all numbers [20]. The Fermat test relies on Fermat's Little Theorem, which states:

Theorem. If p is a prime number and a is any positive integer then $a^p \equiv a(modp)$.

In case $gcd(a, p) = (a, p) = 1$ we have:

Theorem. If p is a prime number and a is any positive integer, $gcd(a, p) = 1$, then we have $a^{p-1} \equiv 1(modp)$.

Proof. Let $S = \{1,2,3, \dots, p - 1\}$.

Let's consider the set $a.S = \{1a, 2a, \dots, (p - 1)a\}$, consisting of the product of the elements of S with a. This set is simply a permutation of S (taken modulo p). In other words, $S = \{1a, 2a, \dots, (p - 1)a\}\ (modp)$. Clearly none of the ia factor for $1 \leq i \leq p - 1$ are divisible by p, so it suffices to show that all the elements in $a.S$ are distinct.

Then the product $1a.2a \dots.(p - 1)a \equiv 1.2 \dots (p - 1)(modp)$.

Cancelling the factors $1.2.3 \dots (p - 1)$ from both sides, we have $a^{p-1} \equiv 1(mod\ p)$.

The Algorithm:

1. For a given positive integer p,
2. Pick an integer a that $2 \leq a \leq p - 2$, and $(a, p) = 1$, and check if it holds or not.
3. If it doesn't hold, we know that p is composite.

In this case we call the base a *Fermat witness* for the compositeness of p.

4. If it holds, we say that p is probably prime.

However, there are composite numbers that may incorrectly pass the Fermat primality test for some values of a. These are called Fermat pseudoprimes, and in such case the base a is called *Fermat liar*.

For $n = 561$, $a = 2$, we have the Fermat formula $2^{560} \equiv 1(mod\ 561)$, but $n = 561$ is a composite, number, divisible by 3,11,17.

For $n = 91, a = 3$ we have $3^{90} \equiv 1(mod91)$, but 91 is also composite, $91 = 13 * 7$.

The reliability of the Fermat test for any integer n can be increased by performing multiple iterations with different random bases a. The more iterations are conducted, the higher the confidence in the primality evaluation.

Euler primality test is an improvement over the Fermat primality test because it adds another equality condition that a prime number must fulfill [21-22].

Theorem. If p is prime and a is an integer, where $p > 2$ and $0 < a < p$, then $a^{\frac{p-1}{2}} \equiv \pm 1\ (mod\ p)$.

Theorem. If n is a positive integer and $a^{\frac{n-1}{2}} \not\equiv \pm 1(mod\ n)$ where $gcd(n, a) = 1$ then n is composite.

Proof. If $a^{\frac{n-1}{2}} \not\equiv \pm 1(mod\ n) \rightarrow a^{n-1} \not\equiv 1(mod\ n)$ then n is composite because of the Fermat's Little Theorem.

The Algorithm:

1. For a given positive integer p,
2. We pick an integer a that $2 \leq a \leq p - 2$, and $(a, p) = 1$, and check if the Fermat formula $a^{\frac{p-1}{2}} \equiv \pm 1(modp)$ holds or not.
3. If it doesn't hold, we know that p is composite.
4. If it holds, we say that p is probably prime.

However, there are composite numbers that may incorrectly pass the Euler primality test for some values of a.

For $n = 341$ and $a = 2$ we have $2^{\frac{341-1}{2}} \equiv 2^{170} \equiv 1(mod341)$, but $341 = 11 * 31$.

Every Euler pseudoprime is also a Fermat pseudoprime.

The Miller-Rabin primality:

Miller-Rabin Primality Test is an improvement over Fermat's test, Miller-Rabin is also probabilistic but more robust. It repeatedly applies a test based on modular exponentiation to determine if a number is composite or probably prime [23].

The Miller-Rabin test relies on the properties of modular arithmetic and the behaviour of certain mathematical functions to efficiently test the primality of a number. It's based on the fact that for prime numbers, certain properties hold true, and if these properties are violated, the number is certainly composite.

The Algorithm:

1. Select an odd number n to test for primality and a base a such that $(1 < a < n)$.
2. Express $(n-1) = 2^s * d$ where s is the largest power of 2 that divides $(n-1)$ and d is an odd number.
3. Compute $a^d (mod\ n)$ and check if the result is congruent to $-1(mod\ n)\ or\ 1(mod\ n)$.
4. If $a^d \not\equiv \pm 1(mod\ n)$ and also, we have $a^{2^r \cdot d} \not\equiv -1(mod\ n), 0 <= r < s-1$ then n is composite and a is a witness for the compositeness.
5. Otherwise, n may be prime.
6. If it does, the number is likely prime.
7. If it becomes (1) after fewer than (s) iterations or if it never reaches $(-1)(mod\ (n))$, then (n) is composite**.**

While the Miller-Rabin test can also yield false positives (declaring a composite number as prime), the chance of error decreases with more iterations. Also, by choosing multiple random bases a and performing the test with each, the probability of misidentifying a composite number as prime diminishes significantly.

However, because of the probabilistic nature it does not guarantee the primality of a number, but rather provides a high probability. Rarely, composite numbers can pass the test for some witnesses.

The effectiveness of the test relies on the choice of witnesses (base a). Although random selection helps, certain sets of witnesses are more effective than others [24].

The Miller-Rabin primality test is a powerful and widely used probabilistic algorithm for evaluating the primality of large numbers efficiently. While not offering absolute certainty, it provides a high degree of confidence, making it an invaluable tool in various fields, especially in cryptography where large prime numbers play a crucial role in ensuring security.

III. RESULTS

The primality tests in Python language are developed by the Computer Science and Mathematics students (University "A. Moisiu Durres. Faculty of IT, Albania).

The sieve of Eratosthenes: For a given integer n, the following program finds all the multipliers of all the numbers from 2 to square root of n and delete them. The remaining numbers are primes.

The program is:

```
# The Sieve of Eratosthenes
import math
import datetime
n=int(input("enter max number, n= "))
print("primes number are:")
n1=int(n.5)
for p in range (2,n+1):
    for d in range(2,n1):
        if p>d and p%d==0:
            p=0
    if p>0:
```

```
    print(p,end=",")
```

```
RUN
enter max number, n= 100
prime number are:
2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,
```

The trial division test:

The following Python program verifies all the integers between two given numbers. All the integers that have not divisors between 2 and n-1 are prime.

```
#Python programming
print("The trial division test")
print("Algorithm: count divisors of n, from 2 to sqr(n)")
print("If number of divisors, nd=0 then n is prime")
print("If not, nd>0 then n is composite")
import math
import datetime
n1=int(input("enter first number,  (n1>1)="))
n2=int(input("enter second number,(n2>n1)="))
k=0
print("nr              n            result")
print("------------------------- ------")
if n1==1:
   n1=2
for n in range (n1,n2+1):
   np=0
   for d in range (2,int(n(.5))+1):
      if n%d==0:
         np=np+1
         d = d + 1
   if np==0:
      k=k+1
      print(k,"-","\t",n,"  prime")
   else:
      k=k+1
      print(k,"-","\t",n,"  composite")
```

```
RUN
enter first number, (n1>1) =125654
enter second number, (n2>n1) =125663
nr          n           result
----------------------------------
1 -        125654   composite
2 -        125655   composite
3 -        125656   composite
4 -        125657   composite
5 -        125658   composite
```

```
6 -      125659   prime
7 -      125660   composite
8 -      125661   composite
9 -      125662   composite
10 -     125663   composite
```

For large numbers, small changes are made:
n1=2^n1+1, n2=2^n2+1.
For n1=2^53+1, n2=2^53+2, the integers between are:

```
nr              n                result
------------------------------------------------
1 -      9007199254740993   composite
2 -      9007199254740994   composite
3 -      9007199254740995   composite
4 -      9007199254740996   composite
5 -      9007199254740997   prime
6 -      9007199254740998   composite
7 -      9007199254740999   composite
8 -      9007199254741000   composite
9 -      9007199254741001   composite
10 -     9007199254741002   composite
11 -     9007199254741003   composite
12 -     9007199254741004   composite
13 -     9007199254741005   composite
14 -     9007199254741006   composite
15 -     9007199254741007   composite
16 -     9007199254741008   composite
17 -     9007199254741009   composite
18 -     9007199254741010   composite
19 -     9007199254741011   composite
20 -     9007199254741012   composite
```

Fermat’s primality test:
For any positive odd integer $n,$ we generate all the bases $a, 1 < a < n - 1$ to calculate $a^{n-1}(mod n).$ For odd composite integers, the test produces correct results for most of the values of base $a.$
However, there are many odd composite integers, for which the test produces false positive result, declaring them probably prime.
The following Python program verifies odd integers n with all the values of base a, less than n and coprime with $n.$

```
print("Python programming")
print("Fermat's Primality test")
print("Theorem. If p is prime number and a an integer, and 1<a<p-1,(a,p)=1 then a(p-1)=1(mod p)")
print("Fermat primality test: if a(n-1)#1(mod n) then n is composite, else n is probably prime")
import datetime
import math; import cmath; import calendar
n=int(input("enter integer n to verify if prime or not, n>2; "))
print("generate base a, a<n-1, (a,n)=1")
print()
print("a    F=a^(n-1)     r=F(mod n)     n=",n,"is")
print("-----------------------------------------------")
for a in range (2,n-1):
    if math.gcd(n,a)==1:
        F=pow(a,(n-1)); r=F%n
        if r!=1:
            print(a,"    \t",F,"\t       ",r,"    \t", "composite")
        else:
            print(a,"    \t",F,"\t       ",r,"    \t", "probably prime")
print()
```

```
RUN
a   F= a^(n-1)        r= F(mod n)      number 11 is
-------------------------------------------------------
2 - 1024               1            probably prime
3 - 59049              1            probably prime
4 - 1048576            1            probably prime
5 - 9765625            1            probably prime
6 - 60466176           1            probably prime
7 - 282475249         1            probably prime
8 – 1073741824      1            probably prime
9 - 3486784401      1            probably prime
```

For odd composite integers:
For $n = 15$ (composite), and base $a = 4;\ a = 11$
the result is Fermat pseudoprime.

```
a   F= a^(n-1)          r= F(mod n)      number 15 is
---------------------------------------------------------
2 - 16384                      4          composite
4 - 268435456                  1          probably prime
7 – 678223072849               4          composite
8 - 4398046511104              4          composite
11 - 379749833583241     1          probably prime
13 – 3937376385699289  4          composite
```

For $n = 25;\ a = 7$, we have another Fermat pseudoprime, because number 25 is composite and $F = a^{n-1} = 7^{24} \equiv 1(mod25)$.
For $n = 33;\ a = 10,\ a = 23$ we have another Fermat pseudoprime, $n = 33$ is composite.
$F = 10^{32} \equiv 1(mod33);\ F = 23^{32} \equiv 1(mod33)$
For $n = 35;\ a = 6, a = 29$ the same result.
$F = 6^{34} \equiv 1(mod35);\ F = 29^{34} \equiv 1(mod35)$.
Other Fermat pseudoprimes are:
For $n = 39, a = 25;$
For $n = 45;\ a = 17,19,26,28,37.$
For $n = 55;\ a = 21,34,$

Euler primality test:

For any positive odd integer n, we generate all the bases $a, 1 < a < n - 1$ to calculate $a^{\frac{n-1}{2}}(mod n)$. For odd composite integers, the test produces correct results for most of the values of base a. However, there are many odd composite integers, for which the test produces false positive result by declaring them probably prime. All Euler pseudoprimes are Fermat pseudoprimes.
The following program in Python verifies if any integer is composite or probably prime. The values of base a are less than the given number n and coprime with n.

```
print("Python programming")
print("Primality test: Euler")
print("Theorem. If p is prime and a an integer,1<a<p-1,(a,p)=1 then a(p-1)/2=+-1(mod p)")
print("Euler primality test: if a(n-1)#+-1(mod n) then n is composite, else n is probably prime")
import datetime
import math; import cmath; import calendar
n=int(input("enter integer n to verify if prime or not, n>2; "))
print("generate base a, a<n-1, (a,n)=1")
print()
print("a=     F=a^[(n-1)/2]      r=F%n       n=",n,"is")
print("-------------------------------------------------")
for a in range (2,n-1):
   if math.gcd(n,a)==1:
      F=pow(a,(n-1)/2); F=int(F); r=int(F%n)
      if r==(n-1):
         r=-1
      if r==1 or r== -1:
         print(a,"    \t",F,"\t       ",r,"    \t", "prime")
      else:
         print(a,"    \t",F,"\t       ",r,"    \t", "composite")
print()
```

```
RUN
a=     F=a^[(n-1)/2]      r=F%n       n= 17 is
---------------------------------------------------
2         256                 1             prime
3         6561               -1             prime
4         65536               1             prime
5         390625             -1             prime
6         1679616            -1             prime
7         5764801            -1             prime
8         16777216            1             prime
9         43046721            1             prime
10        100000000          -1             prime
11        214358881          -1             prime
12        429981696          -1             prime
13        815730721           1             prime
14        1475789056         -1             prime
15        2562890625          1             prime
```

The AKS test:

For any given natural number n.
Calculate the coefficients of $(x + 1)^n$.
$(x - 1)^n - (x^n - 1)$.

İf the remaining coefficients are all multiplies of n then n is prime, else is composite.
The program in Python is:

```
print("The AKS test")
print("Theorem: Number p is prime iff coefficients of (x-1)**p-(x**p-1)")
print("are all multiplies of p")
print()
import math
import datetime
p=int(input("Enter first number, p= "))
c=math.factorial(p)
c1 = [1 for i in range(p+1)]
c2 = [1 for i in range(p+1)]
k = [1 for i in range(p+1)]
sc=0
for i in range (0,p+1):
    c1[i]=(math.factorial(i))
    c2[i]=(math.factorial(p-i))
    #print (c,"-",c1[i],"-", c2[i])
    k[i]=c//c1[i]
    k[i]=k[i]//c2[i]
    k[i]=k[i]*(-1)**(i)
    #print(k[i])
    i=i+1
print("Coefficients of (x-1)^",p,"are:")
for i in range(0,p+1):
    print(k[i],end=", ")
    i=i+1
print()
print("Coefficients of (x-1)^",p,"-(x^",p,"-1) are:")
for i in range(1,p):
    print(k[i],end=", ")
    i=i+1
print()
for i in range(1,p):
    if (k[i]%p)==0:
        sc=sc+1
        i=i+1
print()
print("Result:")
if sc==p-1:
    print("Coefficients are all multiplies of", p, )
    print("Number",p," is prime")
else:
    print("Some coefficients are not multiples of",p)
    print("Number",p," is composite")
print()
```

```
RUN
Enter first number, p= 13
Coefficients of (x-1)^ 13 are:
1, -13, 78, -286, 715, -1287, 1716, -1716, 1287, -715, 286, -78, 13, -1,
Coefficients of (x-1)^ 13 -(x^ 13 -1) are:
-13, 78, -286, 715, -1287, 1716, -1716, 1287, -715, 286, -78, 13,

Result:
Coefficients are all multiplies of 13
Number 13  is prime

RUN
Enter first number, p= 14
Coefficients of (x-1)^ 14 are:
1, -14, 91, -364, 1001, -2002, 3003, -3432, 3003, -2002, 1001, -364, 91, -14, 1,
Coefficients of (x-1)^ 14 -(x^ 14 -1) are:
-14, 91, -364, 1001, -2002, 3003, -3432, 3003, -2002, 1001, -364, 91, -14,

Result:
Some coefficients are not multiples of 14
Number 14 is composite
```

IV. DISCUSSION

The tests and algorithms presented in the paper are coded in Python by students of computer science and mathematics. Prime numbers, together with many theorems and proofs related to them, as well as some of the primality tests such as Ferma's and Wilson's, are covered in the courses of number theory and discrete mathematics. However, the fact that the students also study several programming languages, such as Java, C++, and Python, in their respective courses gave us the idea to develop programs for some of the well-known tests and algorithms and to apply for a project at our university. The benefit to the students of a project that combines mathematics with algorithms and programming is indisputable and offers an excellent perspective for students as mathematicians and programmers of the future. It deeply enhances understandings of theoretical mathematics concepts, deepens the algorithmic and programming culture, fosters critical thinking, fosters collaboration, enhances communication skills, ignites curiosity, builds problem-solving skills, and encourages persistence and innovation.

V. CONCLUSION

Primality testing is not simply a mathematical curiosity; it has wide practical applications. Primality testing finds applications in many fields such as computer science, number theory, and even scientific research involving pattern recognition, pseudo-random number generation, and optimization problems.

Cryptography relies heavily on prime numbers for secure encryption and decryption processes. Algorithms such as RSA encryption, which forms the backbone of secure online transactions and communication, depend on the use of large prime numbers.

Primality tests are proof of the excellent collaboration and interaction between theoretical mathematics and practical applications. From ancient times to today, the quest for efficient primality algorithms has driven mathematical innovation.

As technology advances, so has the advancement and sophistication of priority tests, making them both more accurate and more effective.

The recognition, understanding, and application of primality is a core topic in number theory and computer science. For students of Mathematics and Computer Science, working with primality algorithms have a significant benefit and importance because:

Primality testing deepens students' understanding of number theory. It introduces them to fundamental concepts like divisors, factors, and prime numbers, fostering a strong mathematical foundation.

Implementing primality tests challenges students to think algorithmically.

Primality testing problems provide an excellent opportunity for students to enhance their problem-solving skills. They learn to approach complex mathematical problems methodically and develop strategies for efficient solutions.
Implementing primality tests requires coding skills. Students get hands-on experience with programming languages, data structures, and algorithms. This practical application helps them strengthen their coding abilities.
Primality testing involves logical reasoning and critical thinking. Students must understand the properties of prime numbers and develop logical arguments to design effective algorithms.
For students interested in pursuing advanced studies in computer science or related fields, primality testing serves as a stepping stone to more complex algorithms and mathematical concepts.
Primality testing and coding for such tests are valuable for students as they foster mathematical understanding, algorithmic thinking, coding skills, and preparation for real-world applications and challenges in various domains.
A great example of collaborating theoretical mathematics and algorithms is the AKS (Agrawal-Kayal-Saxena) primality test which is a deterministic algorithm developed by three Indian mathematicians, Manindra Agrawal, Neeraj Kayal, and Nitin Saxena, in 2002. The AKS primality test is significant because it provided the first polynomial-time algorithm for determining whether a given number is prime or composite, without relying on unproven assumptions like the Riemann hypothesis. The development of the AKS test was a significant achievement in computational number theory and algorithm design. It demonstrated that deep mathematical insights could lead to practical computational advances, and it opened new avenues for research in prime number theory and algorithmic complexity.

VI. ACKNOWLEDGMENT

The article was made possible by the sponsorship of University “A. Moisiu” Durres, Albania, through the project “Developing and creating programs and applications (Windows and Android) for mathematics and informatics problems during the teaching process to enrich and educate a creative programming culture.”

REFERENCES

[1] McGregor-Dorsey, Z. S. (1999). Methods of primality testing. *MIT Undergraduate Journal of Mathematics*, *1*, 133-141.
[2] Lander, L. J., & Parkin, T. R. (1966). Counterexample to Euler’s conjecture on sums of like powers. *Bull. Amer. Math. Soc*, *72*(6), 1079.
[3] Duta, C. L., Gheorghe, L., & Tapus, N. (2015, May). Framework for evaluation and comparison of primality testing algorithms. In *2015 20th International Conference on Control Systems and Computer Science* (pp. 483-490). IEEE.
[4] Kuang, R., & Barbeau, M. (2021, September). Indistinguishability and non-deterministic encryption of the quantum safe multivariate polynomial public key cryptographic system. In *2021 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)* (pp. 1-5). IEEE.
[5] AbuDaqa, A., Abu-Hassan, A., & Imam, M. (2020). Taxonomy and Practical Evaluation of Primality Testing Algorithms. *arXiv preprint arXiv:2006.08444*.
[6] Zaka, O. (2021). A description of some facts and open problems in Discrete-Geometry, related to Coverings. In *Mathematical methods in economy: Research and Practice, Conference, Paris, France* (Vol. 25).
[7] Zaka, O. (2022). Computing efficiently the weighted greatest common divisor. *arXiv preprint arXiv:2210.07961*.
[8] Wehrwein, J. (2022). *Primality testing* (Doctoral dissertation, Middlebury).
[9] Kosova, R., Kapçiu, R., Hajrulla, S., & Kosova, A. M. (2023). A Review of Mathematical Conjectures: Exploring Engaging Topics for University Mathematics Students. *International Journal of Advanced Natural Sciences and Engineering Researches (IJANSER)*, *7*(11), 180–186. https://doi.org/10.59287/as-ijanser.581
[10] Kosova, R., Thanasi, T., Mukli, L., & Pëllumbi, L. N. (2016). Traditional mathematics and new methods of teaching through programming together with students.
[11] Kosova, A. G. R. The Performance of University Students and High School Factors. Statistical Analyses And ANCOVA.
[12] Kosova, R., Kapçiu, R., Hajrulla, S., & Kosova, A. M. (2023). The Collatz Conjecture: Bridging Mathematics and Computational Exploration with Python. *International Journal of Advanced Natural Sciences and Engineering Researches (IJANSER)*, *7*(11), 328–334. https://doi.org/10.59287/as-ijanser.637
[13] Gjana, A., & Kosova, R. Traditional Class, and Online Class Teaching. Comparing the Students Performance Using ANCOVA. *Journal of Multidisciplinary Engineering Science and Technology (JMEST)*, 14806-14811.
[14] Hajrulla, S., Demir, T., Bezati, L., & Kosova, R. (2023). The impact of constructive learning applied to the teaching of numerical methods. *CONSTRUCTIVE MATHEMATICS: FOUNDATION AND PRACTICE*, 39.

[15] Diab, A. (2021). Development of sieve of Eratosthenes and sieve of Sundaram's proof. *arXiv preprint arXiv:2102.06653*.
[16] Ganti, I. (2022). Comparing and Reviewing Modern Primality Tests. *Journal of Student Research*, *11*(3).
[17] Valluri, M. R. (2021). Combinatorial primality test. *ACM Communications in Computer Algebra*, *54*(4), 129-133.
[18] Agrawal, M., Kayal, N., & Saxena, N. (2004). PRIMES is in P. *Annals of mathematics*, 781-793.
[19] Tao, T. (2009). The AKS primality test. *Blog post by Terence Tao*.
[20] Gradini, E. (2012). Comparison Study of Fermat, Solovay-Strassen and Miller-Rabin Primality Test Using Mathematica 6.0. *Visipena*, *3*(1), 1-10.
[21] Wang, A. (2023). Gauss-Euler Primality Test. *arXiv preprint arXiv:2311.07048*.
[22] Stüwe, D., & Eberl, M. (2019). Probabilistic primality testing. *Archive of Formal Proofs*.
[23] Burkhardt, J., Damgård, I., Frederiksen, T. K., Ghosh, S., & Orlandi, C. (2023, November). Improved Distributed RSA Key Generation Using the Miller-Rabin Test. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (pp. 2501-2515).
[24] Ishmukhametov, S. T., Rubtsova, R. G., & Khusnutdinov, R. R. (2022). A new primality test for natural integers. *Russian Mathematics*, *66*(2), 70-73.