Generating prime numbers

Žiga Pušnik

University of Ljubljana
Faculty of Computer and Information Science

Cryptography

February 7, 2017

Contents

1 Abstract

The sieve of Atkin [1] is a novel algorithm for generating prime numbers in linear time $O(n)$. While the running time of the algorithm is a big improvement from the sieve of Eratosthenes [2] and the sieve of Sundaram [3], which runs in $O(n \log \log n)$ and $O(n \log n)$ time respectively, the linear running time do not allow us to generate big prime numbers with length more than 1024 bits. Big prime numbers are thus created probabilistically. The algorithms for generating huge prime numbers are also described in the digital signature standard DSA [4].

2 Introduction

Well over 2000 years ago the Euclid have proven that there is infinitely many of prime numbers. The proof is quite simple, say that $p_1 = 2 < p_2 = 3 < ... < p_k$ is the list of all of the possible primes. If we create the number $P = p_1 * p_2 * ... * p_k + 1$. None of the primes from p_1 to p_k cannot divide P, because they would also need to divide their difference $P - p_1 * p_2 * ... * p_k$, which is not possible. Therefore we are left with only two possible solutions, either P is prime, or there exists other prime factor that is not included in list of all possible primes. Either way, the list of all possible prime numbers is not complete and we can keep adding prime numbers until we are left out of time.

Prime numbers are fascinating and because of the interesting nature of prime numbers the mathematicians devised the simple algorithms to list all prime numbers up to some limit from the ancient time. One example of such simple algorithm is sieve of Eratosthenes. While through time and the introduction of computers the algorithms became more and more complex.

In modular arithmetic with generator $< a >$ where modulus of the congruence n form a multiplicative group $\mathbb{Z}_n^*$ of integers modulo n. n is prime if and only if the order of the group is $n-1$. Because of this property the large prime numbers are fundamental in modern cryptography and are used in RSA cryptosystem, ElGamal encryption and in elliptic curve cryptography.

For security reasons the large prime numbers can not be simply recycled from the pre-computed table of large prime numbers and need to be calculated separately. Because the deterministic algorithms are not fast enough, the stohastic generation of prime numbers is required. The general strategy goes as follows: randomly create a big odd number and test if the number is prime with probabilistic primality testing like Solovay–Strassen or Miller-Rabin test. If the number is not prime, generate new random odd integer. The deterministic algorithm for primality testing exists [5], however it is not efficient for large prime numbers.

In this seminar paper we describe the most basic sieve algorithms for finding prime numbers up to some limit integer and implement the sieve of Atkin, one of the novel sieve algorithms for identifying prime numbers. We also evaluate the performance and the difference between naive and efficient implementation of the algorithm. We also describe how big primes are generated in RSA cryptosystem.

3 Sieves

Sieve theory is a set of general techniques designed to count or estimate the sets of sifted sets of integers. The main example are prime numbers.

One such sieve is sieve of Eratosthenes [2]. It is one of the oldest and efficient algorithms designed to find prime numbers less than or equal to a given number N by eliminating all multiples of a prime number. By doing that incrementally we are left with only prime numbers. Sieve of Eratosthenes is efficient in terms of $O(n \log \log n)$ time complexity and it is not suitable for generating large prime numbers.

However there are more advanced algorithms like the sieve of Sundaram [3]. Sieve of Sundaram is simple deterministic algorithm which eliminates all numbers of a form $i+j+2ij$. The remaining integers are doubled and incremented by one. By doing so the sieve of Sundaram achieves $O(n \log n)$ time complexity. We can quickly prove the correctness of the algorithm. If q is an odd integer of a form $2k + 1$, then q is eliminated if and only if k is of a form $i + j + 2ij$. Since such q can be factorized in $(2i + 1)(2j + 1)$ the only possible odd numbers that are still left are prime numbers.

One of the novel and most efficient algorithms in this group is sieve of Atkin [1] which has the $O(n)$ time complexity. In the next few sections we present the algorithm, theoretical background and implementation. While naive implementation may not be as efficient, we improve the efficiency by exploiting the fundamental facts about prime numbers.

3.1 Sieve of Atkin

Sieve of Atkin [1] is a modern algorithm created in 2003 by mathematicians A. O. L. Atkin and Daniel J. Bernstein. It is designed to find all prime numbers up to a specified integer. Unlike sieve of Eratosthenes, which eliminates all multiples of a prime number, the sieve of Atkin eliminates all multiples of a squared prime. By doing some preliminary work, the sieve of Atkins finds candidates for prime numbers and thus achieves better asymptotic complexity than the ancient sieve of Eratosthenes. It is based on the three main theorems proved by Atkin and Bernstein in original paper [1]. The theorems are described below.

Theorem 1

Let n be of a form: $n = 4Z + 1$, then n is prime if and only if the number of solutions to the equation 1 is odd and n is not square-free. In other words n is not divisible by no other perfect square than 1.

Theorem 2

Let n be of a form: $n = 6Z + 1$, then n is prime if and only if the number of solutions to the equation 2 is odd and n is not square-free.

Theorem 3

Let n be of a form: $n = 12Z + 11$, then n is prime if and only if the number of solutions to the equation 3 is odd for every y smaller than x and n is not square-free.

Let r be n *mod* 60. We can observe that if r belongs to $\{1, 13, 17, 29, 37, 41, 49, 53\}$ then n is also congruent to one modulo four. If r belongs to $\{7, 19, 31, 43\}$ then n is also congruent to one modulo six. If r belongs to $\{11, 23, 47, 59\}$ then n is also congruent to eleven modulo twelve.

$$4x^2 + y^2 = n \tag{1}$$

$$3x^2 + y^2 = n \tag{2}$$

$$3x^2 - y^2 = n \tag{3}$$

3.1.1 Naive implementation

The naive implementation of the algorithm is pretty straightforward. Say we want to calculate all prime numbers up to some limit integer l. First we crate a sieve list with an entry for each possible positive integer. Each entry must initially be marked as non-prime. With double for loop we iterate through all $x's$ and $y's$ from 1 to$\sqrt{l}$ and calculate $n_1 = 4x^2 + y^2$, $n_2 = 3x^2 + y^2$ and $n_3 = 3x^2 - y^2$. If n_1 is smaller than l and n_1 *mod* 60 belongs to the set $\{1, 13, 17, 29, 37, 41, 49, 53\}$, we flip the prime flag for the integer n_1. Similarly if n_2 is smaller than l and n_2 *mod* 60 belongs to the set $\{7, 19, 31, 43\}$, we flip the prime flag for the integer n_2. We calculate n_3 only if x is greater than y, and we flip the prime flag only if n_3 is smaller than l and if n_3 *mod* 60 belongs to the set $\{11, 23, 47, 59\}$.

By calculating candidates for prime numbers we can now take the smallest integer in the sieve list still marked prime and include it in the result list. We must also mark every multiple of that squared integer as non prime.

The implementation was also tested for the correctness.

The code for naive implementation in programming language C++ is shown below:

```
bool * primes = (bool *)calloc(limit, sizeof(bool));

long long n;
int modResult;
long long iterLimit = (long long)sqrt((double)(limit));

for (long long i = 1; i < iterLimit; i++) {
  for (long long j = 1; j < iterLimit; j++) {
        n = 4*i*i + j*j;
        modResult = n % 60;
        if (n < limit && (modResult == 1 || modResult == 13 || modResult == 17 || modResult == 29 ||
             modResult == 37 || modResult == 41 || modResult == 49 || modResult == 53)) {
          primes[n-1] = !primes[n-1];
        }
        n = 3*i*i + j*j;
        modResult = n % 60;
        if (n < limit && (modResult == 7 || modResult == 19 || modResult == 31 || modResult == 43)) {
          primes[n-1] = !primes[n-1];
        }
        if(j < i){
          n = 3*i*i - j*j;
          modResult = n % 60;
          if (n < limit && n > 0 && (modResult == 11 || modResult == 23 || modResult == 47 || modResult ==
                59)) {
                primes[n-1] = !primes[n-1];
          }
        }
  }
}
for (long long n = 3; n < limit; n += 2) {
  if (primes[n - 1]) {
        //write off all multiple of prime
        long long ns = n*n;
        long long mns = ns;
        while (mns < limit) {
```

```
            primes[mns - 1] = 0;
            mns = mns + ns;
          }
        }
    }
}
```

3.1.2 Efficient implementation

While naive implementation is really straightforward and simple, it is not as efficient. In naive implementation we iterate through all possible $x's$ and $y's$, thus calculating the equations 1, 2 and 3 for even and odd $n's$, however we can greatly reduce the total number of needed calculations simply by following the fact that n must be odd. The disadvantage of this approach is that we must optimize every equation separately.

A big improvement from the naive implementation is a substitution of multiplication with the addition. In equation 4 we can observe, that the difference between consecutive square numbers is linear and is increasing by a constant factor. By doing so we decrease the time complexity from $O(\log^2 n)$ needed for multiplication to $O(\log n)$ needed for addition, where $\log n$ is proportional to the number of bits in integer n.

The implementation was also tested for the correctness.

For the equation $n = 4x^2 + y^2$ we can iterate only through even $y's$ since the term $4x^2$ is always odd. Furthermore if the term $4x^2$ is divisible by 3, then n is also divisible by 3 if y is divisible by 3. In this manner we can iterate only through even $y's$ skipping every third y. The code for this section is shown below and it is written in programming language C++:

```
//section 4x^2 + y^2 = n
long long xstf = 0;
long long xstfDiff = 4;
long long ysDiff = 0;
long long n;
long long ys;
int modResult;
long long iterLimit = (long long)sqrt((limit - 1) / 4.0f);

//correct result give all x's and only odd y's
for (long long i = 1; i <= iterLimit; i++) {
  xstf += xstfDiff;
  xstfDiff += 8;
  //printf("%d \n", xstf);

  if (xstf % 3 == 0) {
    //y^2 must not be divisible by 3, thus we ignore every third solution
    n = xstf + 1;
    ys = 1;
    ysDiff = -24;
    while (n < limit) {
      modResult = n % 60;
      if (modResult == 1 || modResult == 13 || modResult == 17 || modResult == 29 || modResult == 37 ||
          modResult == 41 || modResult == 49 || modResult == 53) {
        primes[n - 1] = !primes[n - 1];
      }
      ysDiff += 72;
      n += ysDiff;
      ys += ysDiff;
    }
    n = xstf + 25;
    ys = 25;
    ysDiff = 24;
    while (n < limit) {
      modResult = n % 60;
      if (modResult == 1 || modResult == 13 || modResult == 17 || modResult == 29 || modResult == 37 ||
          modResult == 41 || modResult == 49 || modResult == 53) {
        primes[n - 1] = !primes[n - 1];
      }
      ysDiff += 72;
      n += ysDiff;
      ys += ysDiff;
    }
  }
  else {
    n = xstf + 1;
    ys = 1;
    ysDiff = 0;
```

```
    while (n < limit) {
      modResult = n % 60;
      if (modResult == 1 || modResult == 13 || modResult == 17 || modResult == 29 || modResult == 37 ||
          modResult == 41 || modResult == 49 || modResult == 53) {
        primes[n - 1] = !primes[n - 1];
      }
      ysDiff += 8;
      n += ysDiff;
      ys += ysDiff;
    }
  }
}
```

For the equation $n = 3x^2 + y^2$ we can observe that the term $3x^2$ is always divisible by 3 and in order that $n \ mod \ 60$ could belong to the set $\{7, 19, 31, 43\}$, y must be even and x must be odd. We also can ignore every third y since it will be divisible by 3. The code written in programming language C++ for this section is shown below:

```
//section 3x^2 + y^2 = n
xstf = 3;
xstfDiff = 0;
ysDiff = 0;
iterLimit = (long long)sqrt((limit - 1) / 3.0f);
iterLimit = iterLimit / 2 + 1;
//only even x's and odd y's
for (long long i = 1; i <= iterLimit; i++) {
  xstf += xstfDiff;
  xstfDiff += 24;
  n = xstf + 4;
  ys = 4;
  ysDiff = -12;
  while (n < limit) {
    modResult = n % 60;
    modResult = n % 60;
    if (modResult == 7 || modResult == 19 || modResult == 31 || modResult == 43) {
      primes[n - 1] = !primes[n - 1];
    }
    ysDiff += 72;
    n += ysDiff;
    ys += ysDiff;
  }
  n = xstf + 16;
  ys = 16;
  ysDiff = 12;
  while (n < limit) {
    modResult = n % 60;
    if (modResult == 7 || modResult == 19 || modResult == 31 || modResult == 43) {
      primes[n - 1] = !primes[n - 1];
    }
    ysDiff += 72;
    n += ysDiff;
    ys += ysDiff;
  }
}
```

For the equation $n = 3x^2 - y^2$ we can iterate only through even-odd and odd-even combinations. Again we can completely ignore every third y. If the term $3x^2$ is greater than the limit integer we can only calculate minimum y so that the term n is still within the limits. Note that y must not be greater than x. By this inequality we limit the possible number of solutions from infinity to some integer. The code for the last section written in C++ is shown below:

```
// section 3x^2 - y^2 = n
// x must be greater than y
xstf = 0;
xstfDiff = 3;
ysDiff = 0;

iterLimit = (long long)sqrt((limit) / 2.0f) + 1;
long long ydiff;

for (long long i = 1; i < iterLimit; i++) {
  xstf += xstfDiff;
  xstfDiff += 6;
  if (xstf % 2 == 1) {
    //x is odd, y must be even
    long long ymin = 2;
    if (xstf > limit) {
      ymin = (((long long)sqrt((double)(xstf - limit)) >> 1) << 1);
      if (ymin % 6 == 0) {
        ymin = ymin + 2;
      }
```

```
      else if (ymin % 6 == 4) {
        ymin = ymin + 4;
      }
    }

    ydiff = 12 * ymin + 36;
    ys = ymin*ymin;
    n = xstf - ys;

    while (n > 0 && ymin < i)  {
      if (n < limit) {
        modResult = n % 60;
        if (modResult == 11 || modResult == 23 || modResult == 47 || modResult == 59) {
          primes[n - 1] = !primes[n - 1];
        }
      }
      ymin = ymin + 6;
      ys += ydiff;
      ydiff += 72;
      n = xstf - ys;
    }

    ymin = 4;
    if (xstf > limit) {
      ymin = (((long long)sqrt((double)(xstf - limit)) >> 1) << 1);
      if (ymin % 6 == 0) {
        ymin = ymin + 4;
      }
      else if (ymin % 6 == 2) {
        ymin = ymin + 2;
      }
    }
    ydiff = 12 * ymin + 36;
    ys = ymin*ymin;
    n = xstf - ys;

    while (n > 0 && ymin < i)  {
      if (n < limit) {
        modResult = n % 60;
        if (modResult == 11 || modResult == 23 || modResult == 47 || modResult == 59) {
          primes[n - 1] = !primes[n - 1];
        }
      }
      ymin = ymin + 6;
      ys += ydiff;
      ydiff += 72;
      n = xstf - ys;
    }
  }
  else {
    //x is even, y must be odd
    long long ymin = 1;
    if (xstf > limit) {
      ymin = (((long  long)sqrt((double)(xstf - limit)) >> 1) << 1) - 1;
      if (ymin % 3 == 0) {
        ymin = ymin + 4;
      }
      else if (ymin % 3 == 2) {
        ymin = ymin + 2;
      }
    }
    ydiff = 12 * ymin + 36;
    ys = ymin*ymin;
    n = xstf - ys;

    while (n > 0 && ymin < i)  {
      if (n < limit) {
        modResult = n % 60;
        if (modResult == 11 || modResult == 23 || modResult == 47 || modResult == 59) {
          primes[n - 1] = !primes[n - 1];
        }
      }
      ymin = ymin + 6;
      ys += ydiff;
      ydiff += 72;
      n = xstf - ys;
    }
    ymin = 5;
    if (xstf > limit) {
      ymin = (((long  long)sqrt((double)(xstf - limit)) >> 1) << 1) - 1;
      if (ymin % 3 == 0) {
        ymin = ymin + 2;
      }
      else if (ymin % 3 == 1) {
        ymin = ymin + 4;
      }
    }
    ydiff = 12 * ymin + 36;
    ys = ymin*ymin;
    n = xstf - ys;

    while (n > 0 && ymin < i)  {
```

```
      if (n < limit) {
        modResult = n % 60;
        if (modResult == 11 || modResult == 23 || modResult == 47 || modResult == 59) {
          primes[n - 1] = !primes[n - 1];
        }
      }
      ymin = ymin + 6;
      ys += ydiff;
      ydiff += 72;
      n = xstf - ys;
    }
  }
}
```

$$(x+1)^2 = x^2 + 2x + 1 \tag{4}$$

3.1.3 Space and time complexity

The algorithm described above can compute prime numbers using $O(n)$ operations and $O(n)$ bits of memory. This may be unintuitive because of the three series of quadratic equations, however Atkin and Bernstein showed that when the range goes to the infinity, each quadratic form has a constant ratio of operations. Also by naive implementation we can observe that x and y could not exceed$\sqrt{n}$, thus achieving time complexity $O(\sqrt{n} * \sqrt{n}) = O(n)$.

While the time complexity still may be inefficient for calculating 1024 bits primes, the algorithm offers significant improvement from the sieve of Eratosthenes, which has $O(n \log \log n)$ time and $O(n)$ space complexity.

3.1.4 Results and discussion

We tested the naive and efficient implementation on the same machine with 4GB DDR3 RAM on Intel Q9550 processor with frequency of 2.89 Ghz.

From scatter plot 1 we can observe that naive and efficient implementation have $O(n)$ time complexity, however the efficient implementation greatly reduces the asymptotic constant. For efficient implementation the constant is $3.212 * 10^{-8}$ while for the naive implementation the constant is $1.056 * 10^{-7}$. And while efficient implementation is much faster it would still take more than $1.1786 * 10^{62}$ years to calculate all prime numbers up to 2^{256} and even for 64 bit primes the efficient implementation on the same machine would take more than 10000 years.

The algorithm could also be parallelized to improve the efficiency. But by doing so we would only decrease the asymptotic constant and not the time complexity itself. To generate large primes we need to take some different approaches.

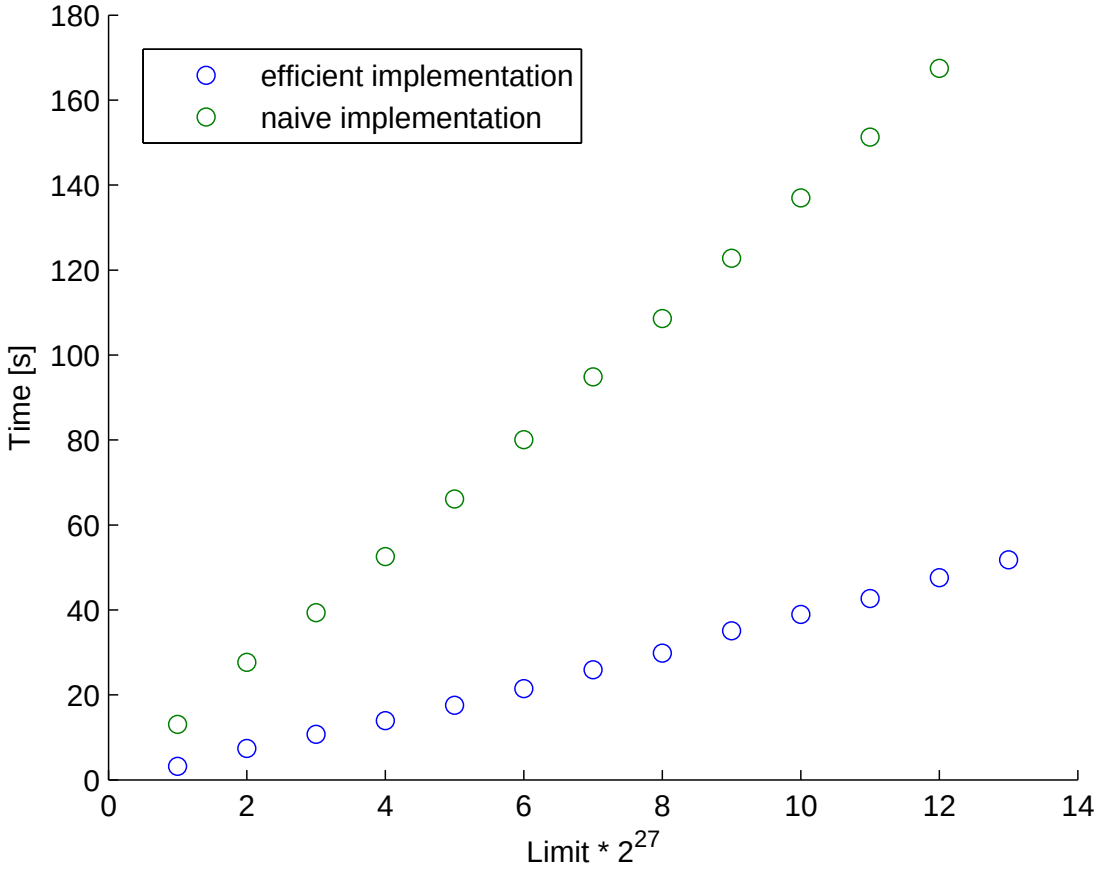

Figure 1: Running time for naive and efficient implementation of the sieve of Atkin.

4 Prime numbers in RSA

RSA is a public key cryptosystem widely used in secure data transmission. RSA was first publicly described in 1977 [6]. The encryption key is public and is different from private decryption key. The public key consist of encryption key and the modulus of the congruence (e, n). The private key consist of decryption exponent d.

Say that Alice wants to send a message m to Bob, so that only Bob can read it. She computes cipher text $c \equiv m^d \ mod \ n$ using Bob public key. Bob can then decrypts the message $m \equiv c^d \ mod \ n$. If Alice also encrypts the cipher text using her private key, then Bob can confirm that the message came from Alice by decrypting the cypher text using her public key.

The keys are generated in the following way:

1. Choose two different prime numbers p and q.
2. Compute $n = pq$.
3. Compute $\phi(n) = (p-1)(q-1)$.
4. Choose an integer e, such that $1 < e < n$ and $gcd(e, \phi(n)) = 1$.
5. Calculate d so that $de \equiv 1 mod \phi(n)$.

While encryption key should not be to large, the two different prime numbers p and q must be huge, more than 1024 bits in length.

The protocols for generating the prime numbers p and q are described in digital signature standard DSS [4]. The DSS specifies algorithms for applications requiring a digital signature for the digital signature algorithm DSA, RSA and for the elliptic curve signature algorithm. DSS is published in Federal Information Processing Standards Publications.

The DSS also describes the method for generating large probable primes, that are 1024, 2048 or 3072 bit long. Both p and q should satisfy some additional conditions:

- $p-1$ has a prime factor p_1
- $p+1$ has a prime factor p_2
- $q-1$ has a prime factor q_1
- $q+1$ has a prime factor q_2

Primes p and q are called strong primes and the reason for this is that the Pollard p - 1 algorithm is especially suited for primes p, when $p-1$ or $p+1$ has only small factors. However RSA has is own weaknesses that can be exploited for the attack [7].

The probable prime numbers are randomly generated and tested using probabilistic primality test like the Miller-Rabin primality test. The numbers should be tested at least few times to reduce the error probability. The standard way of proving a number is prime is by showing it and 1 are its only factors. The Miller-Rabin primality test exploits the fact that if the number n is prime, then for some $1 \leq a < n$, the only possible solutions for $a^2 \equiv 1$

mod n are 1 and -1. In other words, if number n is prime, a^2 has only trivial square roots mod n. If we are able to find some non-trivial square roots, then the number n must be composite. The Miller-Rabin primality test returns the answer "composite" if number n is composite, otherwise the algorithm return the answer "probably prime". Since the worst case of the error probability of the Miller-Rabin primality test is 4^{-k} with k repeated test, the recommended number is at least 64 times for all prime numbers p and q if we want to reduce the error probability below 2^{-128}. If the number is not prime, new candidate number should be randomly generated until a strong prime number is obtained. However if the primality testing returns the wrong answer that a composite number is prime, the worst thing that can happen is that our message m is not deciphered correctly. In that case the sender should repeat a process and pick new pair of private and public key.

And while the algorithm AKS (Agrawal, Kayal, Saxena) performs the deterministic primality testing in polynomial time [5]. The time complexity does not allow us to test large primes that are bigger than 1024 bits. However the algorithm gives us the answer that the deterministic primality testing is possible in polynomial time and the algorithm can still be used to test smaller candidates for prime numbers.

5 Conclusion

We implemented the sieve of Atkin in programming language C++ and compared the efficiency of naive and efficient implementation of the algorithm. And while the running time of the efficient implementation is much faster than the naive implementation, the $O(n)$ time complexity does not allow us to generate very large primes. It would take more than $1.1786 * 10^{62}$ years to calculate all prime numbers up to 2^{256}. And while the parallelization is possible and would greatly reduce the running time, the time complexity of algorithm would still be $O(n)$.

In cryptography the need for large prime numbers is vital and for that purposes the prime numbers are generated randomly and tested with probabilistic primality testing algorithm like Solovay–Strassen or Miller-Rabin test. With the repeated tests we reduce the probability of error to some negligible value. For example to reduce the error to 2^{-128}, the Miller-Rabin primality test should be repeated more than 64 times.

The AKS algorithm returns the answer in polynomial time, however it is still not efficient for very large integers.

References

[1] A Atkin and D Bernstein. Prime sieves using binary quadratic forms. *Mathematics of Computation*, 73(246):1023–1030, 2004.

[2] Nicolaas Govert De Bruijn. On the number of uncancelled elements in the sieve of eratosthenes. *Indag. math*, 12:247–256, 1950.

[3] SP Sundaram and VR Aiyar. Sundaram's sieve for prime numbers. *The Mathematics Student*, 2:73, 1934.

[4] Dan Boneh. Digital signature standard. In *Encyclopedia of Cryptography and Security*, pages 347–347. Springer, 2011.

[5] Martin Dietzfelbinger. *Primality testing in polynomial time: from randomized algorithms to" PRIMES is in P"*, volume 3000. Springer, 2004.

[6] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[7] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs# 1. In *Annual International Cryptology Conference*, pages 1–12. Springer, 1998.