Primality Testing

Randomized Algorithm Project Report

Debraj Karmakar (220329) Sankalp Mittal (220963) Naman Kumar Jaiswal (220687)

April 11, 2025

Contents

1 Problem Statement

Primality testing has long been a fascinating problem, due to the need of large prime numbers (512 to 2048 bit long) in almost all cryptographic encryption techniques such as Diffie-Hellman key exchange or RSA encryption. A breakthrough deterministic polynomial-time algorithm for primality testing was developed at IIT Kanpur, marking a proud milestone for the institute.
However, despite its theoretical significance, randomized algorithms remain preferred due to their significantly smaller exponent and greater simplicity. These algorithms use **basic modular arithmetic, elementary probability techniques, and Fermat's little theorem** to test primality in significantly less amount of time as compared to the deterministic algorithm. Additionally, they provide strong probabilistic guarantees on correctness. However, certain exceptional cases consistently yield incorrect results, limiting straightforward probability amplification. To overcome this, further refinements are necessary, as discussed in this report. This leads to a simple **randomized Monte-Carlo** algorithm for primality testing.
In this project, we test primality for an n-bit number N. [i.e. $n = \mathcal{O}(\log N)$].

2 Deterministic Algorithms

2.1 $\mathcal{O}(\sqrt{N})$-time Algorithm

By definition of primes, a number N is prime if it has no positive divisors other than 1 and itself. This leads to an algorithm that checks whether any integer d in the range $[2, \sqrt{N}]$ divides N. If any such divisor is found, N is composite; otherwise, it is prime.

Algorithm 1 Primality Testing (Fermat)

```
1: function CHECK_PRIMALITY($N$)
2:     for $i = 2$ to $\lfloor\sqrt{N}\rfloor$ do
3:         if $N\%i = 0$ then
4:             return Composite
5:         end if
6:     end for
7:     return Prime
8: end function
```

In the Word RAM model, this algorithm takes $\mathcal{O}(\sqrt{N}) = \mathcal{O}(2^{\frac{n}{2}})$ time to test primality of N.

2.2 AKS Primality Test

AKS primality test was introduced in 2002 at IIT Kanpur. It was the first deterministic primality testing algorithm proven to run in polynomial time in the size of the input that is $\mathcal{O}(n^6) = \mathcal{O}((\log N)^6)$ time for a number N.

3 Need for Randomization

Despite the existence of deterministic polynomial time algorithms for primality testing, their practicality is often limited due to high computational overhead. For instance, testing primality of a 1000 bit number would take around 10^{18} operations by AKS algorithm, which is very inefficient for real world applications.
Randomized algorithms often offer a powerful alternative. They are typically easier to implement

and provide guarantees of correctness, that can be made arbitrarily small by repeating the test with independent random inputs. In practice, these errors are negligible, making such algorithms reliable for most use cases.

4 A Randomized Algoritrhm

We start of by trying to design a randomized algorithm using **Fermat's Theorem**.

4.1 Fermat's Theorem

Theorem 1 (Fermat). *If p is a prime number and a is any positive integer such that $a < p$ then the following is true*

$$a^{p-1} \equiv 1 \mod p$$

As we can see this theorem can be really useful to eliminate numbers that are not prime. Consider any number N for which we wish to find out the primality. If we choose a base a *randomly uniformly* in the range 1 to $N-1$ we will try to find out the probability it satisfies the *Fermat's Condition* which is

$$a^{N-1} \equiv 1 \mod N$$

Lemma 1. *If $gcd(a, N) \neq 1$ then $\forall$ such a*

$$a^{N-1} \not\equiv 1 \mod N$$

Proof. Let $gcd(a, N) = k(\neq 1) \implies a^{N-1}$ and N have a gcd that is a multiple of k. This means if we write a^{N-1} as $qN + r$ then since qN has k as a factor and so does $a^{N-1} \implies$ so does r, hence $a^{N-1} \equiv r \mod N$ with $r \neq 1$ □

Lemma 2. *Consider the set*

$$S = \{a : 0 < a < N, a \in \mathbb{Z}, gcd(a, N) = 1\}$$

Then if for any base $a \in S$

$$a^{N-1} \not\equiv 1 \mod N$$

i.e. a fails the Fermat's test, then at least half of the elements in S will also fail the Fermat's test

Proof. Consider two distinct numbers $a, b \in S$ such that $a^{N-1} \not\equiv 1 \mod N$ say $a^{N-1} \equiv k \mod N$ (for some $k \neq 1$) and $b^{N-1} \equiv 1 \mod N$ since $gcd(a, N) = gcd(b, N) = 1 \implies gcd(ab, N) = 1$. Consider

$$(ab)^{N-1} \mod N$$

Then $(ab)^{N-1} \equiv k \mod N \implies$ for every b that satisfies the *Fermat's Condition* there will be another number in S that does not satisfy the condition given at least one such number exists. This means at least half of the numbers in S do not satisfy *Fermat's Condition* □

Using the above results we can do the following analysis for different categories

- **Prime Numbers:** For this the probability is 1 as all bases work
- **Composite Numbers:** Consider two cases based on $gcd(a, N)$
  - **gcd $\neq$ 1** If $gcd(a, N) \neq 1 \implies$ that the Fermat's condition is never satisfied

- **gcd = 1** If $gcd(a, N) = 1$ then if Fermat's condition fails for even one base then at-least half of these bases will satisfy the condition $\implies$ probability of choosing a base that fails is $\geq \frac{1}{2}$

These results seem really promising but we have **still not proved** that we can use Fermat's Theorem for all possible numbers but since this seems good enough we will design a simple randomized algorithm and then do some experiments to see how the results look like.

4.2 The Algorithm

Algorithm 2 Primality Testing (Fermat)

```
function CHECK_PRIMALITY($N$)
    if $N = 2$ then
        return Prime
    end if
    if $N\%2 = 0$ then
        return Composite
    end if

    $t \leftarrow$ # of iterations
    for $i = 1$ to $t$ do
        $a \leftarrow$ Pick a random integer randomly uniformly from $[1, N-1]$
        if $a^{N-1} \neq 1 \pmod N$ then
            return Composite
        end if
    end for
    return Prime
end function
```

4.2.1 Analysis

Assuming we only choose N that are either prime or composites such that the probability of *Fermat's Condition* not being satisfied are $\geq \frac{1}{2}$. If the algorithm outputs *Composite* it is always correct and if it outputs *Prime* it is wrong with probability $\leq \frac{1}{2}$. Hence after t iterations it is wrong with probability $\leq (\frac{1}{2})^t$. This takes time $\mathcal{O}(t \cdot n^2) = \mathcal{O}(t \cdot (\log N)^2)$.
[Considering that multiplying two r-bit numbers takes $\mathcal{O}(r)$ time, ignoring logarithmic factors]
We cannot prove that this algorithm will not give the correct value for all values of N so we will run some experiments to analyse on which values (if any) the algorithm fails on.

4.3 Experimentation

Calculating the probabilities for a base that is co-prime to the number to fail the *Fermat's Test* for all composite numbers from 2 to 20000 the following was the distribution

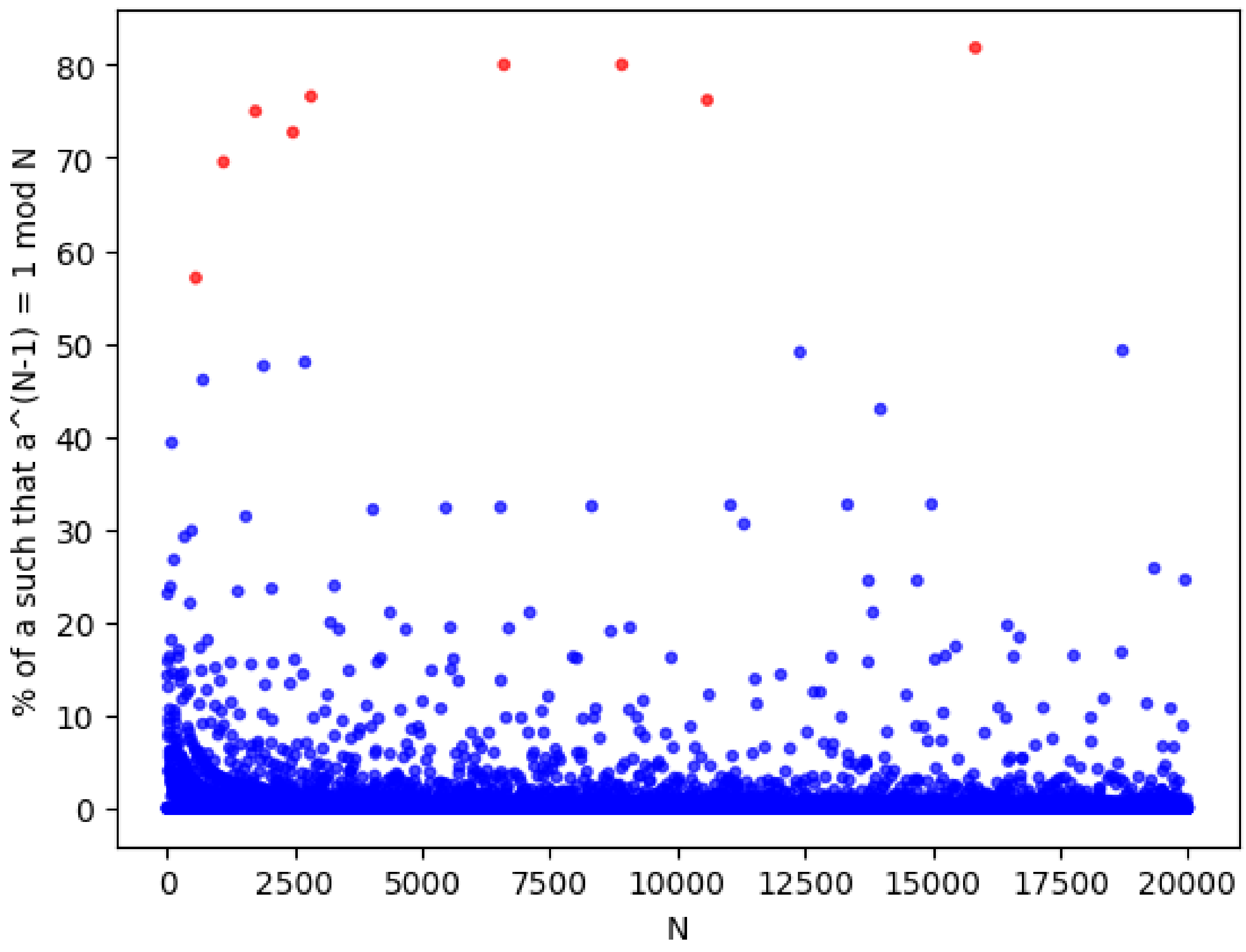


Figure 1: Scatter plot of number of bad a vs N

As we can clearly see there are some clear outliers in the figure although they are a really small fraction of values there are still some values that fail for all bases. These are a major roadblock towards our current algorithm and it will consistently give wrong answers on these and hence we cannot give any upper bounds on the error probabilities for these values. Therefore we need to think of some alternative algorithm.

5 Another Randomized Algorithm

5.1 Idea

Firstly, if N is even and $N \neq 2$ then we declare it to be composite and end the algorithm. If $N = 2$, we declare it to be prime.
Consider the case when N is odd. Let $N - 1 = 2^k \cdot m$ where $k > 0$ and is maximal, i.e. m is odd. For all $a \in \mathbb{Z}$, we can factorize $a^{N-1} - 1$ as,

$$a^{N-1} - 1 = a^{2^k \cdot m} - 1 = (a^{2^{k-1} \cdot m} + 1) \cdot (a^{2^{k-2} \cdot m} + 1) \ldots (a^{2^1 \cdot m} + 1) \cdot (a^m + 1) \cdot (a^m - 1)$$

As per Fermat's theorem, if N is prime then $a^{N-1} - 1$ is divisible by N. Thus, at least one of

$$(a^{2^{k-1} \cdot m} + 1), (a^{2^{k-2} \cdot m} + 1), \ldots, (a^{2^1 \cdot m} + 1), (a^m + 1), (a^m - 1)$$

is divisible by N.

So, we choose a randomly uniformly from $\{1, 2, \ldots, N-1\}$ and check whether $a^m \equiv 1 \pmod N$ or $a^{2^i \cdot m} \equiv -1 \pmod N$ for some $i \in \{0, 1, 2, \ldots, k-1\}$.
In case the above condition turns out to be true then $a^{N-1} \equiv 1 \pmod N$ and we declare that N is prime. Else we say that N is composite.

5.2 Pseudocode

Algorithm 3 Primality Testing

```
function IS_PRIME($a, N, m, k$)
    if $N = 2$ then
        return Prime
    end if
    if $N\%2 = 0$ then
        return Composite
    end if

    $x \leftarrow a^m \pmod N$
    if $x = 1$ then
        return Prime
    end if
    for $i = 1$ to $k$ do
        if $x = N - 1$ then
            return Prime
        end if
        $x \leftarrow x^2 \pmod N$
    end for
    return Composite
end function

function CHECK_PRIMALITY($N$)
    $k \leftarrow 0$
    $m \leftarrow N - 1$
    while $m\%2 = 0$ do
        $k \leftarrow k + 1$
        $m \leftarrow m/2$
    end while

    $t \leftarrow$ # of iterations
    for $i = 1$ to $t$ do
        $a \leftarrow$ Pick a random integer randomly uniformly from $[1, N - 1]$
        if IS_PRIME($a, N, m, k$) = Composite then
            return Composite
        end if
    end for
    return Prime
end function
```

5.3 Time Complexity Analysis

a and N are n bit-long numbers. Multiplying two r bit numbers takes $\mathcal{O}(r)$ operations [ignoring logarithmic factors] $\implies$ multiplying two n bit numbers takes $\mathcal{O}(n)$ operations and $k \leq \log_2 N$. Thus, one call to Is_Prime function takes $\mathcal{O}(n^2) = \mathcal{O}((\log N)^2)$ operations.
Hence, the time complexity of Check_Primality function is $\mathcal{O}(t \cdot n^2) = \mathcal{O}(t \cdot (\log N)^2)$

5.4 Error Analysis

We show that the result of one call to Is_Prime function has error with probability $\leq \frac{1}{2}$. Thereafter, we show that $P(\text{Check\_Primality function gives error}) \leq 2^{-t}$.

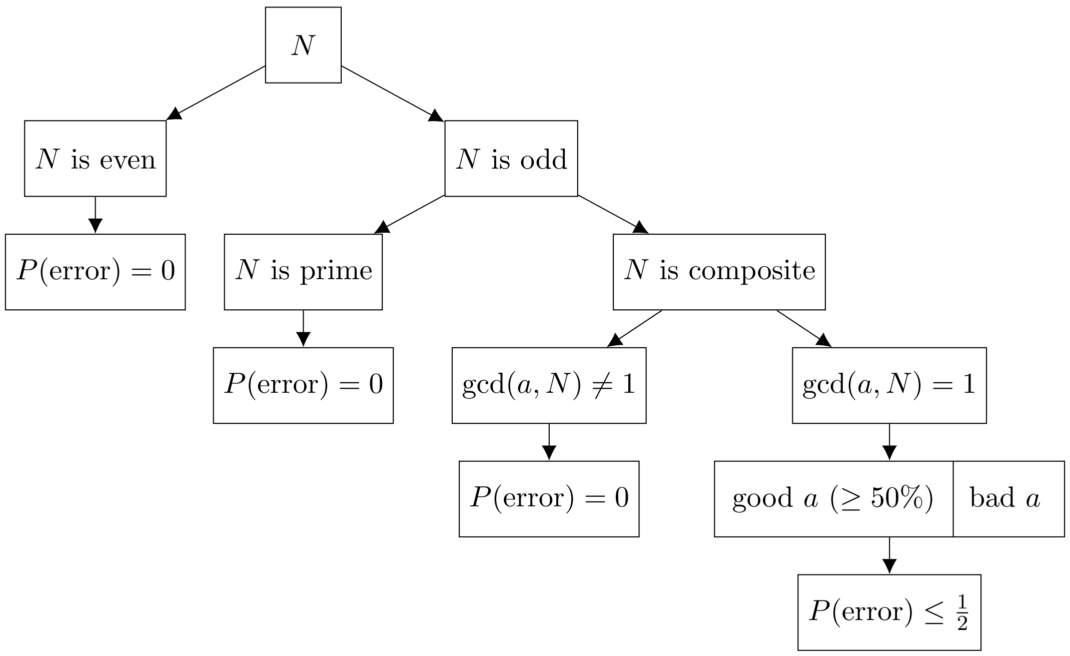


Figure 2: Overall proof idea to show $P(\text{Is\_Prime function fails}) \leq \frac{1}{2}$

If N is even, we output return prime if $N = 2$ and composite otherwise. Thus,

$$P(\text{error} \mid N \text{ is even}) = 0$$

As discussed in the proof idea, if N is **prime** then at least one of

$$(a^{2^{k-1}\cdot m} + 1), (a^{2^{k-2}\cdot m} + 1), \ldots, (a^{2^{1}\cdot m} + 1), (a^m + 1), (a^m - 1)$$

is divisible by $N \implies \forall a \in \{1, 2, \ldots, N-1\}$ either $a^m = 1 \pmod N$ or $a^{2^i \cdot m} = -1 \pmod N$ for some $i \in \{0, 1, 2, \ldots, k-1\}$. Thus, Is_Prime function always returns prime and

$$P(\text{error} \mid N \text{ is odd prime}) = 0$$

If N is **odd composite** and $\mathbf{gcd(a, N) \neq 1}$ then from Lemma 1, $a^{N-1} - 1$ is not divisible by N. So, none of

$$(a^{2^{k-1}\cdot m} + 1), (a^{2^{k-2}\cdot m} + 1), \ldots, (a^{2^{1}\cdot m} + 1), (a^m + 1), (a^m - 1)$$

is divisible by N. Thus, $a^m \not\equiv 1 \pmod N$ and $a^{2^i \cdot m} \not\equiv -1 \pmod N$ for all $i \in \{0, 1, 2, \ldots, k-1\}$. Hence, *Is_Prime* function returns composite and

$$P(\text{error} \mid N \text{ is odd composite and } \gcd(a, N) \neq 1) = 0$$

For the case when N is **odd composite** and $\mathbf{gcd(a, N) = 1}$, define a to be

- **good** if $a^m \not\equiv 1 \pmod N$ and $a^{2^i \cdot m} \not\equiv -1 \pmod N$ for all $i \in \{0, 1, 2, \ldots, k-1\}$
- **bad** otherwise, i.e. if $a^m = 1 \pmod N$ or $a^{2^i \cdot m} = -1 \pmod N$ for some $i \in \{0, 1, 2, \ldots, k-1\}$

Is_Prime function outputs prime if a is bad. We will show that fraction of bad $a \leq \frac{1}{2}$ and hence

$$P(\text{error} \mid N \text{ is odd composite and } \gcd(a, N) = 1) \leq \frac{1}{2}$$

Let N be an odd and composite number. Define the set of all a's that are invertible w.r.t. N

$$\mathcal{I} = \{a \mid a \in \{1, 2, 3, \ldots N-1\} \text{ and } \gcd(N, a) = 1\}$$

Let,

$$i_0 = \max_{i \in \{0, \ldots, k-1\}} (\exists\ a \in \{1, \ldots, N-1\} \mid a^{2^i} = -1 \pmod N)$$

and a_0 be the corresponding number such that $a_0^{2^{i_0}} = -1 \pmod N$
Such a_0 and i_0 always exists since $a_0 = N-1$ and $i_0 = 0$ satisfy the condition

$$(N-1)^{2^0} \equiv -1 \pmod N$$

Let,

$$\mathcal{B} = \{a \in \mathcal{I} \mid a \text{ is bad}\}$$

and

$$\mathcal{C} = \{a \in \mathcal{I} \mid a^{2^{i_0} \cdot m} = -1 \pmod N \text{ or } a^{2^{i_0} \cdot m} = 1 \pmod N\}$$

Lemma 3. *For the a_0 such that $a_0^{2^{i_0}} = -1 \pmod N$, gcd$(a_0, N) = 1$*

Proof. $a_0 \cdot a_0^{2^{i_0+1}-1} = a_0^{2^{i_0+1}} = (a_0^{2^{i_0}})^2 = (-1)^2 \pmod N = 1 \pmod N$
Thus, a_0 is invertible with respect to $N \implies \gcd(a_0, N) = 1$ □

Lemma 4. $\mathcal{B} \subseteq \mathcal{C}$

Proof. Consider a bad $a \in \mathcal{B}$.

- **Case 1:** $a^m = 1 \pmod N \implies a^{2^{i_0} \cdot m} = (a^m)^{2^{i_0}} = 1^{2^{i_0}} \pmod N = 1 \pmod N \implies a \in \mathcal{C}$
- **Case 2:** $a^{2^i \cdot m} = -1 \pmod N$ where $i < i_0$
$\implies a^{2^{i_0} \cdot m} = (a^{2^i \cdot m})^{2^{i_0 - i}} = (-1)^{2^{i_0 - i}} \pmod N = 1 \pmod N$ [since $i_0 - i \geq 1$] $\implies a \in \mathcal{C}$
- **Case 3:** $a^{2^i \cdot m} = -1 \pmod N$ where $i = i_0 \implies a^{2^{i_0} \cdot m} = -1 \pmod N \implies a \in \mathcal{C}$
- **Case 4:** $a^{2^i \cdot m} = -1 \pmod N$ where $i > i_0$
Let, $a_1 = a^m \pmod N$. Then, $a_1^{2^i} = a^{2^i \cdot m} \pmod N = -1 \pmod N$
i_0 is the max i for which $\exists\ x$ such that $x^{2^{i_0}} = -1 \pmod N \implies i_0 \geq i$. Thus, this is a contradiction and this case is not possible.

Hence we have successfully shown that the result holds for all possible cases. □

We will define a useful set of numbers as follows
Definition: A number $N > 1$ is called **carmichael** if N is composite and $\gcd(a, N) = 1 \implies a^{N-1} = 1 \pmod N$ for all $a \in \mathbb{Z}$ [1]

Lemma 5. *Any Carmichael number is square-free, i.e. it has no repeated prime factor.*

Proof. Let $N = p^t m$ where p and m are co-prime and $t \geq 2$ be a carmichael number
For any a such that $\gcd(a, N) = 1$, using the carmichael property

$$a^{N-1} \equiv 1 \pmod N$$

and hence $a^{N-1} \equiv 1 \pmod{p^t} \implies a^N \equiv a \pmod{p^2}$.

Using $a = 1 + p$, we get $(1+p)^n \equiv 1 + p \pmod{p^2}$. Now expanding $(1+p)^n$ using binomial theorem modulo p^2 we are left with $1 + Np$, since $p^2 | N$, we have

$$(1+p)^N \equiv 1 \pmod N$$

But this means $p \equiv 0 \pmod{p^2}$, this is always false and hence we have a contradiction. □

The following theorem will be useful in proving our claims

Theorem 2 (Chinese Remainder Theorem)**.** *If $x \equiv \alpha \pmod p$ and $x \equiv \beta \pmod q$ where p, q are coprime then x has a unique solution $\pmod{p \cdot q}$*

Lemma 6. *For all odd composite $N \in \mathbb{N}$, there always exists $b \in \mathcal{I} \backslash \mathcal{C}$*

Proof. We will proceed using seperate cases
Case 1: *N is a* ***carmichael*** *number.*

$$N = p_1 \cdot p_2 \dots p_l$$

where p_i's are distinct prime numbers and $l \geq 3$ [1]. Let,

$$N' = p_2 \cdot p_3 \dots p_l$$

Choose $b \in \{1, 2, 3, \dots, N-1\}$ such that

$$b = a_0 \pmod{p_1} \quad \text{and } b = 1 \pmod{N'}$$

Since $\gcd(p_1, N') = 1$, by *chinese remainder theorem*, there exists a unique b satisfying the above property.
From Lemma 3 and construction of b, $\gcd(b, N) = 1 \implies b \in \mathcal{I}$.

Now we will assume that $b \in \mathcal{C} \implies b^{2^{i_0} \cdot m} = 1 \pmod N$ or $b^{2^{i_0} \cdot m} = -1 \pmod N$, using this we will try to show a contradiction.

Case 1.1: $b^{2^{i_0} \cdot m} = 1 \pmod N$

$$\implies b^{2^{i_0} \cdot m} = \lambda \cdot N + 1$$

for some $\lambda \in \mathbb{Z}$

$$\implies b^{2^{i_0} \cdot m} = 1 \pmod{p_1}$$

Now,

$$a_0^{2^{i_0}} = -1 \pmod N \implies a_0^{2^{i_0}} = \delta \cdot N - 1$$

for some $\delta \in \mathbb{Z} \implies a_0^{2^{i_0}} = -1 \pmod{p_1}$
By construction of b, $b = a_0 \pmod{p_1} \implies b^{2^{i_0} \cdot m} = (a_0^{2^{i_0}})^m \pmod{p_1} = (-1)^m \pmod{p_1} = -1 \bmod p_1 \neq 1 \pmod{p_1}$. This is a contradiction.

Case 1.2: $b^{2^{i_0}\cdot m} = -1 \pmod{N}$

$$\implies b^{2^{i_0}\cdot m} = \mu \cdot N - 1$$

for some $\mu \in \mathbb{Z}$

$$\implies b^{2^{i_0}\cdot m} = -1 \pmod{N'}$$

But $b = 1 \pmod{N'} \implies b^{2^{i_0}\cdot m} = 1 \pmod{N'} \neq -1 \pmod{N'}$. This is a contradiction.
Since in both cases, we get a contradiction. Thus, our assumption is wrong and $b \notin \mathcal{C}$. Hence, the above $b \in \mathcal{I}\backslash\mathcal{C}$.

Case 2: *N is not a* ***carmichael*** *number*
By definition of carmichael numbers, $\exists\ a \in \mathbb{Z}$ such that

$$\gcd(a, N) = 1 \quad \text{but } a^{N-1} \neq 1 \pmod{N}$$

. Take $b = a \pmod{N}$.

$$\gcd(b, N) = \gcd(a \pmod{N}, N) = \gcd(a, N) = 1 \implies b \in \mathcal{I}$$

Now assume $b \in \mathcal{C} \implies b^{2^{i_0}\cdot m} = \pm 1 \pmod{N}$

$$\implies a^{N-1} = b^{N-1} \pmod{N} = (b^{2^{i_0}\cdot m})^{2^{k-i_0}} \pmod{N} = (\pm 1)^{2^{k-i_0}} \pmod{N} = 1 \pmod{N}$$

This is a contradiction since $a^{N-1} \neq 1 \pmod{N}$. Thus, our assumption is wrong and $b \notin \mathcal{C}$. Hence, the above $b \in \mathcal{I}\backslash\mathcal{C}$. □

Consider a $b \in \mathcal{I}\backslash\mathcal{C}$.

$$\implies \gcd(b, N) = 1 \text{ and } b^{2^{i_0}\cdot m} \neq \pm 1 \pmod{N}$$

Take a $a \in \mathcal{C}$.

$$\implies \gcd(a, N) = 1 \text{ and } a^{2^{i_0}\cdot m} = \pm 1 \pmod{N}$$

Let, $c = ab \pmod{N}$

$$\implies c^{2^{i_0}\cdot m} = (a^{2^{i_0}\cdot m}) \cdot (b^{2^{i_0}\cdot m}) = \pm(b^{2^{i_0}\cdot m}) \pmod{N} \neq \pm 1 \pmod{N}$$

$$\implies c \notin C$$

Since a and b both are co-prime with N

$$\gcd(c, N) = \gcd(ab \pmod{N}, N) = \gcd(ab, N) = 1$$

Thus, for each $a \in C$, $ab \pmod{N} \in \mathcal{I}\backslash\mathcal{C}$

Lemma 7. *For any $a_1, a_2(\neq a_1) \in \mathcal{C}$ and $b \in \mathcal{I}\backslash\mathcal{C}$, $a_1 b \pmod{N} \neq a_2 b \pmod{N}$.*

Proof. If possible, assume $a_1 b \pmod{N} = a_2 b \pmod{N} \implies a_1 b - a_2 b = 0 \pmod{N}$. Thus, N divides $|a_1 - a_2| \cdot b$.
But, $\gcd(b, N) = 1$ and $1 \leq |a_1 - a_2| \leq N - 2$. Hence, N does not divide $|a_1 - a_2| \cdot b$.
Thus, our assumption is wrong and $a_1 b \pmod{N} \neq a_2 b \pmod{N}$ □

Thus, each $a \in \mathcal{C}$ is maps to a unique $ab \pmod N \in \mathcal{I}\backslash\mathcal{C}$. Hence,

$$|\mathcal{C}| \leq |\mathcal{I}\backslash\mathcal{C}| = |\mathcal{I}| - |\mathcal{I} \cap \mathcal{C}| = |\mathcal{I}| - |\mathcal{C}| \implies |\mathcal{C}| \leq \frac{1}{2} \cdot |\mathcal{I}| \qquad [\text{ since } \mathcal{C} \subseteq \mathcal{I}] \tag{1}$$

$$\begin{aligned} P\,(\text{error} \mid N \text{ is odd composite and } \gcd(a,N)=1) &= P\,(a \text{ is bad} \mid N \text{ is odd composite and } \gcd(a,N)=1) \\ &= \frac{|\mathcal{B}|}{|\mathcal{I}|} \leq \frac{|\mathcal{C}|}{|\mathcal{I}|} \leq \frac{1}{2} \end{aligned}$$

[from Lemma 4 and (1)]

By partition theorem,

$$\begin{aligned} P(\text{error}) = {} & P(\text{error} \mid N \text{ is even}) \cdot P(N \text{ is even}) \\ & + P(\text{error} \mid N \text{ is odd prime}) \cdot P(N \text{ is odd prime}) \\ & + P(\text{error} \mid N \text{ is odd composite and } \gcd(a,N) \neq 1) \cdot P(N \text{ is odd composite and } \gcd(a,N) \neq 1) \\ & + P(\text{error} \mid N \text{ is odd composite and } \gcd(a,N) = 1) \cdot P(N \text{ is odd composite and } \gcd(a,N) = 1) \\ & = P(\text{error} \mid N \text{ is odd composite and } \gcd(a,N) = 1) \cdot P(N \text{ is odd composite and } \gcd(a,N) = 1) \\ & \leq P(\text{error} \mid N \text{ is odd composite and } \gcd(a,N) = 1) \leq \frac{1}{2} \end{aligned}$$

Check_Primality function gives wrong output only when N is composite and Is_Prime function returns prime in every iteration. Hence,

$$P(\text{Check\_Primality fails}) = (P(\text{error}))^t \leq 2^{-t}$$

6 Experimentation

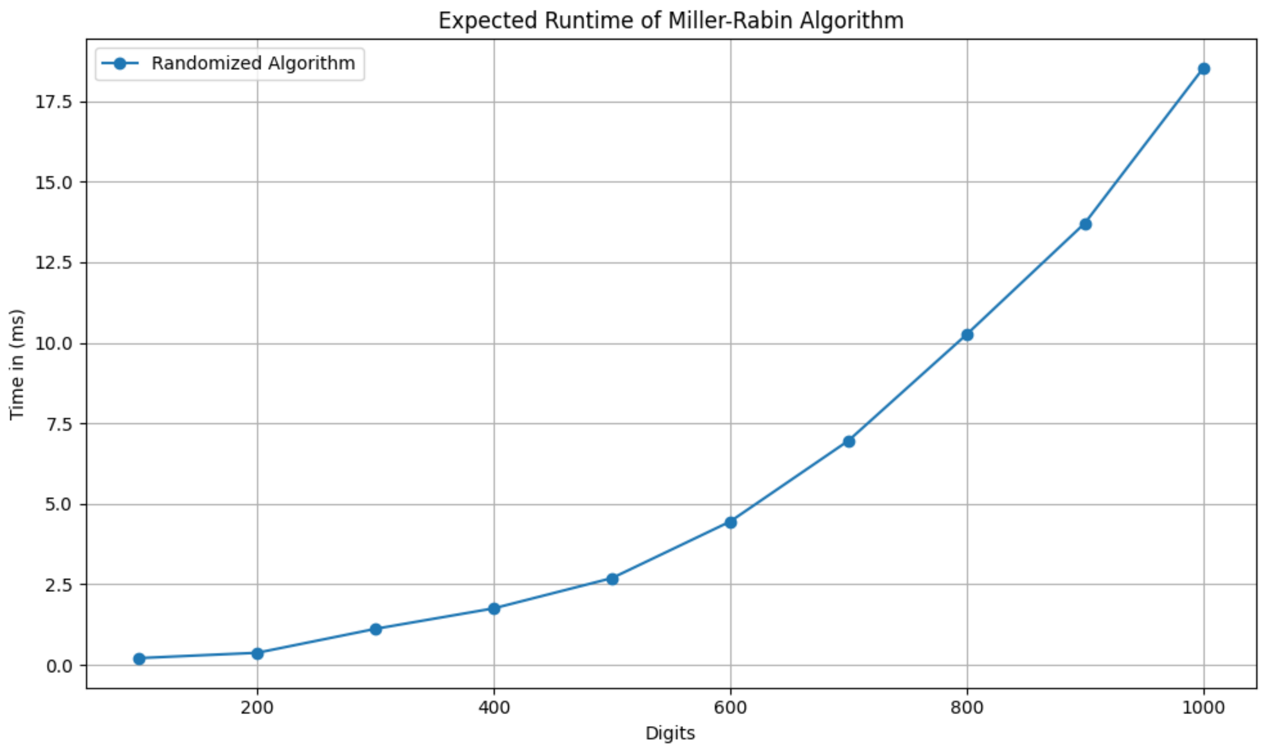
6.1 Verification of running time

Taking $t = \log n$, the worst case running time for a n bit number is $\mathcal{O}(n^2 \log n)$, we ran our algorithm of different sized numbers for verifying this we got the following results

Number of Digits	**Runtime (in ms)**
100	0.21375
200	0.378845
300	1.12211
400	1.75716
500	2.69595
600	4.4511
700	6.95912
800	10.2553
900	13.7075
1000	18.5088

Table 1: Running Time vs Input Size

As evident from the table and from the plot below the runtime is around what we expected $\mathcal{O}(n^2 \log n)$



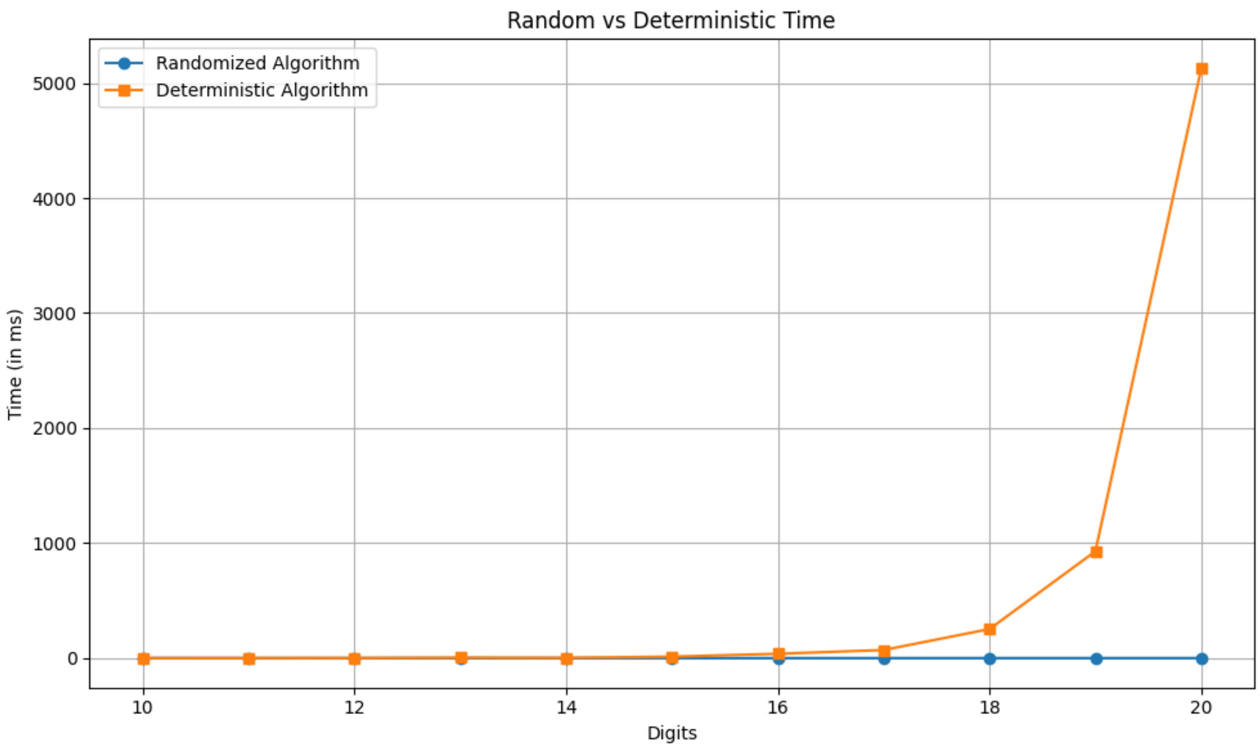
6.2 Comparison of Randomized and $\mathcal{O}(\sqrt{N})$ algorithm

We will compare our randomized algorithm with the trivial $\mathcal{O}(2^{n/2})$ algorithm (n is the input size). We will only use small n as obviously the time in the trivial algorithm increases extremely quickly with increase in n.

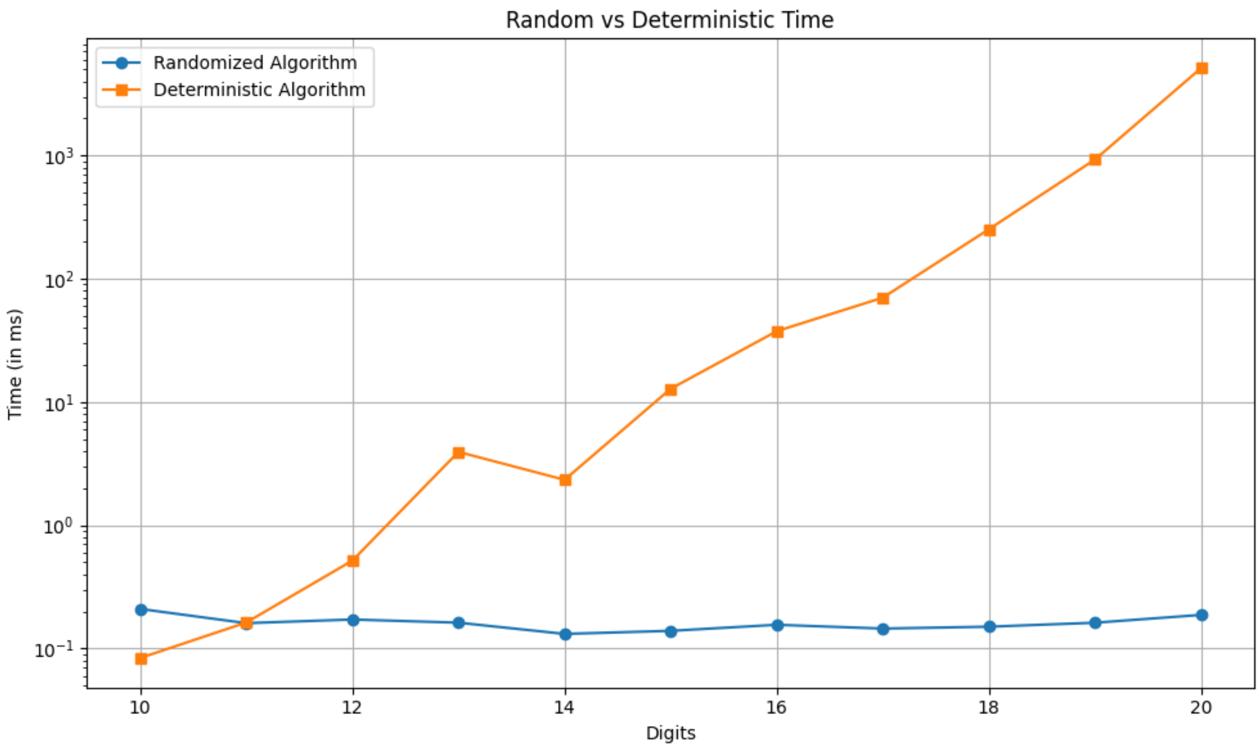
Number of Digits	Randomized Algo(in ms)	Deterministic Algo(in ms)
10	0.209326	0.083652
11	0.160492	0.162398
12	0.17181	0.516129
13	0.161881	3.93635
14	0.131291	2.33564
15	0.139025	12.8007
16	0.155669	37.5555
17	0.145033	70.2461
18	0.150207	252.387
19	0.161797	928.462
20	0.187234	5126.59

Table 2: Comparison of Running Time with Input Size

The following is a graph plotted on the log scale comparing the two algorithms, it is obvious just how much speedup we achieve.



We can even see the plot with a logarithmic axis to see just how much better our algorithm is



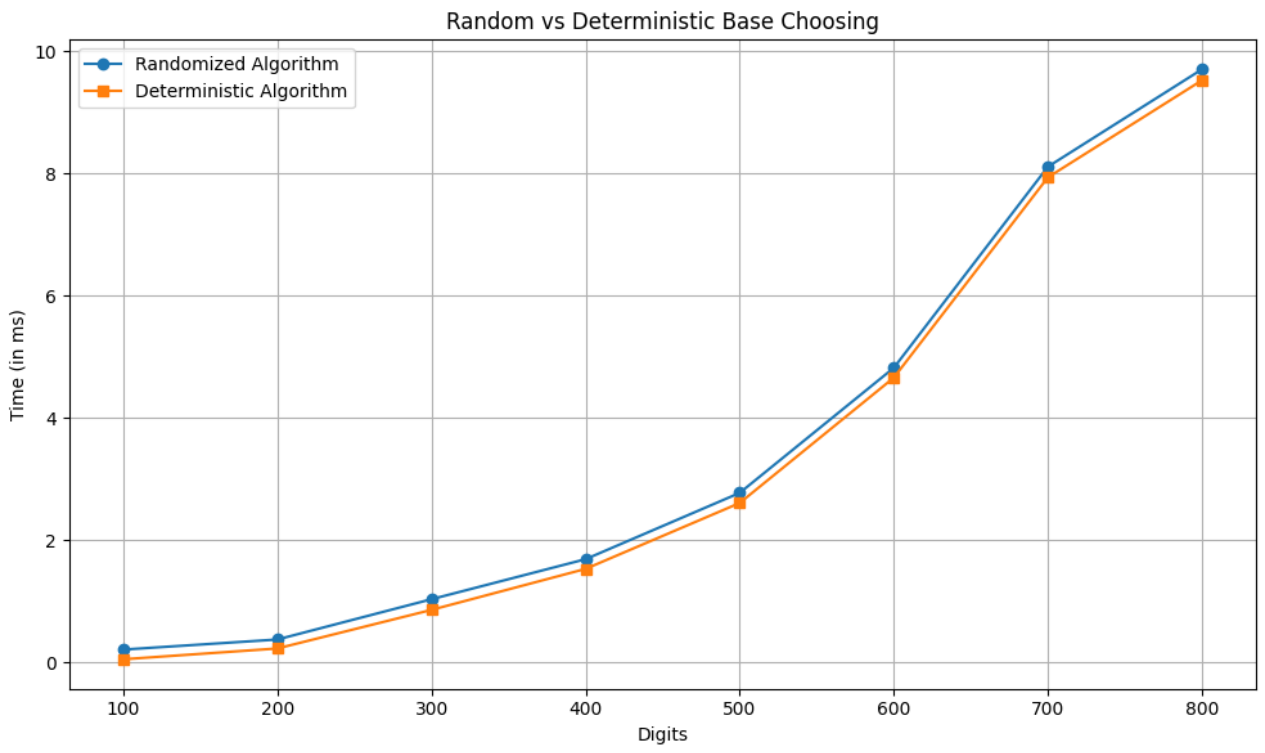
6.3 Usage of Randomization

Is randomization really important? What if we take the base sequentially from 2 to $1 + t$, and stop after for $t = \log n$. Taking the same maximum t for both cases we will compare the running time of both algorithms.

Number of Digits	Random Bases(in ms)	Sequential Bases(in ms)
100.0	0.207868	0.0494236
200.0	0.372822	0.225997
300.0	1.03214	0.8577
400.0	1.68758	1.52874
500.0	2.77264	2.60728
600.0	4.81948	4.6583
700.0	8.11095	7.93717
800.0	9.70584	9.52614

Table 3: Comparison of Running Time with Input Size based on bases selection

If we choose a composite N randomly uniformly, somewhat unexpectedly the sequential algorithm gives better running times that choosing the base randomly although they are really close. This can probably be exploited to create some heuristic for improving the running time further and can be used for future research.



We get this result as actually choosing the base randomly takes some time, and that is an almost constant difference as can be seen from the graph. We still prefer the randomized algorithm because the deterministic algorithm will always give errors for some fixed numbers might take a huge amount of time to run for some. The randomized algorithm might face the same issues but it will never face these issues for the same number in multiple runs so we can actually get guarantees on the probability for all numbers meanwhile the deterministic algorithm can not provide any such guarantees for all.

One such example is the following number

28871482380507712126714295971303939919776094592797227009265160241974323037991527
33116328983144639225941977803110929349655578418949441740933805615113979999421542
41693397290542371100275104208013496673175515285922696291677532547504444585610194
94042000399044321167766199496295392504526987193290703735640322737012784538991261
20309244841494728976885406024976768122077071687938121709811322297802059565867

Here, the randomized algorithm takes $0.0057s$ on average, but the deterministic algorithm takes $0.3063s$ which is almost 60 times higher.

6.4 Expected Number of iterations

6.4.1 Formal Analysis

We will first proceed to do a formal analysis of the expected number of iterations of the algorithm before betting a correct answer.

The expected number of iterations for a prime number will be equal to the maximum number (t) we decide, but for a composite number the number of runs can be modeled using a geometric random variable X which equals the number of times the algorithm repeats. We know that

$$P(X = x) = (1-p)^{x-1}p \quad p \text{ is the probability of given correct answer}$$

$$\implies E[X] = \frac{1}{p}$$

$$E[X] \leq 2 \quad \because p \geq \frac{1}{2}$$

6.4.2 Experimental Results

Now we will test this result, using experiments where we get the following data

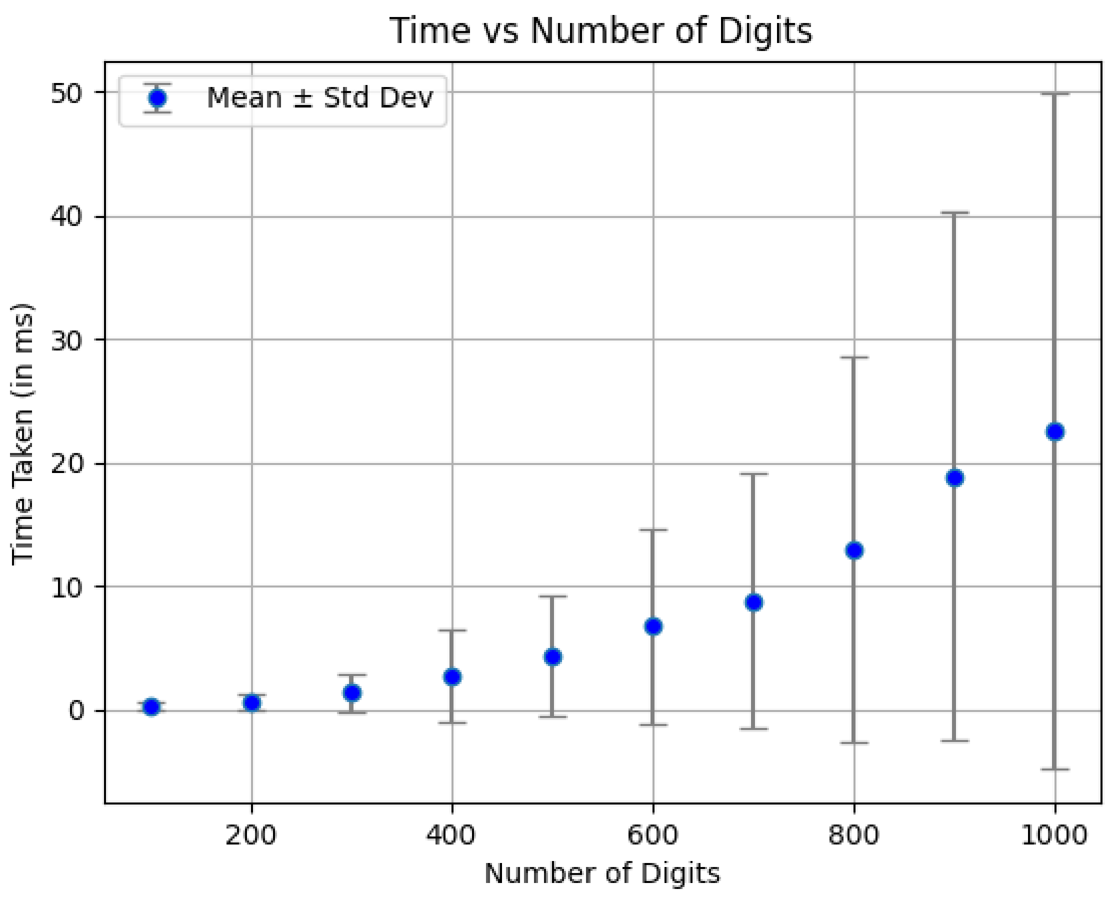
Number of Digits	**Expected # of Iterations**
100	1.00
200	1.00
300	1.00
400	1.00
500	1.00
600	1.00
700	1.00
800	1.00
900	1.00
1000	1.00

Table 4: Average Number of Iterations versus Input Size

It can be clearly seen that for almost all random inputs the number of iterations is 1, which shows the power of our algorithm.

6.5 Deviation from average

What about the deviation from the average time? What if some cases take much more time that the others, we will check this by finding out the deviation of the data

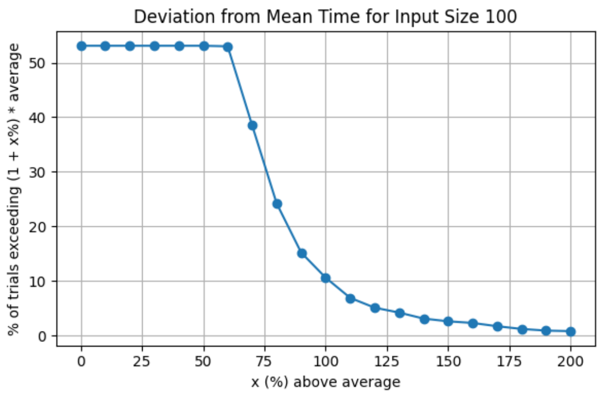


Clearly the deviation in increasing a lot for higher number of digits so to explore this further we will find the percentage of values exceeding the average case behavior by some particular fraction

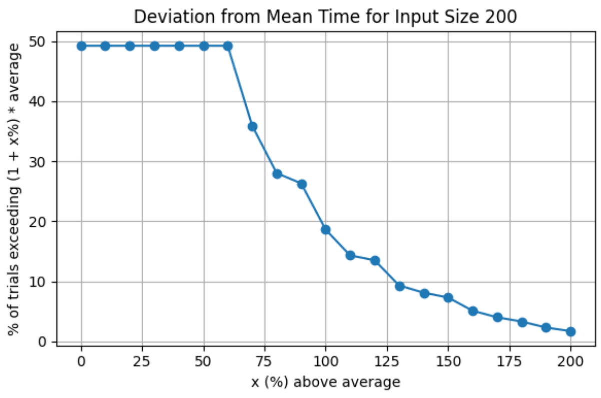
x	100	200	300	400	500	600	700	800	900	1000
0	53.1	49.2	50.0	53.1	50.7	50.8	50.3	47.5	50.7	50.4
10	53.1	49.2	50.0	53.1	50.7	50.8	50.3	47.5	50.7	50.4
20	53.1	49.2	50.0	53.1	50.7	42.9	50.3	47.5	50.1	50.4
30	53.1	49.2	50.0	53.1	50.7	41.4	50.3	41.4	42.1	50.4
40	53.1	49.2	50.0	38.8	43.9	40.8	40.5	36.7	37.1	40.6
50	53.1	49.2	33.7	29.5	30.9	40.6	29.2	35.3	36.7	30.5
60	53.0	49.2	29.7	28.2	30.1	27.7	27.1	34.8	36.3	26.2
70	38.5	35.9	28.3	27.2	29.6	24.2	26.9	30.4	25.5	26.0
80	24.2	28.0	26.2	21.6	29.2	22.9	26.9	24.5	22.6	24.8
90	15.2	26.3	18.9	14.8	17.6	21.2	17.6	21.2	20.9	20.4
100	10.6	18.6	17.2	13.5	16.1	19.8	15.4	20.3	19.2	16.2
110	6.9	14.3	16.4	13.1	15.7	14.8	14.8	19.2	16.9	14.5
120	5.1	13.5	14.4	10.9	15.5	12.5	14.8	18.0	12.6	14.2
130	4.2	9.3	10.8	7.7	10.2	11.2	11.8	14.1	10.8	12.0
140	3.1	8.1	9.5	6.9	8.2	10.9	8.8	12.4	10.2	11.5
150	2.6	7.3	8.7	6.8	7.9	10.2	8.3	11.3	9.4	8.9
160	2.3	5.1	7.1	5.7	7.8	7.7	7.8	9.9	9.4	8.2
170	1.7	4.0	6.2	4.0	6.9	6.5	7.7	9.4	8.8	7.2
180	1.2	3.3	5.5	3.7	5.6	5.7	5.7	8.5	6.2	5.8
190	0.9	2.3	4.9	3.6	5.2	5.4	4.8	6.7	5.3	5.6
200	0.8	1.7	4.3	3.0	5.1	4.5	4.6	5.3	4.8	4.4

Table 5: % of times the algorithm exceeds average by x% vs input size (n)

Across the different input sizes it is there is a significant deviation from the average case behavior this is something that we should improve our algorithm on and try to do some improvement so that the deviation reduces as this can actually impact our algorithm a lot.
Lets also plot this so that we have a visual understanding of this

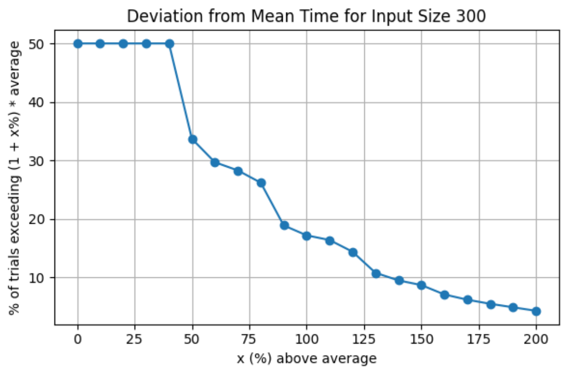


(a) Digits = 100

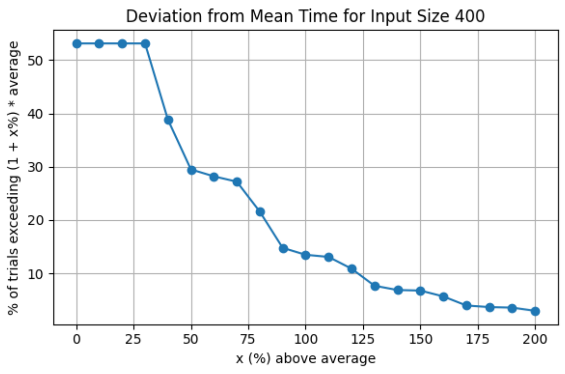


(b) Digits = 200

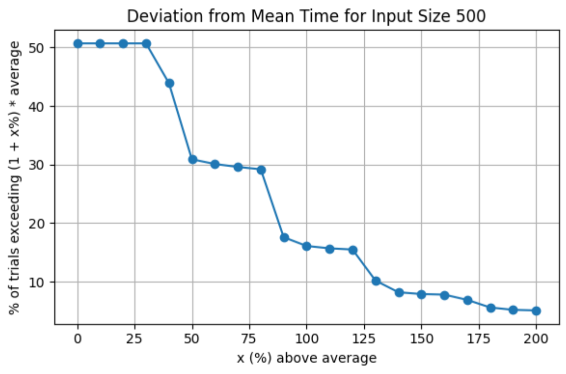
Figure 3: Deviation plots for various input sizes. Each shows the % of trials exceeding x% of the mean runtime.



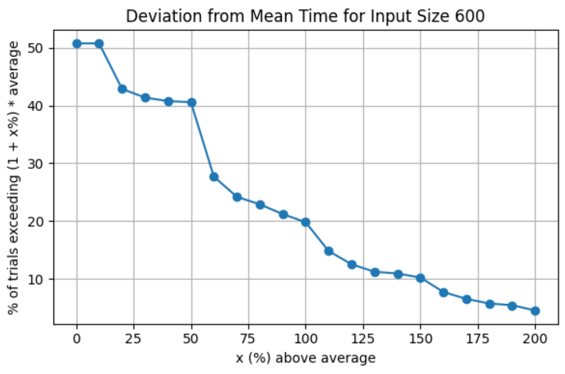
(a) Digits = 300



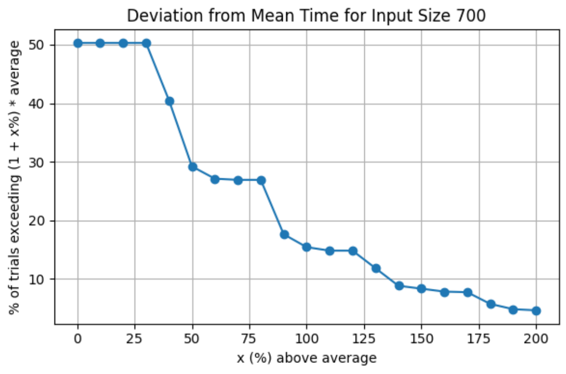
(b) Digits = 400



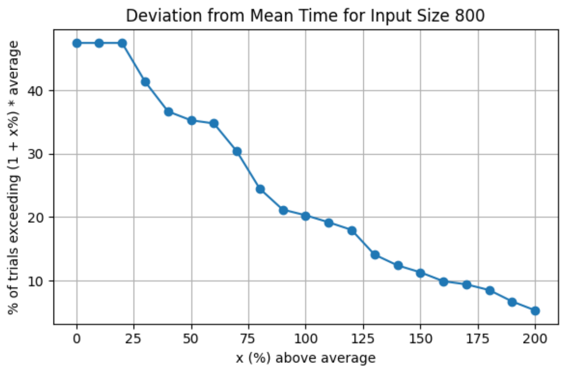
(c) Digits = 500



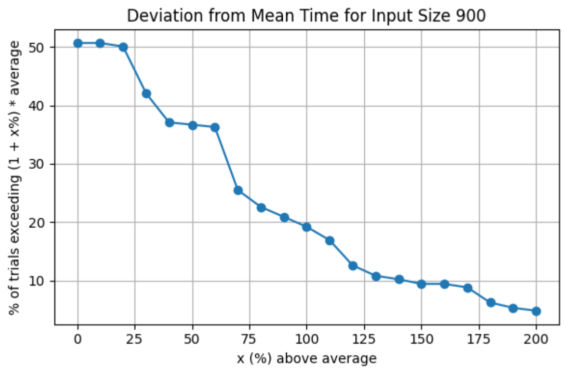
(d) Digits = 600



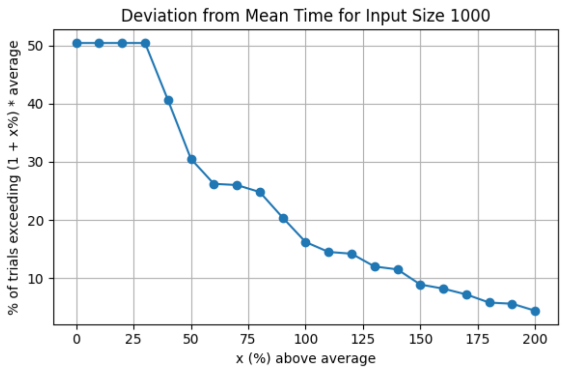
(e) Digits = 700



(f) Digits = 800



(g) Digits = 900



(h) Digits = 1000

Figure 4: Deviation plots for various input sizes. Each shows the % of trials exceeding $x\%$ of the mean runtime.

7 Conclusion and Future Work

So we can clearly see that the trivial algorithm is not practical for even numbers as small as 20 digits, even the *AKS Algorithm* would give time that is a cube of the current expected time by the randomized algorithm. For not too many iterations we can get a error probability that is even less than the chance of hardware failure on a modern computer. This shows the power of randomization

The algorithm still has scope for improvement by decreasing the running time with some heuristics and implementation improvements this will help in making it much better.

8 Acknowledgment

We express our gratitude to **Professor Surender Baswana** for his unwavering guidance and support throughout the duration of this project. His assistance significantly influenced the course of our journey, making it markedly distinct from what it would have been otherwise.

All the code used can be found in the following **Github Repository**

References

[1] Keith Conrad. Carmichael numbers and korselt's criterion. 2004. Available online at `https://kconrad.math.uconn.edu/blurbs/ugradnumthy/carmichaelkorselt.pdf`.