An Empirical Study of Deterministic and Non-Deterministic Primality Testing

Alex Feaser, David Golbari, Muneeb Khawaja
[CSCI 323-12]-Design and Analysis of Algorithms
Professor Lawrence Teitleman
Semester: Fall 2019

Abstract: This paper explores and contrasts the differences between various deterministic and non-deterministic algorithms for primality testing considering their resource overhead, run time, accuracy, and correctness. The deterministic algorithms under consideration include a Brute Force Approach, Wilson's Theorem and the Agrawal-Kayal-Saxena algorithm. Whereas, the non-deterministic set of algorithms consists of Fermat's Little Theorem and the Miller-Rabin algorithm. All algorithm implementations are iterative in nature. The study is developed using Java as the language of choice and the data used for testing consists of large arbitrary numbers managed with Java's BigInteger class.

For the **brute-force approach** to verifying primality, we use an optimized version of the algorithm that capitalizes on the fact that if there exists some number that divides the number n under consideration wholly, it must lie in $(1, \sqrt{n}]$. Consequently, the theoretical time complexity turns out to be $O(\sqrt{n})$. The pseudocode is as follows:

```
1.  Input: Number $n$.
2.  Loop: While integer i = 0 < $\sqrt{n}$:
        2.1.1.  Check: $n \ (mod\ i) \equiv 0$
            2.1.1.1.  If True: Increment $i$ and continue to the next iteration.
            2.1.1.2.  If False: n is composite, return False.
3.  return True
4.  Output: True if n is prime, False if n is composite.
```

The second Deterministic algorithm is by the English Mathematician John Wilson who discovered the algorithm in the 18th century. **Wilson's Theorem** states that if (p > 1) | 'p' is an integer, then p is a prime if and only if $[(p-1)! + 1] \ (mod\ p) \equiv 0$. This was proved by Lagrange in 1771. The theoretical time complexity is $O(n!)$as the factorial dominates the process. The pseudocode is as follows

```
1.  Input: Number n.
2.  return $[(n-1)! + 1] \ (mod\ n) \equiv 0$
3.  Output: True if n is prime, False if n is composite.
```

The **Agrawal-Kayal-Saxena primality test**, abbreviated as the AKS primality test, is a deterministic algorithm that can determine the primality of numbers within polynomial time. The algorithm is the first primality-proving algorithm to be simultaneously general, polynomial, deterministic, and unconditional. This means it can test any number in polynomial time and give a confident answer on any number without relying on other unproven hypothesis. The algorithm is based on a theorem that is a generalization of Fermat's little theorem. It states that given an integer, and some given integer that is coprime, then is prime if and only if the relation holds. The pseudocode is as follows:

```
1.  Input: Number $n$ > 1
2.  Edge Cases that ensure that n > 4 :
    2.1.  If (n ≤1) return false.
    2.2.  If (n ≤ 3) return false.
    2.3.  if(n % 2 == 0 or n % 3 == 0) return false
3.  Loop: i = 5
    3.1.  While i $^n$ < n
    3.2.  If (n % i == 0 or n % (i + 2) == 0 ) return false
4.  Return true, as whatever condition remains should hold to (x + a) $^n$≡
          (x $^n$+ a) (mod x $^r$ - 1, n)
5.  Output: True if n is prime, False if n is composite.
```

The first non-deterministic algorithm is based on **Fermat's Little Theorem,** which fits the Monte Carlo class of algorithms. The algorithm was first stated by Pierre de Fermat in 1640. The theorem states that if the number p is a prime number, then $a^p - a$, where a is any number, is an integer multiple of p. This leads to Fermat's

Primality Test which states that if a is any number not divisible by p and if p is a prime number, then:$a^{p-1} \equiv 1 \,(mod\, p)$. The theoretical time complexity turns out to be $O(k \times log^2(n))$ | k = number of trials, n = number under consideration, with the assumptions that exponentiation and multiplication are optimized. The pseudocode is as follows:

```
1.   Input: Number $n$ and number of trials $k$.
2.   Edge Cases that ensure that n > 4 :
     2.1.  If n ≤1 or If n ≡4 i.e. when n is 4, 1 or negative return false.
     2.2.  If n ≤ 3 i.e. when n is 3 or 2 return true.
3.   Loop:While k trials > 0 :
             Generate random $a$ between $[1, n-1]$
             Check: $a^{n-1}\ mod\ n \equiv 1$
             3.1.1.1.  If True:  Decrement k and continue to next iteration.
             3.1.1.2.  If False:  n is composite, return False.
4.   return True as n is probably a prime.
5.   Output: True if n is prime, False if n is composite.
```

The second non-deterministic algorithm, also belonging to the Monte Carlo family of algorithms, is the **Miller-Rabin algorithm**. This algorithm was rediscovered by Miller in 1976 and was originally deterministic in nature. Rabin introduced probability which leads to the algorithm under consideration in this study. The algorithm is similar to Fermat's Primality Test in terms of implementation but involves more pre-processing in comparison. The theoretical time complexity for this algorithm turns out to be $O(k \times log^3(n))$ | k = number of trials, n = number. The pseudocode is as follows:

```
1.   Input: Number $n$ and number of trials $k$.
2.   Edge Cases that ensure that n > 4 :
     2.1.  If n ≤1 or If n ≡4 i.e. when n is 4, 1 or negative return false.
     2.2.  If n ≤ 3 i.e. when n is 3 or 2 return true.
3.   Set $d \rightarrow n - 1$
4.   Loop: While $d$ (mod 2) is equal to 0:
     4.1.  Update $d \rightarrow \frac{d}{2}$
```

```
5.  Loop: While k trials > 0 :
        5.1.1.  Check: Perform Miller test on d and n.
            5.1.1.1.  If True: Decrement k and continue to next iteration.
            5.1.1.2.  If False:  n is composite, return False.
6.  return True as n is probably a prime.
7.  Output: True if n is prime, False if n is composite.
```

Miller Test Subroutine Pseudocode:

```
1.  Input: $n$ and $d$.
2.  Generate random $a$ between $[2, n-1]$
3.  Update $a \rightarrow (a\ (mod\ d)\ )^n$
4.  Check: If $a$ is equal to 1 or ($n$-1)
    4.1.  If True, return True
    4.2.  If false, continue to the next line.
5.  Loop: While $d$ is not equal to ($n$ -1):
        5.1.1.  Update $a \rightarrow (a \times a)\ (mod\ n)$
        5.1.2.  Update $d \rightarrow (d \times 2)$
        5.1.3.  Check: If $a$ is equal to 1
            5.1.3.1.  If True, return false.
            5.1.3.2.  If False, continue to the next line.
        5.1.4.  Check: If $a$ is equal to ($n$-1)
            5.1.4.1.  If True, return True.
            5.1.4.2.  If False, continue to the next iteration.
6.  return True as n is probably a prime.
7.  Output: True if n is prime, False if n is composite.
```

The numbers under consideration are generated using Java's **BigInteger** Class's consturctor that returns a random BigInteger upto a specified upper bound. Big integer numbers lie between: $-2^{Integer.MAX\_\ VALUE} < Numbers < 2^{Integer.MAX.VALUE}$

The constructor we are utilizing accepts a bit length and an object of type Random, and returns a BigInteger in the range $0 \leq n \leq 2^{bitLength} - 1$ with uniform distribution. After generating this number, we use the BigInteger Class's

nextProbablePrime() function to generate the next probable prime. This is done to ensure that the non-deterministic algorithms do all operations and are not cut short. Moreover, for the non-deterministic algorithms, the number of trials k is dependent on the number of bits in the number n and is approximated as half the bitlength of n i.e. $k \simeq \frac{log_2 n}{2}$.

We execute each of our tests in succession on each randomly generated probable prime, timing and tracking the number of operations for each test. and take the average of 10 executions per algorithm for a given bit length and a given random number.

The charts generated by our study are as follows:

Run Time Comparison:

Decimal Digits	**Bit Length.** ***n***	**Num Trials** ***k*** *(if NDA)*	**Brute Force**	**Wilson's Theorem**	**Aggarwal-Kayal-Saxena**	**Fermat's Primality**	**Miller-Rabin Primality**
5	**16**	**8**	0.000390	0.790445	0.000153	0.000251	0.000387
10	**32**	**16**	0.019565	-	0.007574	0.000160	0.000254
20	**64**	**32**	-	-	-	0.000602	0.000867
39	**128**	**64**	-	-	-	0.001702	0.001706
78	**256**	**128**	-	-	-	0.011339	0.012559

Theoretical Operation Comparison:

Decimal Digits	**Bit Length.** ***n***	**Num Trials** ***k*** *(if NDA)*	**Brute Force** $O(\sqrt{n})$	**Wilson's Theorem** $O(n!)$	**Aggarwal-Kayal-Saxena** $O(log\ \ (n)^6)$	**Fermat's Primality** $O(k.log^2n)$	**Miller-Rabin Primality** $O(k.log^3n)$
5	**16**	**8**	*256*	(65,536)!	29	185	893
10	**32**	**16**	65,536	(4,294,967,296)!	58	1484	14302
20	**64**	**32**	4,294,967,296	(8,446,744,073,709,551,616)!	114	11877	228833
39	**128**	**64**	8,446,744,073,709,551,616	(340,282,366,920,938,463,463,374,607,431,768,211,456)!	231	95020	3661332
78	**256**	**128**	340,282,366,920,938,463,463,374,607,431,768,211,456	(115,792,089,237,316,195,423,570,985,008,687,907,853,269,984,665,640,564,039,457,584,00	462	760167	58581323

				7,913,129, 639,936)!			

Actual Operation Comparison:

Decimal Digits	**Bit Length.** n	**Num Trials** k *(if NDA)*	**Brute Force** $O(\sqrt{n})$	**Wilson's Theorem** $O(n!)$	**Aggarwal-Kayal-Saxena** $O(log\ \ (n)^6)$	**Fermat's Primality** $O(k.log^2n)$	**Miller-Rabin Primality** $O(k.log^3n)$
5	**16**	**8**	624	85902	120	70	107
10	**32**	**16**	169730	-	28317	143	235
20	**64**	**32**	-	-	-	287	578
39	**128**	**64**	-	-	-	578	804
78	**256**	**128**	-	-	-	1153	2042

The predicted operation counts do not match our actual counts for a number of reasons. Firstly, the absolute worst case is considered for the theoretical count. This doesn't necessarily hold up in practice as the numbers are generated randomly and span a wide range of numbers. Secondly, since we do not count every single operation and consider only expensive operations, a discrepancy could arise from that.

Based on our results, for very very large numbers, it seems prudent to use the random algorithms with the square root witness optimization as it is much faster and generally very reliable. In fact, Java's own implementation of nextProbablePrime() is based on a version of the Miller - Rabin Primality algorithm.

If we had more time, we would look at other practical optimizations and explore what limitations are caused because of hardware limitations. In addition, we could also improve our studying by looking at the distribution of random numbers tested and by increasing the number of trials and bits under consideration.

Sources

“AKS Primality Test.” *Wikipedia*, Wikimedia Foundation, 14 Nov. 2019, en.wikipedia.org/wiki/AKS_primality_test#History_and_running_time.

“Fermat Primality Test.” *Wikipedia*, Wikimedia Foundation, 20 Sept. 2019, en.wikipedia.org/wiki/Fermat_primality_test#Complexity.

“Fermat's Little Theorem.” *Wikipedia*, Wikimedia Foundation, 5 Dec. 2019, en.wikipedia.org/wiki/Fermat%27s_little_theorem.

“Miller–Rabin Primality Test.” *Wikipedia*, Wikimedia Foundation, 5 Nov. 2019, en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test#Complexity.

“Power of Two.” *Wikipedia*, Wikimedia Foundation, 3 Dec. 2019, en.wikipedia.org/wiki/Power_of_two.

“Primality Test: Set 2 (Fermat Method).” *GeeksforGeeks*, 3 May 2019, www.geeksforgeeks.org/primality-test-set-2-fermet-method/.

“Primality Test: Set 3 (Miller–Rabin).” *GeeksforGeeks*, 26 Feb. 2019, www.geeksforgeeks.org/primality-test-set-3-miller-rabin.

Priy, Surya. “AKS Primality Test.” *GeeksforGeeks*, 10 Dec. 2018, www.geeksforgeeks.org/aks-primality-test/.

user 10496 user 10496, et al. “Trial Divisions before Miller-Rabin Checks?” *Cryptography Stack Exchange*, 1 July 1964, crypto.stackexchange.com/questions/17707/trial-divisions-before-miller-rabin-checks.

“Wilson's Theorem.” *Https://Artofproblemsolving.com*, artofproblemsolving.com/wiki/index.php/Wilson%27s_Theorem.

“Wilson's Theorem.” *Wikipedia*, Wikimedia Foundation, 10 Dec. 2019, en.wikipedia.org/wiki/Wilson%27s_theorem#cite_note-4.

Zijing Wu ZijingWu 20122 silver badges33 bronze badges, et al. "Why Miller–Rabin Instead of Fermat Primality Test?" *Computer Science Stack Exchange*, 1 Mar. 1964, cs.stackexchange.com/questions/21462/why-miller-rabin-instead-of-fermat-primality-test.

jdk8/jdk8/Jdk: 687fd7c7986d Src/Share/Classes/Java/Math/BigInteger.java, hg.openjdk.java.net/jdk8/jdk8/jdk/file/tip/src/share/classes/java/math/BigInteger.java.