**Design and Analysis of Algorithms to Find Prime Numbers**

Seth Futrell,Dakota Perryman,

Matthew Ritchie, Mark Thompson

*University of North Carolina,Wilmington*

**Abstract**

The computational complexity of prime number analysis, although an incredibly researched area, will continue to be a highly interesting field of study as computational power continues to double every year until Moore’s law is proven obsolete. Since prime numbers have an increasing impact of our everyday lives, with fields like encryption relying heavily on primes,as growing scientists that we continue the tradition of exploring prime numbers and their irregular behaviors. Through our research we implemented five algorithms to study the analysis of both time and space in respect to creating a list of prime numbers under a bound. Although analyzing these algorithms with Big O notation is essential, we found it necessary to also implement these algorithms in multiple programming languages and do direct analysis to see the correlation that may exist. Our goal for this project is to do in depth research as to why each algorithm behaves the way it does, and if there are irregularities, point out as to why they are there.

Key Words: Sieve, Prime, Big O, Algorithm,

**Introduction**

Prime numbers, numbers that can only be divided by one and its self, have fascinated inquisitive minds throughout human history. The earliest surviving works on prime numbers date back to circa 300 B.C with the Greek Mathematician Euclid’s *Elements* (1). Even with over 2,000 years of research on the subject there are still many questions that mathematicians are still trying to find answers to, famous examples of this are the Twin Primes Theorem which states that there exists infinitely many primes with only one composite number between and Goldbach’s conjecture which says that every even number larger than two can be written as a sum of two primes, even with these questions unsolved there is still a lot we know about prime numbers and some of these facts will be important throughout this discussion. First and foremost, the set of prime numbers is infinite. This was first proven by Euclid the proof is as follows:

*Suppose that p1=2 < p2 = 3 < ... < pr are all of the primes. Let P = p1p2...pr+1 and let p be a prime dividing P; then p can not be any of p1, p2, ..., pr, otherwise p would divide the difference P-p1p2...pr=1, which is impossible. So this prime p is still another prime, and p1, p2, ..., pr would not be all of the primes. [1]*

**Constraints**

Armed with the knowledge that the set of primes has no end, it allowed for a better understanding of the size of the problem,knowing that theoretically our search for primes could outlast the earth and even the Milky Way galaxy itself, the decision was made for reasons related to computing power and time of computation to keep the search space of numbers restricted to that of the limitations of a 32 bit system, thus allowing for a search space of 0 to 2,147,483,647.

**Problem Statement**

In simple terms our problem was to create a set of all the prime numbers less than a given number n. To begin formally stating our problem we must first define a set of constructs.

Define set S as: Given a counting number n define set S as all counting number < n.

Define a prime number as : p∈N, ∀k∈N,((k∣p)⇒(k=1)∨(k=p)) (2)

Define set P as: A set of positive integers where each element is prime.

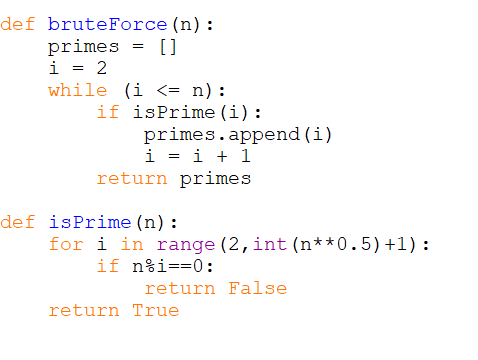
Formally stated, our problem can be defined as:

∀p∈S, IFF((p∈N, ∀k∈N,((k∣p)⇒(k=1)∨(k=p)) : (P← P U {p})

**Bench Marks**

For this project a brute force implementation was chosen to be the benchmark comparison for the other algorithms used. The algorithm used follows the given steps:

For every counting number from 2 to n divide that number by every number less than its square root and see if it divides with a remainder of zero. A Python implementation of this algorithm can be seen below.



**Big O analysis**

Considering the while loop has a time complexity O(n) and the loop for isPrime can be computed as O(√n) and since isPrime loop is nested inside the while loop we can multiply the complexities thus giving us a total complexity of O(n√n).

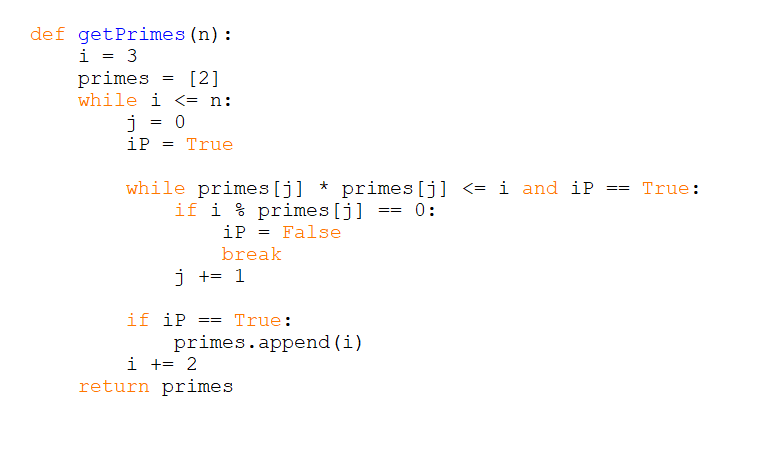
**The Algorithms**

**Hour Algorithm- Original Implementation**

**Algorithm**

The Hour Algorithm is an improvement on the brute force implementation. First, create a list of primes containing only the number 2, starting at the number 3 check if the number is divisible by any number in the current prime list it is not prime, since all numbers can be written as a product of primes, if it is not divisible add it to the prime list increase by two and repeat until the range of desire values has been reached.

**Python Implementation and Big O analysis**

****

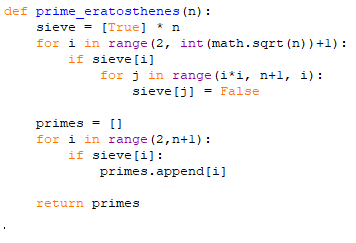
The first loop will give us a time complexity of O(n) and since the second loop will be looped through while i is less than jth element of the primes list squared and since you also have bigger likelihood to break out of the loop with the if statement it can be viewed as a O(log(n)) giving a total complexity of O(nlog(n)).

**Sieve of Eratosthenes**

**Algorithm**

The Sieve of Eratosthenes, attributed to Eratosthenes of Cyrene, can be dated back to ancient Greece (4). This method for finding prime numbers takes the strategy of creating a list of consecutive counting numbers from 2 to a given number N. Then starting with 2 take all its multiples up to the square root of N (Since a number greater than the square root can be written as a multiple of a number below N), marking each as a non prime and continue with this process with each number. Once all the numbers have been completed the numbers which have not been marked as composite are the set of prime numbers less than a given N.

**Big O analysis**

****

Using this example we can see that the first loop will go up to the square root of (n) meaning it will give a time complexity of O(√(n)) the next loop will only loop through in steps of i giving a time complexity of O(Log(n)). This can be derived from the fact that if you loop from a list of [1,2,3,4] by 2 for example you will hit 1 and 3 meaning you hit 2 elements in the list or in other terms log2(4) which would be 2 matching the number of elements hit in the list, and since our chances of going into the second loop also decrease logarithmically we must also must take into account the last loop that loops through the whole list which is O(N) and this will give us a total time complexity of O(√(n)log(log(n))+O(N)) and since we can take the bigger of the two when adding with BigO this will give us O((n)log(log(n))).

**Sieve of Sundaram**

**Algorithm**

Developed in 1934 by Indian Mathematician S.P. Sundaram. The Sieve of Sundaram is reminiscent of Eratosthenes but differs in its approach as given a number *n* it will find all the primes 2*n*+2. Starting with a list of integers 1 to n, remove all the numbers that can be formulated from the inequality

*i*+*j*+2*ij≤*n*.* Where i, j ⋲ N, 1 *≤ i ≤ j.* Once all the numbers have the possible numbers have been removed, multiple the remaining by 2 and add 1.

**Mathematical Justification**

Let *q* be an odd number written as 2*k*+1, *q* cannot be prime if *k* can be written in the form *i*+*j*+2*ij.*

This is because:

2(*i*+*j*+2*ij)*+1

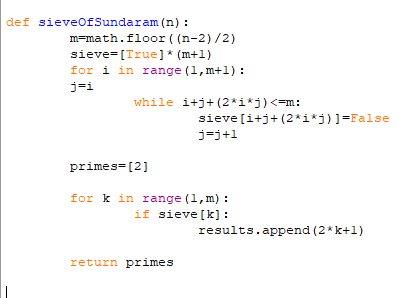
2*i*+2*j+*4*ij* +1

(2*i*+1)(2*j*+1)

Meaning q is a composite number comprised of 2 odd factors. (7)

**Big O analysis**

Below is our Python implementation of the Sieve of Sundaram.

****

The first for loop is O(m) with m being the desired limit of the prime numbers minus and divided by two. The while loop is O(log(m)) because j is being set to i each time during the for loop and thus cutting the amount of combinations that can be held for the while loop. Thus resulting in a final time complexity of O(nlog(n)).

**Sieve of Atkin**

**Algorithm**

The sieve of Atkin was created in 2003, by A.O.L. Atkin and Daniel J. Bernstein. Along with Sundaram, it is similar to Eratosthenes. First, a sieve list of n elements and a results list with {2, 3, 5} are created. For all values of x and y less than the square root of n, three different values of z are generated by three different binary quadratic functions. Each value of z is then divided by 60, if the remainder is an element in a predetermined set, specific to each function, the entry in the sieve list corresponding to z is flipped (prime becomes non-prime, and vice versa). The last step is to find every entry in the sieve list marked prime and mark all entries for the square of that number and the multiples of the square as non-prime. It is important to note that in the implementation of this algorithm, it is possible and more efficient to use 12 for the remainder division, as this reduces the set of numbers for each function by a factor of 4.

**Mathematical Justification**

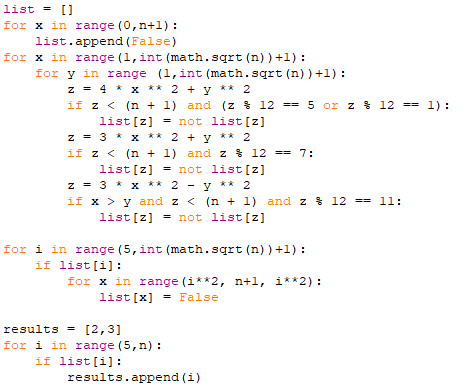
Whereas the sieve of Eratosthenes enumerates values of reducible binary form *xy*, the strategy of Atkin is to enumerate values of specific irreducible binary quadratic forms. Atkin and Bernstein give this example in their paper:

a square free positive integer p ∈ 1+4Z is prime if and only if the equation

4x2 + y2 = p has an odd number of positive solutions (x,y). There are only O(N) pairs(x,y) such that 4x2 + y2 ≤ N.

Atkin and Bernstein mention that it is possible to vary the functions and the (x,y) restraints, as well as noting that at the time of the paper, an optimal combination had not been found.

**Big O Analysis**



Big O = O(sqrt n \* sqrt n + sqrt n \* log log n + n)

= O(n + sqrt n log log n + n)

= O(n + sqrt n log log n)

= O(n log log n)

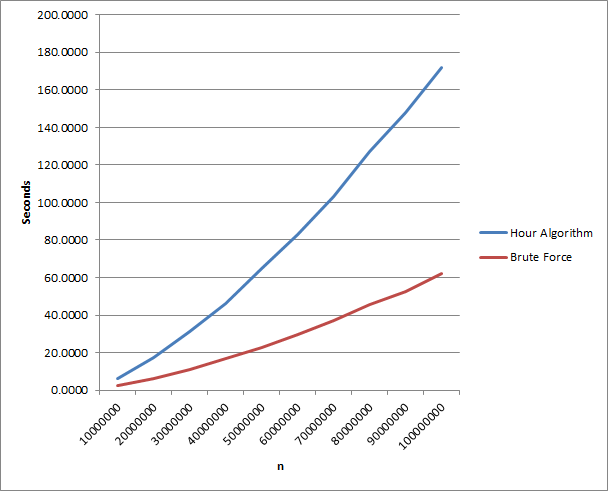
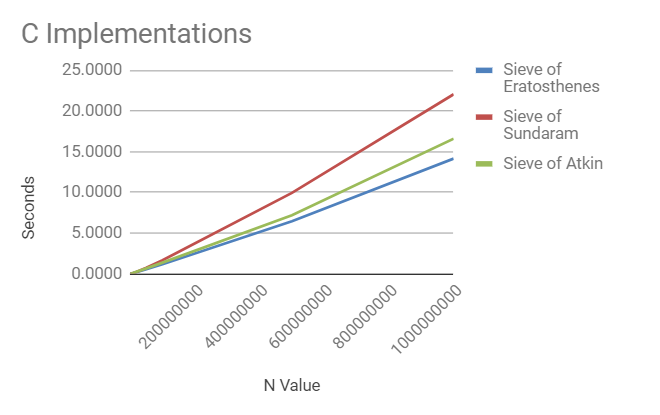
**Data Analysis**

All algorithms were implemented in both C and Python, which allowed us to do multiple Analyses of the algorithms based upon their design and the programming languages they were implemented in. The data we collected generally corresponded with our expectations of the algorithms with the exception of a few cases. The sieve of Eratosthenes proved to be the fastest, finding all the primes under a billion in an average of 14.190 seconds when implemented in C. Atkin’s proved to be the second fastest algorithm for large integers behind Eratosthenes, followed by Sundaram, Brute Force, and Hour respectively. Interestingly there does prove to be some range of numbers for which Sundaram, does outperform Atkins and Eratosthenes. This seems to contradict the complexity measurements. We speculate that this occurred due to the spatial distribution of primes in the region of 5 million to 15 million, sundaram.

Essential to the process of implementation in the C programming language, we had to explicitly declare the size of the array of prime numbers less than n. Since there is so formula for deciding exactly how many primes are less than a certain integer, we had to use a mathematical proof which sets an upper bound for how many primes there may be under an integer. If we let π(n) be the function that counts all primes less than n, the prime number theorem states that n / log n is a good approximation to π(n), because the limit of the quotient of the two functions π(n) and n / log n as n approaches infinity is 1. This is known as the asymptotic law of distribution of prime numbers, which tells us that the relative error of this approximation approaches 0 as n increases without bound (8). The formula we used to determine how many elements would be in the array is 1.25506 \* ( n / ln n), where n is the integer we are finding all primes below. For large values of n, π(n) will consistently be very close to, but never exceed, the result of this function (9). During our analysis we found that when n equals one billion, less than one percent of this allocated space was unused.

Below is a data sheet that corresponds to the comparison of algorithms implemented in C. The data shows Eratosthenes continuously outperforms all algorithms, except in the one case were the distribution of primes is larger. Here one can also recognize from the data that Brute Force out performed Hour Algorithm. This was unexpected since both algorithms have the same complexity and Hour Algorithm is doing less operations, but we believe this data could be explained by the fact that Hour has to access and array element for every check of a prime number this increases the time it takes to actually calculate if a number is prime.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| n | Sieve of Eratosthenes | Sieve of Sundaram | Sieve of Atkin | Hour Algorithm | Brute Force |
| 100000 | 0.0003 | 0.0004 | 0.0005 | 0.0103 | 0.0042 |
| 500000 | 0.0019 | 0.0022 | 0.0023 | 0.0929 | 0.0357 |
| 1000000 | 0.0035 | 0.0047 | 0.0045 | 0.2436 | 0.0937 |
| 5000000 | 0.0202 | 0.0247 | 0.0307 | 2.4019 | 0.8802 |
| 10000000 | 0.0716 | 0.0540 | 0.0932 | 6.4122 | 2.3250 |
| 50000000 | 0.5386 | 0.7240 | 0.6497 | 63.7648 | 22.9797 |
| 100000000 | 1.1584 | 1.6416 | 1.3502 | 172.1864 | 60.8557 |
| 500000000 | 6.4388 | 9.9291 | 7.1783 | 877.9912 | 306.5961 |
| 1000000000 | 14.1490 | 22.0586 | 16.5934 | 4896.3190 | 1714.8030 |

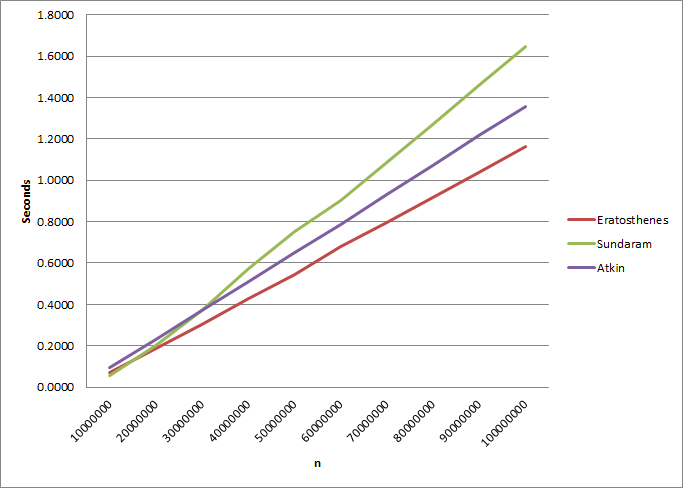
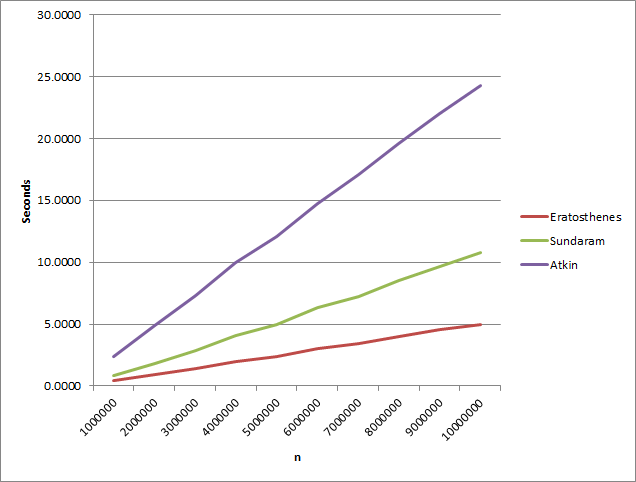


**Python Implementation**

Although it is crucial to examine how the implemented algorithms should react under constrained circumstances, another large aspect of our project was analyzing as to why the implementations performed so much better in the C programming language compared to python. Originally we decided to implement the algorithms in python since the language is more suited to handle large integers and the language is easier to convert mathematical proofs into code. But, as our research continued we realized that because Python is an interpreted language, we would not be able to run the algorithms on N values large enough to map out the differences in the data. We began to run into memory issues when attempting implementation of two of the sieves since we were creating sized N arrays and python would not allow us to create arrays that large. This was a major flaw that set us back since we would only be able to test the algorithms to a certain value where cpu processes may be able to largely affect our data.

Below is the data of the Python implementations compared to C for all five algorithms. These charts allow us to visually represent the scale of integers that python and C were able to handle. As visualized below, one can see how much more efficiently the C language performs under these circumstances as compared to Python.

Python Implementations C Implementations



**Conclusion and Further Research**

After conducting our research and processing our results, the five implementations we introduced brought specific advantages and disadvantages to the project. Although some of the implementations were just used as benchmarks and others were used to optimize time, it is clear that this problem of constructing a set of prime numbers under an integer will continue to fascinate the theoretical community. If we were able to consult with other researchers on their data in respect to their algorithms, we would like to compare and do more in depth research as to why the sieve of Sundaram and Hour Algorithm performed the way they did. In regards to future work, with the introduction of quantum computing and increasing computing power, the relevance of prime numbers and their uses for encryption, hash functions, and number theory continue to expand exponentially.

**Questions**

1.What is the big O when we are doing brute force to find the number of primes when we are only doing modulus division of sqrt(n)?

1. n\*sqrt(n)

2.What is the standard big O of sieve of Eratosthenes?

1. n\*log(log(n))

3.What is an ulam spiral?

1. A graphical depiction of the set of all prime numbers.

4.How would you construct 24 as multiple of prime factors?

1. 3\*2\*2\*2

5.What causes our implementations to be run much faster in C rather than in python?

1. Python is an interpreted language compared to C which is compiled directly by the CPU.

**Sources**

1. <http://primes.utm.edu/notes/proofs/infinite/euclids.html>
2. <http://mathworld.wolfram.com/PrimeNumber.html>
3. <http://www.research.ibm.com/ibm-q/learn/what-is-quantum-computing/>
4. <https://www.geeksforgeeks.org/sieve-of-eratosthenes/>
5. <http://www.dtc.umn.edu/~odlyzko/doc/arch/analytic.pi.of.x.pdf>
6. <http://number.computation.free.fr/Constants/Primes/countingPrimes.html>
7. <http://banach.millersville.edu/~bob/math478/History/Sundaram.html>
8. <https://en.wikipedia.org/wiki/Prime-counting_function>
9. <https://projecteuclid.org/download/pdf_1/euclid.ijm/1255631807>