PrimeAlgorithms Document

February 21, 2024

Primetest Algorithm Summary

This document presents the performance analysis of three algorithms implemented in Java and Python. They are all Boolean that check if a given integer is prime or not.

Performance Results

The following table summarizes the time required to complete the algorithms for given integers, measured in nanoseconds.

N	primetest1	primetest2	primetest3
1013	200	200	100
1017	100	100	200
10037	400	100	100
10012	100	200	100
99991	300	200	200
100001	200	200	100

Table 1: Performance analysis of prime-testing algorithms in Java.

N	primetest1	primetest2	primetest3
1013	100135.80	58889.39	5722.05
1017	81062.32	52213.67	4053.12
10037	660181.05	55789.95	7867.81
10012	713109.97	62942.50	3814.70
99991	7125854.49	32186.51	19788.74
100001	7472753.52	34093.86	5722.05

Table 2: Performance analysis of prime-testing algorithms in Python.

Summary

In general, it seems that Java is the faster language in terms of computing prime numbers as Java is a static typing language while Python is a statically typed language.

Algorithm 1 is the slowest as the algorithm has to iterate through all "n" values of the given number to test.

Algorithm 2 is the second slowest as it the range of values calculated is lesser, sqrt(n). This algorithm utilizes the relation that by sqrt-ing the number still calculates the same factors while doing less computations. While primetest2 was faster than primetest1, it still iterated through all possible n values.

Algorithm 3 was the quickest in general as it was the same as algorithm 2 but had a break statement. The statement allowed the algorithm to stop as soon as it reached a conclusion thus doing less computation.

Generally, looking at the graphs underneath, the runtime to check if a number is prime depended on its quantity. The higher the number, the slower the runtime and vice versa. Some graphs behaved differently, but likely due to other software environments and conditions.

However, it is worth noting that prime numbers took longer to run from the primetests than composite numbers. That's because in all 3 algorithms there is a for loop to check if there were any divisors. 10012 is divisible by 2, and 100001 is divisible by 11. When they are discovered to be divisible by a prime number other than the number itself the function comes to a stop. This is especially true in algorithm 3 because of break as it immediately stops the function.

Graph

The third algorithm from Python best visually displays the contrast in runtime within different values of n.

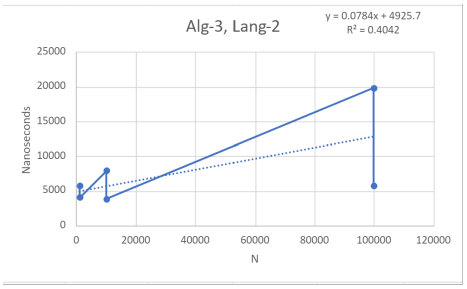

Figure 1: Runtime of Primetest3 on Python

PrimeList Algorithm Summary

Another part of this document is it presents the performance analysis of four primelist algorithms in Java specifically to see which one has the fastest runtime. Primelist returns all prime numbers between 2 and the input number itself.

Performance Results

The following table summarizes the time required to complete the algorithms for given integers, measured in seconds instead.

Table 3: Time required to complete the algorithms (in seconds)

N	primelist 1	primelist 2	primelist 3	primelist 4
1,000	0.0015553	0.000421699	0.000247301	0.0008518
10,000	0.0756627	0.0048326	0.001661	0.0054766
100,000	7.081404299	0.1139968	0.0046891	0.065951301
1,000,000	696.8002679	3.5125859	0.0990696	4.4120957
10,000,000	67577.268112001	107.3008532	2.3680974	308.4644303

Summary

Algorithm 1 had a runtime of $O(n^2)$ because of the 2 nested for loops. It checks every number from 2 to n - 1 to determine if n is prime. For each number up to n, it performs n calculations. Basically every number of assigned integer j is checked until n-1 to see if it's divisible, which is highly inefficient.

Algorithm 2 had a runtime of $O(n\sqrt{n})$. There are two nested for loops except the inner loop ranges from 2 to the square root of n. This algorithm is able to work despite a shorter for loop as it checks the smaller factors as checking the larger factors are unnecessary because it links up with each corresponding smaller factor.

Algorithm 3 has the same runtime as algorithm 2 at $O(n\sqrt{n})$. The break gives an early exit upon finding that the number j within n has a divisor. That is able to save time and avoid unnecessary checks.

Algorithm 4 has a fluctuating runtime. There exists a nested for loop, but the inner loop range increases if the arraylist of prime numbers increases. It also benefits from an early termination upon finding a divisor. Algorithm 3 is faster as it has the least number of operations. Algorithm 4 in the long run is slower than algorithm 2 because the inner loop size eventually surpasses sqrt(j). However, algorithm 4 emphasizes that most composite numbers are divisible by 2 or 3. The reason the time it takes to check if a number is composite is faster is because the loop will reach a divisor before it ends.