**2.** Test your method to ensure that it works correctly. Print out or attach the list of primes generated by your program for =100.

**A:** To verify the correctness of my Sieve of Eratosthenes implementation, I ran the algorithm for `n=100`. The output was:

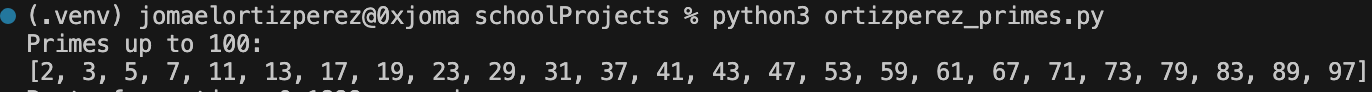


Figure 1: This list correctly represents all prime numbers up to 100.

**3.** Add code that will call your new method along with the brute force method and measure the time taken for both algorithms to generate for =200,000. Comment on the results you see. (e.g., Which is greater or smaller? Why does one take more time than the other? How might this scale if we increased *n* significantly, say to 10 million?)

**A:** I implemented timing code for both the Sieve of Eratosthenes and the brute force method for `n=200,000` The results were:

A black background with white text

Description automatically generated

Figure 2: These results clearly demonstrate the superior efficiency of the Sieve of Eratosthenes algorithm.

**Answer continued:** The implementation of Figure 2 completed the task approximately 17.4 times faster than the brute force method.

The significant time difference can be attributed to the fundamental approaches of these algorithms:

1. The brute force method checks each number individually for primality, leading to a time complexity of `O(n√n)`. For each number, it potentially checks divisibility up to its square root.
2. The Sieve of Eratosthenes, on the other hand, eliminates composite numbers in batches. Its time complexity is `O(n log log n)`, which is much more efficient for large `n`.

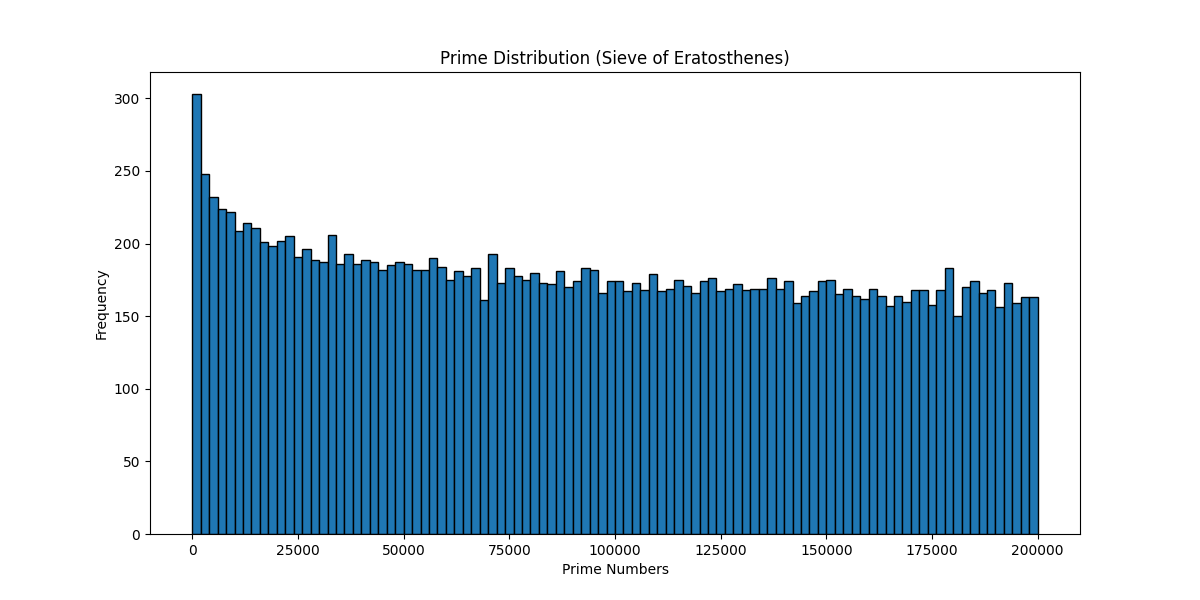
If we were to increase `n` to 10 million, the performance gap would widen dramatically:

1. The brute force method's runtime would increase substantially, potentially taking several minutes or even hours.
2. The Sieve of Eratosthenes would likely complete in a matter of seconds, though its memory usage would increase significantly.

This scalability difference underscores the importance of algorithm selection in computational tasks, especially when dealing with large datasets or intensive calculations.

**4.** Plot the primes generated by your method in a histogram. You may replace the plot of the brute force algorithm already displayed. Comment on the distribution of primes. (e.g., Are the primes evenly distributed? Do they tend to occur in a specific range plotted?)

**A:** I generated a histogram of the prime numbers up to 200,000 using matplotlib.



The resulting plot reveals several interesting characteristics of prime number distribution:

1. The primes are not evenly distributed across the range. There is a clear concentration of primes among smaller numbers, as shown by the taller bars on the left side of the histogram.
2. The frequency of primes decreases as the numbers get larger. This is evident from the gradually shortening bars from left to right in the histogram.
3. Primes tend to occur more frequently in the lower range of the plotted numbers. The highest concentration is visible in the leftmost part of the histogram, likely representing primes up to about 10,000.
4. While primes become less frequent in higher ranges, they continue to occur throughout the entire plotted range. Even at the right end of the histogram (near 200,000), we can still see bars indicating the presence of primes, albeit at a much lower frequency than at the start.
5. The overall shape of the distribution resembles a hyperbolic curve, with a steep decline in frequency at the beginning and a more gradual decrease as the numbers get larger.

Source file: