## Implementation:

The Brute Force Search Algorithm was implemented using Python 3.7, using the time and matplotlib libraries to get an average runtime for the algorithm being ran N times. The matplotlib library is used to generate a bar plot of 3 distinct test cases, with each test case being a sequence of the Worst, Best, and Average case scenarios for the Brute Force Search Algorithm.

## Testing:

Let N be the # of times the function is ran. It is beneficial to get an average of the amount of time it takes to run a function multiple times.

In this analysis we are testing for:
-the average runtime for the Brute Search Algorithm
Based on:

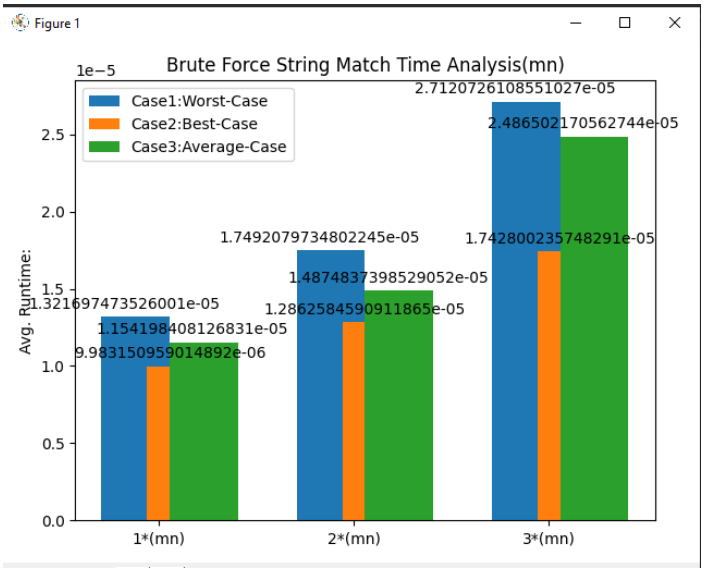
1. Different inputs of size (mn)
2. Worst-Case, Best-Case, and Average-Case time efficiencies

## Methodology:

Given the known theoretical efficiency for the Worst, Best, and Average case efficiencies for the Brute Force String Match Algorithm.

We can used these known values as control variables for testing the algorithm against itself. If we pick an arbitrary starting value close to when the program began, it's value should match approximately compared to the experimental value we received from the visualization program.

## Visualization:




Using O(n) notation, we can alter n and observe if the theoretical values match up with the experimental values produced by the program.

**Types of cases:**

Worse-Case Scenario:

The pattern to be matched is located at the end of the text string.

m(n-m+1) character comparisons for the worst-case scenario.

$$= O(mn)$$

Best-Case Scenario:

The pattern to be matched is at the beginning of the text string.
Only m key comparisons will be made (the length of the pattern).

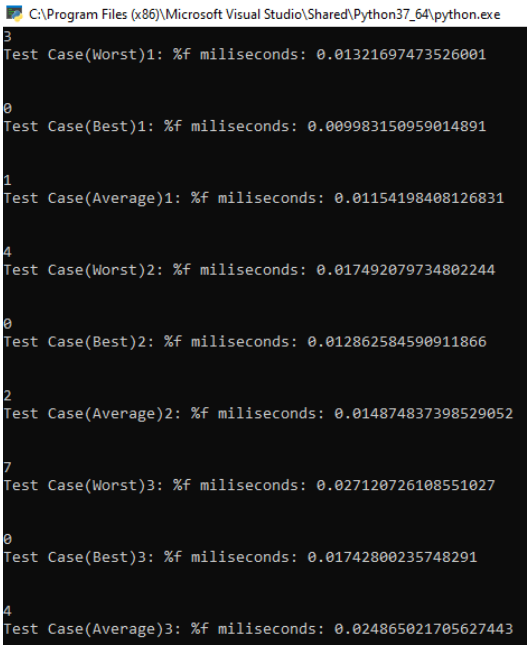
$$= O(m)$$

Average-Case Scenario:

The pattern to be matched is located somewhere in the middle of the text string.

$$= O(n)$$

**Results:**




| Experimental | 1* nm | 2*nm | 3*nm |
|---|---|---|---|
| Worst-Case | .01321 | .01749 | .02712 |
| Best-Case | .00998 | .01286 | .01742 |
| Average-Case | .01154 | .01487 | .02486 |

| Control | 1*control | 2*control | 3*control |
|---|---|---|---|
| Worst-Case | .01321 | .02642 | .03963 |
| Best-Case | .00998 | .01497 | .01996 |
| Average-Case | .01154 | .01731 | .02885 |

$$\%\ \boldsymbol{error} = \frac{\boldsymbol{theoretical - experimental}}{\boldsymbol{theoretical}} * \mathbf{100}\%$$

$$\%\ \boldsymbol{error}(\boldsymbol{Worst}) = \frac{.03963 - .02712}{.03963} * \mathbf{100}\% = \mathbf{31.5}\%\ \boldsymbol{error}$$

$$\%\ \boldsymbol{error}(\boldsymbol{Best}) = \frac{.01996 - .01742}{.01996} * \mathbf{100}\% = \mathbf{12.7}\%\ \boldsymbol{error}$$

$$\%\ \boldsymbol{error}(\boldsymbol{Average}) = \frac{.02885 - .02486}{.02885} * \mathbf{100}\% = \mathbf{13.83}\%\ \boldsymbol{error}$$

**Conclusion:**

The time complexity of the Brute Force Search Algorithm is dependent upon either m, n, or m*n. The experimental data produced by the program produces a linear trend for these various input sizes.

The values recorded are not precise when compared to the theoretical values. If I had more time for revisions/implementations, I would make use of the timeit library. I did a brute force calculation for the average run time, but the timeit library is much more exact.

Upon running the program multiple times, a linear trend is still produced that somewhat matches the theoretical data. The starting point of the timer varies based on the system timer and local environment. Therefore, it may not be appropriate to analyze these values for precision with spending more time on timer implementation.