A proposal for computationally efficient prime number generation for one-time secret password.

Kush Jayank Pandya, Siddharth Swarup Rautaray, Manjusha Pandey

School of Computer Science
Kalinga Institute of Industrial Technology
Bhubaneshwar, Odisha
1705701@kiit.ac.in

Abstract. The prime numbers play a very important role in the generation of public and private key in cryptography transaction. Every online/card payment, every password one enters, it uses the prime number to encrypt it. Prime number and its factorization have own Importance in Cryptography. This paper presents an analysis with a unique defined method using arithmetic progression for the generation of prime numbers with a range of the variable. This gives a computationally efficient way to generate a prime number.

1 Introduction

Online banking and digital transaction are the core of today's economy. All the passwords, emails, secured chats, and online transaction relies on cryptography. Cryptocurrency or BlockChain have given new dimensions to methods of encryption and Cryptography.

Methodology and efficient generation of prime numbers are extremely important in encryption and decryption of files, password, etc. which help us from protecting from online fraud and hacking.

This paper tries to present a unique way to identify whether a number is prime or not in a computationally efficient way. We use arithmetic progression and some basic math to reduce the redundancy while using a brute force method. We will try to increase our entropy(Later define for this paper's context) as much as possible in this method. This can be used to generate a one-time secret password on a very large scale.

We will also check the method's characteristic with time and space complexity with its entropy analysis.

2 State of Art

Arnab Kanti Tarafder , in his research work titled - A Comparative Analysis of General, Sieve-of-Eratosthenes and Rabin-Miller Approach for Prime Number Generation. Showed how the different methods of finding prime numbers performed.

Vladislav S. Igumnov in the year 2004 titled - Generation of the Large Random Prime Numbers Showed new approach to find the prime numbers.

Soonwook Hwang showed in his paper titled - Load-Balancedarallel Prime Number Generator with Sieve of Eratosthenes on Cluster Computer showed the complexity of prime numbers and how much time it took to compute it.

3 Methodology

3.1 How to Detect Prime No

Prime numbers are any natural numbers which have two factors one and the number itself. There are many methods to compute prime numbers we are going to discuss one of them today. Here is the C code to perform a simple operation.

```
void func1(int n)
{
      int t=0;
      int sqr=sqrt(n);
      for( i=2;i<=sqr;i++)
      {
            if(n%i==0)
            {
               printf("Not a prime");
               t++;
               break;
            }
```

```
        }
        if(!t)
              printf("Prime no");

}
```

Let us explain how it works.
So let the number 'n' passed in the function. It definitely has two factors 1 and the number itself ("n"). So if we can find a number between 1 and n that can divide it perfectly we can say it is not a prime number. So the algorithm starts the loop that keeps checking whether the number is prime or not until it reaches the root of n (i.e $\sqrt{n}$). The reason for the range ending with $\sqrt{n}$ and not being n is below;
Let's assume a number k which has got more than 2 factors(other than 1 and k), then other factors will be in pairs in such a way that multiplying both numbers will give you k. The property of these pairs would be that one number is greater than the perfect/imperfect root($\sqrt{k}$) and the other is less than $\sqrt{k}$. So if we are sure that there is no factor of a number(k) from [2,$\sqrt{k}$]. Then it would not have a factor from ($\sqrt{k}$,k).
Now let's do an analysis, the loop in the above function runs for $\sqrt{n}$ times(if n is a prime number) which means the for loop will do $\sqrt{n}$ comparisons (i.e i<=sqr) ,$\sqrt{n}$ additions (i.e i++). Also, it will run if statement $\sqrt{n}$ times, which means $\sqrt{n}$ times modulus operator(%) and $\sqrt{n}$ times comparison operator(==) would also run.

3.2 Notation to Measure Time Complexity

The efficiency of the above function depends on the no of operation that will occur in the loop. Rest all the parts would be very insignificant compared to it. Here are some notation of how we can measure it in a better way.

α =No of times the loop will run.

β =No of time comparison happens in loop.

γ =No of times addition takes place in loop.

δ =No of times comparison operator will be executed till the end of loop.

ε =No of times modulus operator will be executed till the end of loop.

NOTE - ALL THE ABOVE PARAMETERS ARE CONSIDERED WHEN THE NUMBER INPUT IN THE FUNCTION IS PRIME.

Below we show the snippet of the loop in the above function in addition to what the above notation conveys as comments.

```
for( i=2 ; i<=sqr  /* ← β */ ; i++  /* ← γ */ ) // ← α
{
if(n%i   /* ← ε */  ==  /* ← δ  */  0)
          {
            printf("Not a prime");
            t++;
            break;
          }
}
```

Now lets see the value for the above function(i.e func1) -

$\alpha = \sqrt{n}$

$\beta = \sqrt{n}$

$\gamma = \sqrt{n}$

$\delta = \sqrt{n}$

$\varepsilon = \sqrt{n}$

3.3 Improving The Time Complexity

Now let's improve the function by which our parameters value decreases. Here is the code

```
void func2(int n)
{
      int t=0;
      int sqr=sqrt(n);
      if(n%2==0)
            t++;
      else
      {
            for( i=3;i<=sqr;i+=2)
            {
                  if(n%i==0)
```

```
                        {
                              printf("Not a prime");
                              t++;
                              break;
                        }
                  }
            }
      if(!t)
            printf("Prime no");
}
```

So here we tried to increase its efficiency. We first check the number is taken as input *n*, whether it's a multiple of *2* or not. If not, then we compare it against all the modulus of odd numbers.
The logic behind this step is that if *n* is not divisible by *2* then we can be sure that all other numbers which are divisible by *2* can never be able to divide the number.

$$\text{If} \quad n \bmod 2 = 0 .$$

$$\text{Then} \quad n \bmod k = 0 .$$

$$\text{Where} \quad k \bmod 2 = 0 .$$

So what is the advantage of it?
If we see the parameter notation for the above function we see the difference

$$\alpha = \frac{\sqrt{n}}{2}$$

$$\beta = \frac{\sqrt{n}}{2}$$

$$\gamma = \frac{\sqrt{n}}{2}$$

$$\delta = \frac{\sqrt{n}}{2}$$

$$\varepsilon = \frac{\sqrt{n}}{2}$$

We can see that the parameters have reduced by half . Which will reduce time complexity significantly.

Now before we go on further to make the function more efficient let's see some math-

let K_i *be a set. such that*

K_i = { *Set of first i prime no* | $n \in N \cup \{0\}$ }

$m_i = \prod_{c \in K} c$ (*i.e multiplication of all the numbers in the set* K_i)

$V_{i,t}$ = { $m_i t$, $m_i t + 1$, $m_i t + 2$, . . ., $m_i(t+1) - 1$ | *where* $t \in N \cup \{0\}$}

$A_{i,t}$ = {$\forall$ *ele which are not divisible by any element of* K_i| *where ele* $\in V_{i,t}$}

In func2 we saw that if the number is divisible by 2.Then we do not take out the modulus of n with any other number which is divisible by 2. Now what if we can do this process for number 2 as well as for number 3, Or for any first *i* prime numbers, where $i \in$ N.So to reflect the first *i* prime numbers we use K_i.
Now let us take an example of for $i = 2$
From now onwards when we refer *i* value We will refer the whole set of { K_i , m_i , $V_{i,t}$, $A_{i,t}$ }.

Let $i = 2$

Therefore $K_i = \{2, 3\}$ & $m_i = 6$

Let for $t \in N \cup \{0\}$ (whole no)

their be set $Q_t = \{6t , 6t+1 , 6t+2 , 6t+3 , 6t+4 , 6t+5\}$

Now in this set

$6t = 2 \times 3 \times t$ $\therefore$ *It is divisible by* 2, 3

$6t + 2 = 2 \times (3t + 1)$ $\therefore$ *It is divisible by* 2

$6t + 3 = 3 \times (2t + 1)$ $\therefore$ *It is divisible by* 3

$6t + 4 = 2 \times (3t + 2)$ $\therefore$ *It is divisible by* 2

But $6t + 1$ *and* $6t + 5$ *are not*

Also written as $6t \pm 1$ *if* $t \in N$

We can use the above result to make a more efficient function. Here's how to do it in func3 -

```
void func3(int n)
{
int t=0;
int sqr=sqrt(n);
      if(n%2==0||n%3==0)
            t++;
      else
      {
      for( i=6;i<=sqr;i+=6)
      {
            if(n%(i+1)==0||n%(i-1)==0)
            {
                  printf("Not a prime");
                  t++;
                  break;
            }
      }
}
      if(!t)
            printf("Prime no");
}
```

Now lets see the parameter notation value for this equation.

$$\alpha = \frac{\sqrt{n}}{6}$$

$$\beta = \frac{\sqrt{n}}{6}$$

$$\gamma = \frac{\sqrt{n}}{6}$$

$$\delta = \frac{\sqrt{n}}{3}$$

$$\varepsilon = \frac{\sqrt{n}}{3}$$

we here introduce another parameter called η

η = *No of times and* (&&) *operator will be executed till the end of loop.*

the value for η *in func*3 *is* $\eta = \frac{\sqrt{n}}{6}$

So, we see that efficiency has increased.We have set Q_t to reflect $V_{i,t}$ and the set of $\{6t \pm 1\}$ to denote $A_{i,t}$.

Now if we were to generalize the above notation in for any i. We would need Ki , mi , Vi,t , Ai,t which we can easily identify.
Now if we were to generalize the above notation in for any i. We would need K_i , m_i , $V_{i,t}$, $A_{i,t}$ which we can easily identify.

Therefore for a given *i the parameters are*

$$\alpha = \frac{\sqrt{n}}{m_i}$$

$$\beta = \frac{\sqrt{n}}{m_i}.$$

$$\gamma = \frac{\sqrt{n}}{m_i}$$

$$\delta = \frac{\sqrt{n} \times |A_{i,t}|}{m_i}$$

$$\varepsilon = \frac{\sqrt{n} \times |A_{i,t}|}{m_i}$$

$$\eta = \frac{\sqrt{n} \times (|A_{i,t}| - 1)}{m_i}$$

So we can see that as the *i* value increases the parameters value decreases, Implying that the time required to calculate will decrease.

Here is a flow chart depicting the whole scenario.

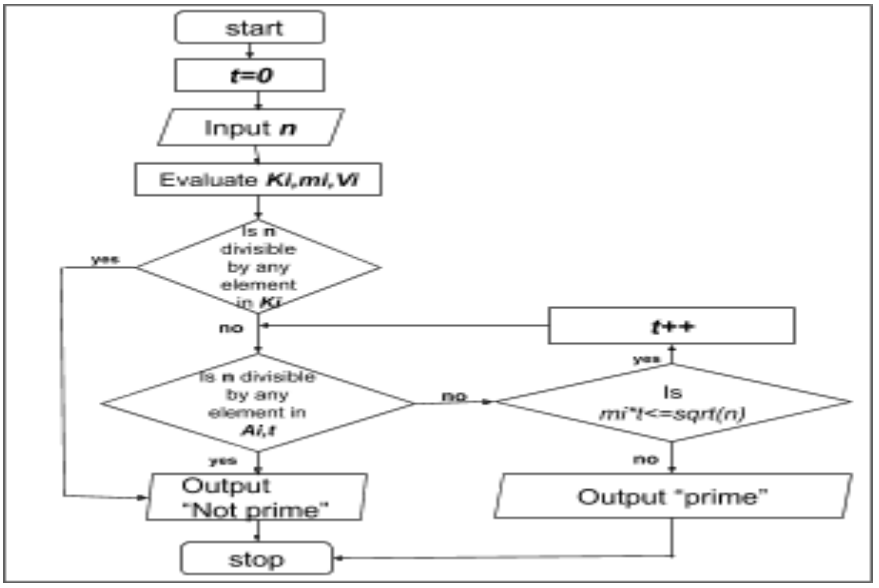

Fig. 1. Flow chart of the algorithm done using the notation above.

3.4 Entropy Analysis

Now let's do entropy analysis of the process. Before proceeding for the analysis, let's first define entropy.

Entropy can be seen as the probability of being a prime number while running an algorithm to perform modulus function with numbers of iterations. Similarly, let us define Redundancy, it can be seen as times when we perform modulus operation of an input n with respect to a number which is not prime.

Redundancy or R_i *for a particular i can be defined by* −

$$R(i) = \frac{|A_{i,t}|}{|V_{i,t}|} = \frac{|A_{i,t}|}{m_i}$$

Entropy will be defined as $H(i) = 1 - R(i)$

4 Experimental Setup

To do computation we are using ubuntu 16.04 LTS in a virtual box on a windows OS. Using Intel® Core™ i5-7200U CPU @ 2.50GHz , 64-bit system.

To test whether the theory is valid or not we measured the time taken by a function for i=0,1,2,3, to find out prime numbers between first 10^3,10^4,10^5,10^6,10^7 natural numbers.

Here is the C code to test out the efficiency of the code.

```
void func(int n)
{
          int prime=0;
          clock_t start,end;
          long double time;
          int k=0,order=1;
          int t=0,j=0,i=2;
          for( k=1;k<8;k++)
          {
           order=pow(10,k);
           prime=0;
           start=clock();
           for( j=2;j<order;j++)
           {

//The code to find out whether the given number (i.e j
variable in this code )
            /* Here is an example for a code.



            t=0;
            for( i=2;i<=sqrt(j);i++)
```

```
                if(j%i==0)
                {
                  //printf("Not a prime");
                  t++;
                  break;
                }
               if(t==0)
                prime++;
                //printf("%d ",j);

            Code end   */

}
             printf(" Order :- %d",k);
             printf(" No of prime :- %-8d",prime);
             end=clock();
             time = ((long
  double)(end-start))/CLOCKS_PER_SEC;
             printf("  Time taken = %Lf \n",time);
           }
           printf("\n\n Func  End");
  }
```

5 Results

We see that as we increase the value of i (i.e increase the set K_i) the time taken decreases significantly.

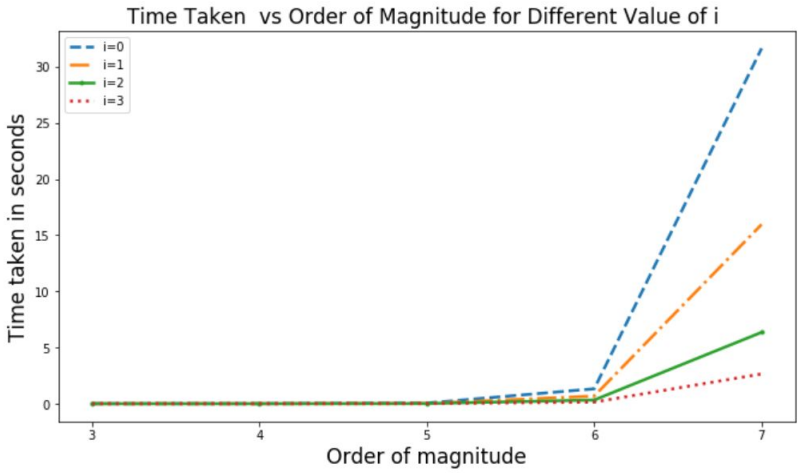

Fig. 2. Results showing how the time is taken for different *i* value .The plot shows us that as we increase the *i* value the time taken by the function decreases.

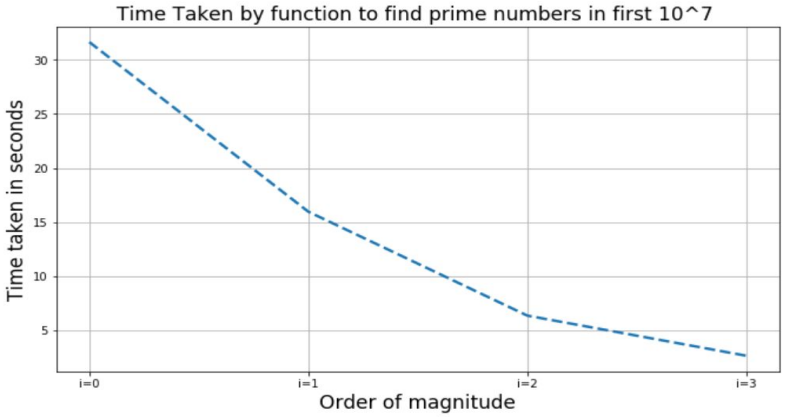

Fig. 3. This figure shows how the time required to identify all the prime numbers in the first 10^7 numbers.We see that the trend is decreasing showing how that we should increase our *i* for better efficiency.

Table 1. Showing the Readings of the above experiment. The readings are in seconds.

Value of *i*	0	1	2	3
Order of magnitude				
3	0.000103	0.000049	0.00003	0.000005
4	0.002161	0.000994	0.000544	0.000304
5	0.054234	0.022953	0.012051	0.005598
6	1.323016	0.679118	0.325649	0.140055
7	31.650303	15.965183	6.355153	2.647136

6 Conclusion

In conclusion,e can safely say that with the increase in *i* value the time complexity decreases.

7 References

1. K. H. Rosen and K. Krithivasan, "Discrete mathematics and its applications(with combinatorics and graph theory),7e," p. 223.
2. T. Desai, "Application of Prime Numbers in Computer Science and the Algorithms Used To Test the Primality of a Number," International Journal of Science and Research(ISJR), pp. 3–4, 2013.
3. C. T. Long, Elementary introduction to number theory, 1972.
4. Sieve of Eratosthenes. [Online]. Available: http://www.geeksforgeeks.org/sieve-of-eratosthenes/
5. L. Monier, "Evaluation and comparison of two efficient probabilistic primality testing algorithms," Theoretical Computer Science, vol. 12, no. 1, pp. 101–102, 1980.
6. C. Pomerance, J. L. Selfridge, and S. S. Wagstaff, "The pseudoprimes to 25·109," Mathematics of Computation, vol. 35, no. 151, pp. 1003–1026, 1980.
7. E. W. Weisstein, "Fermat's little theorem," 2004.
8. A.V.Bajandin, To distribution of simple numbers to set of natural integers. - Novosibirsk 19 999 .27 p.
9. A.V Cheryomushkin, Lectures on arithmetic algorithms in cryptography - Moscow 2002. 104 p.
10. S.H. Bokhari, Multiprocessing the sieve of Eratosthenes, IEEE Computer 20(4): April 1984, pp.50-58

11.G. Pailard, C. Lavult, and R. Franca, A distributed prime sieving algorithm based on Scheduling by Multiple Edge Reversal, Int'l Symp. On Parallel and Distributed Computing, Univ. Lille 1, France, July 4-6, 2005.
12.R. Rivest, A. Shamir, L. Adleman, A method for obtaining digital signatures and public key cryptosystems, Comm. ACM 21 (Feb. 1978), pp. 120126