An Empirical Study of Three Primality Algorithms
James Murphy & Guolong Luo
Team stayHome
CSCI 323 - Analysis of Algorithms
Semester: Spring 2020

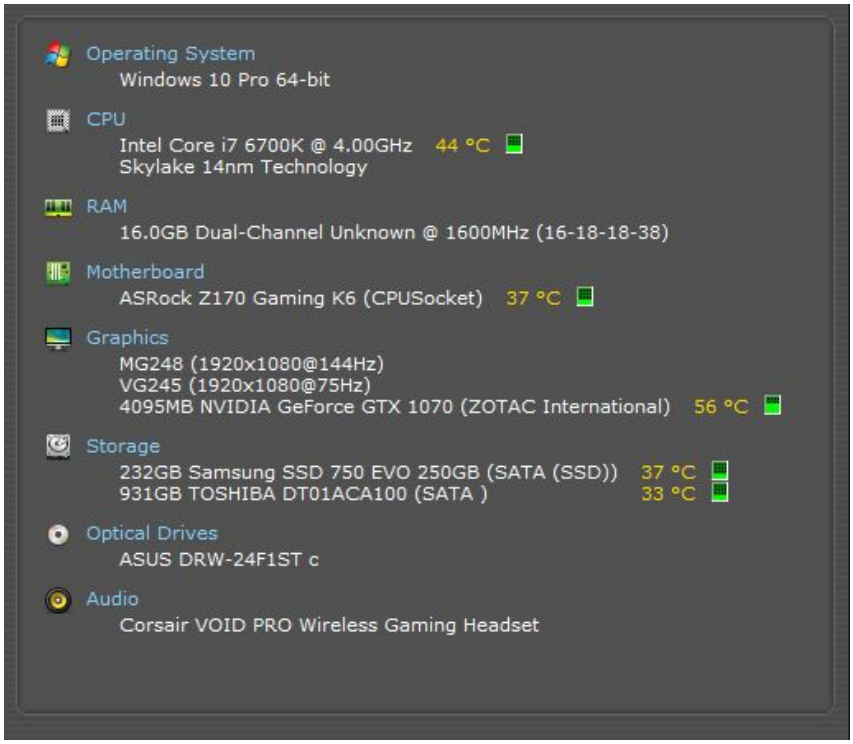
Introduction

A short paragraph describing what you sought to do in this project - list the three (or however many) algorithms you were comparing, what you compared them for, etc.

In this project we attempted to track how many operations (comparisons) each algorithm took on various datasets to find out how it compares to their run times. The 3 algorithms used were Brute Force, Sieve Of Eratosthenes and Fermat's Little Theorem. The data sets used were a randomly generated array of 10000 integers, 10000 randomly generated integers * 2 to only produce even numbers and an array of the first 10000 prime numbers that were generated by running the Sieve Of Eratosthenes algorithm on the number 104730 since the 10000th prime number is 104729.

Implementation Considerations

For this project we used Eclipse and wrote the program in Java. The machine that executed the code used in the tables was on Windows 10 and its specs are :



The algorithm was defined inside an object and was given its dataset through the creation of the object.

The algorithms were modified to keep track of comparisons in the following ways :

Brute Force : In the Brute Force algorithm, the algorithm iterates in a for loop from 2 to n-1. It checks if n % i == 0. This comparison is tracked. On an even integer, it'll be only one comparison since n % 2 is 0. If it's an odd number, it has to iterate through up to n to try and find a number that n is divisible by that isn't n and 1. This algorithm also keeps track of the corner case, if the number n <= 1, return false.
Implementation time complexity : O(n).

Sieve Of Eratosthenes : In the Sieve Of Eratosthenes algorithm, the algorithm iterates in a loop from p and counts up in increments of p. (p*p) Then, it marks each of these numbers greater than p in the bool array. Then, find the first number greater than p in the list that is not marked. If there wasn't a number, stop. If there was, let p equal this number and repeat the process again. When the algorithm terminates all the numbers in the list that aren't marked are prime numbers. The comparisons in the implementation is made when the algorithm checks if the integer is marked.
Implementation time complexity :O(n*log(log(n))).

Fermat's Little Theorem : In Fermat's Little Theorem, the comparisons are tracked in a few different ways. One, on the corner cases : if n <= 1 or if n == 4, increment the comparisons and return false since those are not prime numbers. If n <= 3, return true since 2 and 3 are both prime numbers and increment comparisons. Otherwise, the algorithm has to test for primality and run k times. The while loop checks if k > 0 and when it does, increment comparisons. Next the algorithm checks if the power function for FLT is not 1, comparisons are incremented there as well. In the power function there is a while loop that checks while n > 0, so comparisons are incremented. Lastly, the algorithm checks if n is odd and if it is it changes the result. On this comparison to see if it's odd, comparisons is incremented.
Implementation time complexity : O(klogn).

Data Considerations

Data Set 1 : A data set of 10000 randomly generated integers using the java.util.Random library

Data Set 2 : A data set of 10000 randomly generated integers using the java.util.Random library and multiplied by 2 to create a data set of all even numbers.

Data Set 3 : A dataset of the first 10000 prime numbers. Since one of the algorithms being used in this project is Sieve of Eratosthenes it is used to generate our first 10000 primes. The 10000th prime number is 104729, so to completely fill the array we need to run the algorithm on the number 104730 since that's the whole purpose of using this algorithm.

Algorithm Overview + Tracking of Operations

Brute Force

Overview : The Brute Force algorithm takes the input number n and in a for loop from 2 to n - 1, n is taken modulo i. If n % i == 0, then n is divisible by another number that is not 1 and itself. This then makes it not prime and return false. Otherwise, it is a prime number and return true. 1 and n are excluded due to those being the only factors a prime number has.

Tracking of Operations :

Size (n)	Random Integers	Random Even Integers	First 10000 Prime Numbers
10	134	19	119
100	76321	199	24033
1000	597357	1999	3681913
10000	5847631	19999	496155411

Time Taken :

Size (n)	Random Integers (Nanoseconds)	Random Even Integers (Nanoseconds)	First 10000 Prime Numbers (Nanoseconds)
10	4800	37900	1000
100	1015800	11600	125100
1000	2968800	44800	9074300
10000	14418500	312300	1135205600

Sieve Of Eratosthenes :

Overview : This is an ancient algorithm used to track and generate prime numbers. It is useful and has utility that the brute force algorithm doesn't. It doesn't just check a specific prime number, it also creates a table of all prime numbers in the range up to the inputted number n. This algorithm was used to generate our array of the first 10000 primes for a data set. It creates a list of consecutive integers from 2 to n, and then lets p start as 2. (The first prime number.) Then, starting from p, count up to n in increments of p and make each multiple of itself in the list (Ex : 2 : Mark 2, 4, 6, …. n * 2). Then, find the first number greater than p in the list that is not marked. (Ex : after 2 make p = 3) If there wasn't one, stop. Otherwise, p is now equal to that number (the next prime) and the whole process is repeated.

Tracking of Operations :

Size (n)	Random Integers	Random Even Integers	First 10000 Prime Numbers
10	159	254	15
100	1820	2467	569
1000	18357	24800	16166
10000	184364	243659	455284

Time Taken :

Size (n)	Random Integers (Nanoseconds)	Random Even Integers (Nanoseconds)	First 10000 Prime Numbers (Nanoseconds)
10	929800	153700	1400
100	2312000	1352600	26900
1000	8588700	13945700	3346600
10000	66123800	93962500	553446700

Fermat's Little Theorem :

Overview : This implementation is a probabilistic function. This means that the algorithm approximates the result. Accuracy comes with more time and work assigned to the function. For prime numbers, the algorithm always produces a correct result. The accuracy of deciding whether a number is composite or not is what comes into question. The more times you run the algorithm, the better it is at deciding whether or not if a number is composite. The accuracy of the algorithm increases by how many k times you run and repeat the following steps :

1. Pick a random integer in range [2, n - 2]
2. If the gcd of (a, n) != 1, return false.
3. If a^{n-1} != 1 (mod n), return false.

Otherwise, the algorithm returns true.

3 Tests Case to test accuracy and work required

Take the integer being tested and test 3 cases

1. N / 5 -> K = N / 5, run the algorithm = 20% of the integer N
    - Worst for correct composite results
2. N / 2 -> K = N / 2, run the algorithm = 50% of the integer N
    - Average ~ Okay for composite results
3. N -> K = N, run the algorithm 100% of the Integer N
    - Best result possible, requires the most work.

K = arr[i] / 5, or n / 5 times. (Worse for approximation)

Tracking of Operations :

Size (n)	Random Integers	Random Even Integers	First 10000 Prime Numbers
10	192	211	167
100	339524	2277	67560
1000	2511361	22945	14673568
10000	23999292	228514	609110991

Time Taken :

Size (n)	Random Integers (Nanoseconds)	Random Even Integers (Nanoseconds)	First 10000 Prime Numbers (Nanoseconds)
10	87900	29800	2300
100	3023800	20300	520900
1000	17698600	210900	103521000
10000	170766800	1723400	4169466000

K = arr[i] / 2, or n / 2 times. (Average for approximation)

Tracking of Operations :

Size (n)	Random Integers	Random Even Integers	First 10000 Prime Numbers
10	173780	217	460
100	914995	2303	169843
1000	6679077	22759	36697819
10000	62627376	228092	1522674452

Time Taken :

Size (n)	Random Integers (Nanoseconds)	Random Even Integers (Nanoseconds)	First 10000 Prime Numbers (Nanoseconds)
10	1764200	2200	36500
100	7937600	16500	1467400
1000	47442000	199000	258165800
10000	447974100	1888400	10364537300

K = arr[i], or n times. (Best for approximation)

Tracking of Operations :

Size (n)	Random Integers	Random Even Integers	First 10000 Prime Numbers
10	64120	238	976
100	1291353	2275	340936
1000	13648040	22877	73414162
10000	124227004	228174	1249628014

Time Taken :

Size (n)	Random Integers (Nanoseconds)	Random Even Integers (Nanoseconds)	First 10000 Prime Numbers (Nanoseconds)
10	1330000	3700	10000
100	10389200	20900	2628300
1000	98873200	213200	518516700
10000	883372000	1679600	21170025200

Conclusion / Analysis :

Pre Thought : The 3 datasets were chosen meticulously. A mix of prime numbers, even numbers and all prime numbers would show in the case of :

- Prime Numbers : The worst possible run time since the checks in these algorithms are checking for divisibility of other numbers.
- Even Numbers : Once the number 2 or number that is multiple of 2 is checked it's almost immediately easily detectable that the number is composite
- Mix of Randomly Generated Even & Odd Numbers : Saying the average case is incorrect because it's possible to generate all even and all odd numbers, however it is the inbetween of the 2 other extremes. It's the "average," approximation / representation of numbers. There is a very high probability that both even and odd numbers are going to be in the data set.

Operations :

Brute Force had the least amount of operations for Random Numbers and Even Numbers. This is the expected output. Once an even number n hits n % 2 on its first iteration, the algorithm is completed on one comparison. This is huge in terms of operations. However with prime numbers it has to climb and check all the way to n -1 which means it's going to have to make n - 2 operations for each number. The Sieve Of Eratosthenes algorithm has to do much less comparisons than Brute Force on all datasets besides even numbers. This is because of the way the algorithm marks numbers and creates a table of prime numbers. The comparisons are derived from checking which numbers are marked. Since it's just counting up in multiples it doesn't have to do many comparisons until it's checking which numbers are actually prime. Fermat's Little Theorem makes a lot of comparisons, but it can also either do a lot more work or a lot less work depending on the implementation. For ⅛ of k runs, it still does A LOT more iterations than the previous 2 algorithms, but that's because it's running ⅛ of k on each element in the dataset and that number goes up based on how many more k times you want to check for approximation. It has to do much more work than the previous 2 algorithms. The algorithm doesn't have a problem in telling you if a number is a prime, it's an issue of finding composite numbers. Better accuracy comes with more work. I took the algorithm and ran it 3 times to compare the total amount of work it had to do and time it took to do all that work on a dataset. It has to do either ⅛ k, ½ k or K runs * dataset size. This gets worse the larger the number is.

Time Analysis :

The Time Analysis for each dataset was wildly different. Fermat's Little Theorem by far took the most time and depending on how many iterations the algorithm ran for, the worse it got which was expected. For 10000 prime numbers on k iterations, it took 21170025200 nanoseconds which is roughly 21 seconds. That's a long time. Compared to the Sieve Algorithm which took .51 seconds and Brute Force which took 1.14 seconds, that's a huge increase in time. This also falls in lines with the operations.

Each input number has to be run either k, k/2 or k/5 times depending on how well the needs for approximation are. Brute Force took the least amount of time for each dataset besides all prime numbers, which makes sense since it's an O(n) algorithm. The Brute Force Algorithm on a prime number is basically the worst case scenario of the algorithm. Sieve took longer than Brute Force in the randomly generated integers and even integers time as well. This is because of the n % 2 check. The algorithm may take longer on prime numbers, but the time saved on instantly finding an even number is huge. The small time in nanoseconds between the datasets is negligible for a 10,000 element dataset. This could scale more depending on how big the dataset is. The Sieve algorithm provides utility that brute force doesn't. If you store all the prime numbers you collect in a data structure like a hashmap, the Sieve algorithm is definitely a better choice. Yes the time to calculate a number may be shorter with the brute force approach, but the lookup time of a hashmap is O(1), and since you don't have to recalculate and can instantly find a prime number in your set that's smaller than a previous iteration it will over the long run save a lot of time. There's a valid argument for which algorithm to use of the 3, and it depends on the use case.

Sources :

Brute Force: https://www.geeksforgeeks.org/primality-test-set-1-introduction-and-school-method/?ref=lbp

Sieve of Eratosthenes: https://www.tutorialspoint.com/Sieve-of-Eratosthenes-in-java

Fermat's Little Theorem: https://www.geeksforgeeks.org/primality-test-set-2-fermet-method/?ref=lbp