Analysis of the String Matching Problem

Problem Description:

My proposal is to observe empirically the complexity of different algorithms solving the String Matching Problem. The string matching problem is described such that we are given a Text (T) and we want to find a Pattern (P).

Real-World Applications of the Problem:

The String Matching Problem is common in text editor applications when the user is searching for a particular word in the document. String matching algorithms also search for patterns in DNA sequences. Internet search engines also use them to find web pages relevant to queries. The method of finding a Pattern (P) in a Text (T) can be applied to all of these applications.

Algorithms Used to Solve the Problem:

Note: n is the length of the Text and m is the length of the Pattern

Brute-Force String Matching Algorithm – Worst case RT = Θ(nm)

The Brute-Force method of solving the String-Matching problem uses a for loop to shift the position and checks if the pattern matches the text. This method can be thought of as a template of the pattern sliding over the text and stopping at each letter to see if the pattern begins to match at this position.

```
//Brute Force Pattern Matching
//RT = O(mn)
//m = P.length
//n = T.length
static int BruteForce_PatternMatching(String P, String T, int m, int n)
{
    for(int i = 0; i <= (n-m); i++)
    {
        for( int j = 0; j < m; j++)
        {

            if(P.charAt(j) != T.charAt(i+j))
            {
                break;//Pattern is not T[i...i+m]
            }
            if(j==m-1)
            {
                //Pattern was matched
                return i;//return index where pattern begins
            }
        }
    }
    return -1; //no match found
}
```

Knuth-Morris-Pratt String Matching Algorithm – Worst case RT = Θ(n + m)

The KMP method of solving the String-Matching problem consists of two phases, a pre-fix phase and a searching phase. It uses an auxiliary function π which is computed in the pre-fix phase from the pattern and stored in an auxiliary array π[1…m]. The array π will tell us the length of the longest substring that matches with the prefix of the pattern. During the searching phase when a mismatch is detected we will already know some of the characters in the pattern have been matched and we use this information to avoid matching the character we know already matched.

```
//Knuth-Morris-Pratt Pattern Matching
    //RT = O(m + n)
    //m = P.length
    //n = T.length
    static int KMP_PatternMatching(String P, String T, int m, int n)
    {
        int[] pi = Compute_Prefix(P, m);
        int i=0;
        //int i = -1;
        int match_index;

        for(int j=0; j<n; j++)
        {
            while(i>0 && P.charAt(i+1) != T.charAt(j))
            {
                i = pi[i];
            }
            if(P.charAt(i+1) == T.charAt(j))
            {
                i++;
            }
            if(i == m-1)
            {
                match_index = j - m + 1;
                return match_index;
            }
        }
        match_index = -1;
        return match_index;
    }
```

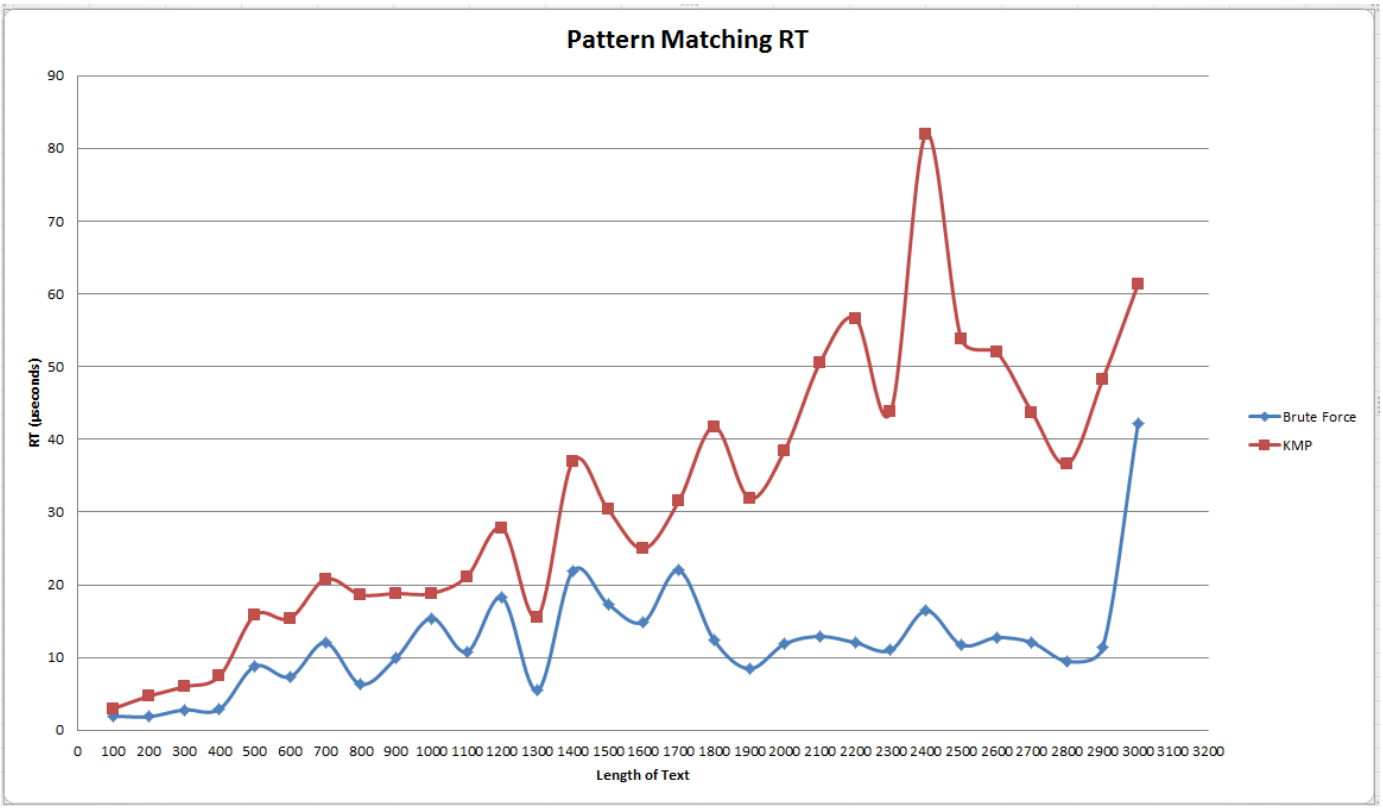
```
static int[] Compute_Prefix(String P,int m)
{
    int[] pi = new int[m+1];
    pi[0] = 0;
    pi[1] = 0;
    int k = 0;
    for(int q = 2; q < m; q++)
    {
        while(k>0 && (P.charAt(k+1) != P.charAt(q)))
        {
            k = pi[k];
        }
        if(P.charAt(k+1) == P.charAt(q))
        {
            k++;
        }
        pi[q]=k;
    }
    return pi;
}
```

Reference: Cormen, T., Leiserson, C., Rivest, R., Stein, C., (2009) *Introduction to Algorithms*. Lebanon, New Hampshire.

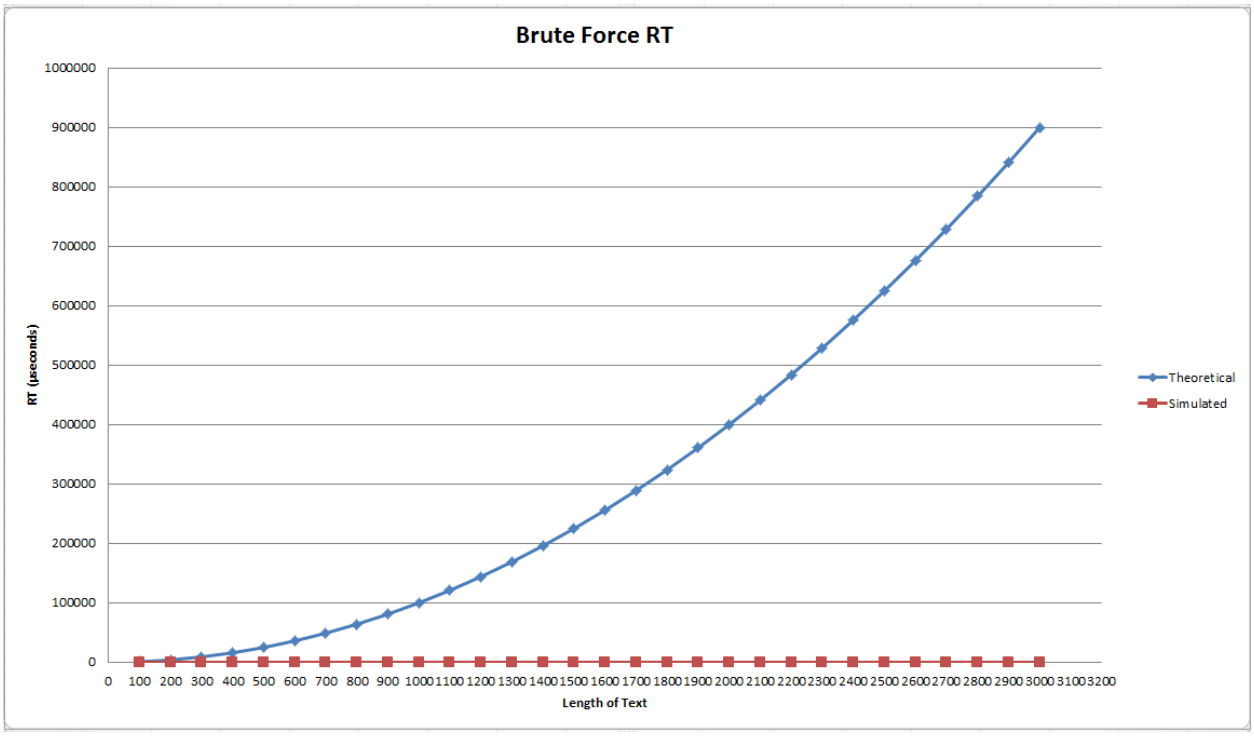
The Experiments:

To generate the comparison for the two algorithms I tested Text (T) with a length of 100 up to 3000 in increments of 100. I also changed the length of the Pattern (P) accordingly such that P.length = T.length/10 and randomly chose the starting index of the pattern to be between 0 and n. I did 5 trials for each scenario and plotted the average time.

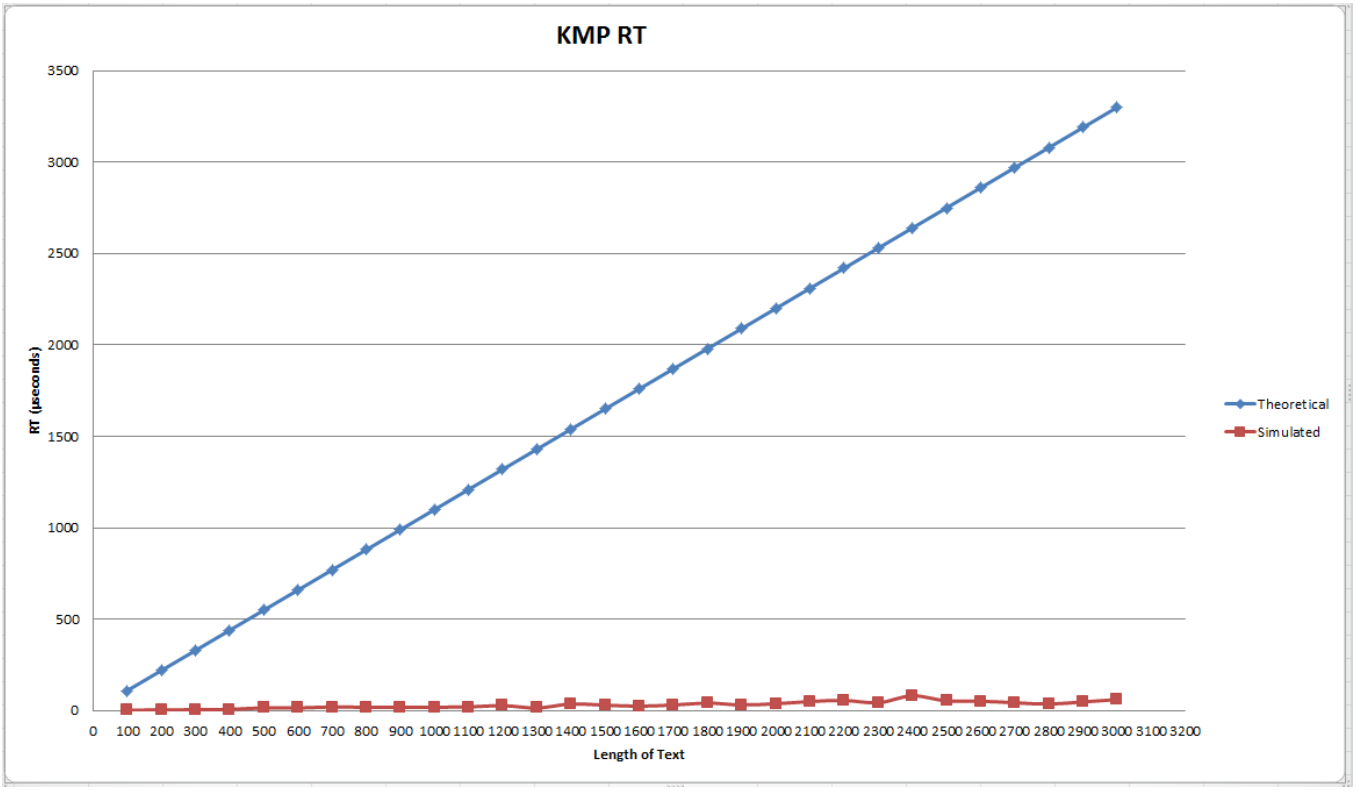
In this graph I compare simulated RT of the 2 algorithms Brute-Force vs Knuth-Morris-Pratt. The graph's y-axis is in microseconds and the x-axis will be the Text (T) length:



In this graph I compare simulated and theoretical RT of the Brute-Force algorithm. The graph's y-axis is in microseconds and the x-axis will be the Text (T) length:



In this graph I compare simulated and theoretical RT of the KMP algorithm. The graph's y-axis is in microseconds and the x-axis will be the Text (T) length:



Here I compare the Brute Force Simulated RT with the Theoretical RT and calculate the hidden constant:

BRUTE FORCE				
N	M	Theoretical RT=O(nm)	Simulated RT (µsec)	Hidden Constant
100	10	10^3	1.9674	0.001967
200	20	4x10^3	1.8824	0.000471
300	30	9x10^3	2.822	0.000314
400	40	16x10^3	2.9082	0.000182
500	50	25x10^3	8.8094	0.000352
600	60	36x10^3	7.3556	0.000204
700	70	49x10^3	12.06	0.000246
800	80	64x10^3	6.329	0.000099
900	90	81x10^3	9.9216	0.000122
1000	100	100x10^3	15.3954	0.000154
1100	110	121x10^3	10.7768	0.000089
1200	120	144x10^3	18.2176	0.000127
1300	130	169x10^3	5.4738	0.000032
1400	140	196x10^3	21.8958	0.000112
1500	150	225x10^3	17.3628	0.000077
1600	160	256x10^3	14.8824	0.000058
1700	170	289x10^3	21.981	0.000076
1800	180	324x10^3	12.402	0.000038
1900	190	361x10^3	8.4674	0.000023
2000	200	400x10^3	11.889	0.00003
2100	210	441x10^3	12.9148	0.000029
2200	220	484x10^3	12.0598	0.000025
2300	230	529x10^3	11.0336	0.000021
2400	240	576x10^3	16.422	0.000029
2500	250	625x10^3	11.7178	0.000019
2600	260	676x10^3	12.744	0.000019
2700	270	729x10^3	12.06	0.000017
2800	280	784x10^3	9.4938	0.000012
2900	290	841x10^3	11.3758	0.000014
3000	300	900x10^3	42.1664	0.000047
C = max(individual hidden constant cell values) =				0.001967

Here I compare the Knuth-Morris-Pratt Simulated RT with the Theoretical RT and calculate the hidden constant:

KNUTH-MORRIS-PRATT

N	M	Theoretical RT=O(n + m)	Simulated RT (µsec)	Hidden Constant
100	10	110	2.9934	0.027213
200	20	220	4.7036	0.02138
300	30	330	5.9874	0.018144
400	40	440	7.441	0.016911
500	50	550	15.823	0.028769
600	60	660	15.3958	0.023327
700	70	770	20.7836	0.026992
800	80	880	18.6458	0.021188
900	90	990	18.8166	0.019007
1000	100	1100	18.8168	0.017106
1100	110	1210	21.126	0.01746
1200	120	1320	27.7978	0.021059
1300	130	1430	15.5662	0.010885
1400	140	1540	36.9492	0.023993
1500	150	1650	30.3632	0.018402
1600	160	1760	24.9748	0.01419
1700	170	1870	31.4754	0.016832
1800	180	1980	41.7388	0.02108
1900	190	2090	31.903	0.015265
2000	200	2200	38.4886	0.017495
2100	210	2310	50.5488	0.021883
2200	220	2420	56.6212	0.023397
2300	230	2530	43.7914	0.017309
2400	240	2640	81.938	0.031037
2500	250	2750	53.7988	0.019563
2600	260	2860	52.0026	0.018183
2700	270	2970	43.7058	0.014716
2800	280	3080	36.6072	0.011885
2900	290	3190	48.2392	0.015122
3000	300	3300	61.3256	0.018584
C = max(individual hidden constant cell values) =				0.031037

Experiment Results:

My findings from the results of this project were unexpected and interesting. I expected the KMP algorithm to have a faster simulated run-time since its theoretical run-time implies it would be much faster. However the simulation showed that the Brute Force algorithm actually ran faster. After doing further research I discovered that KMP will only run faster for enormous strings of text or when the same pattern will be repeatedly searched for, since the π array could then be used over again without reconstruction. The KMP algorithm takes a little longer in the simulation because of the construction of the π array. The Brute Force algorithm is simple and proved for relatively short text strings it actually runs faster than KMP. For both algorithms the simulated run time is much faster than expected from the theoretical prediction.