Practical 7 – Analysis (SubString Searches)

6th April 2021

19428806

1. Implement a bruteforce substring search algorithm

```
public static int bruteForcePatSearch(String txt, String pat) {
    int n = txt.length();
    int m = pat.length();

    for (int i = 0; i <= n - m; i++) {
        int k;
        for (k = 0; k < m; k++) {
            if (txt.charAt(i + k) != pat.charAt(k)) break;
        }
        if (k == m) return i; // i is the index at where the pattern
begins.
    }
    return n;
}

public static void printPattern(int position, String txt, String
pat){
    int m = pat.length();
    for (int i = 0; i < m; i++){
        System.out.print(txt.charAt(position));
        position++;
    }
}
```

2. Implement a version of Knuth-Morris-Pratt algorithm

```
public void Knuth_Morris_Pratt(String txt, String pattern) {
    int M = pattern.length();
    int N = txt.length();

    //lps[] will hold the longest suffix values for pattern
    int[] lps = new int[M];

    int j = 0; // index for the pattern
    computeLPSArray(pattern, M, lps);

    int i = 0; // index for txt[]

    while (i < N) {
        if (pattern.charAt(j) == txt.charAt(i)) {
            j++;
            i++;
        }
        if (j == M) {
            System.out.println("Found pattern " + "at index " + (i -
j));
            j = lps[j - 1];
        }


        // mismatch after j matches

        else if (i < N && pattern.charAt(j) != txt.charAt(i)) {
```

```
                // Do not match lps[0..lps[j-1]] characters,
                // they will match anyway
                if (j != 0)
                    j = lps[j - 1];
                else
                    i = i + 1;
            }
        }
}
```

```
void computeLPSArray (String pattern, int m, int[] lps) {
    //length of the previous longest prefix suffix
    int len = 0;
    int i = 1;
    lps[0] = 0; // lps[0] is always 0;

    // the while loop computes lps[i] for i = 1 to m - 1
    while (i < m) {
        if (pattern.charAt(i) == pattern.charAt(len)) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0)
                len = lps[len - 1];
            else {
                lps[i] = len;
                i++;
            }
        }

    }
}
```

3. Assess the performance difference between the two algorithms with different inputs

Testing method

```
5 input files corresponding number of words to count
 - int[] dataCount = new int[]{10,100,1000,10000,58110};
```

5 targets each corresponding to the index of dataCount.

```
String[] targets = new String[]{"form", "inic" , "stroy", "eet", "oom"};
```

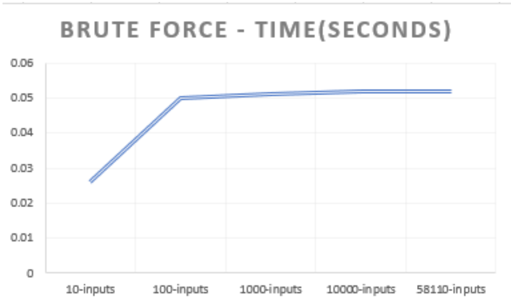
Both arguemnts used on both algorithms for accurate results.

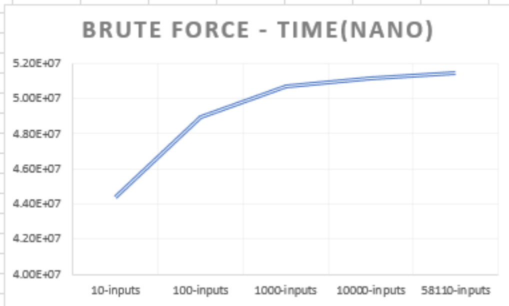
Timing of Each algorithm

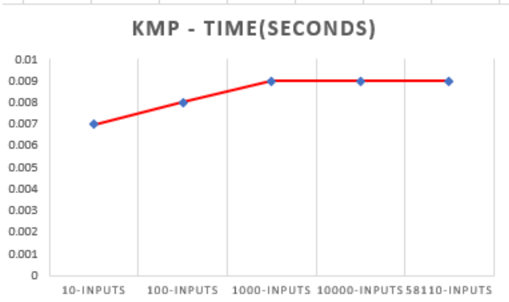
Algorithm	Inputs	Time(Seconds)	Time(nano)
BruteForcePatSearch	10-inputs	0.026	4.44E+07
	100-inputs	0.05	4.90E+07
	1000-inputs	0.051	5.07E+07
	10000-inputs	0.052	5.12E+07

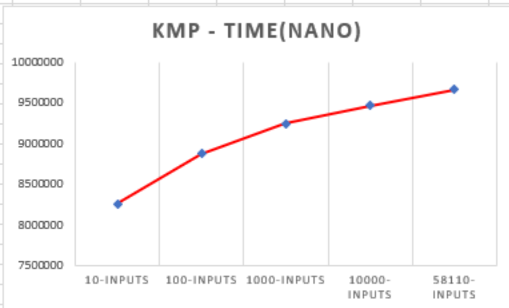
	58110-inputs	0.052	5.15E+07

Algorithm	Inputs	Time(Seconds)	Time(nano)
Knuth-Morris-Pratt	10-inputs	0.007	8254900
	100-inputs	0.008	8872100
	1000-inputs	0.009	9.25E+06
	10000-inputs	0.009	9.46E+06
	58110-inputs	0.009	9.66E+06





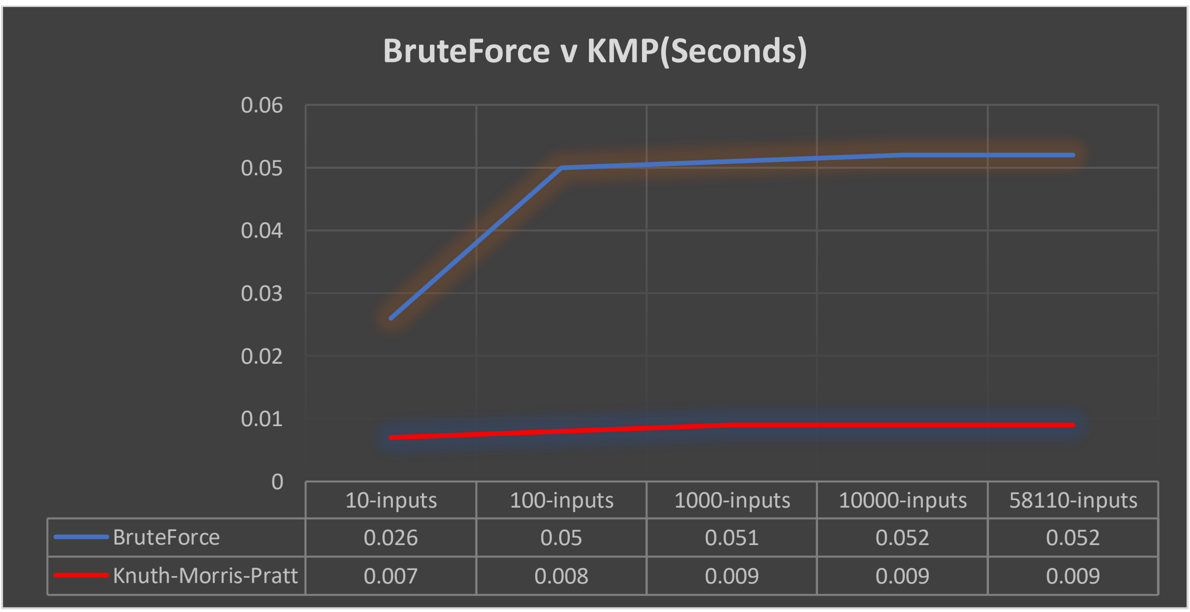




1. Brute Force – Evident quadratic shape in the nano timing graph

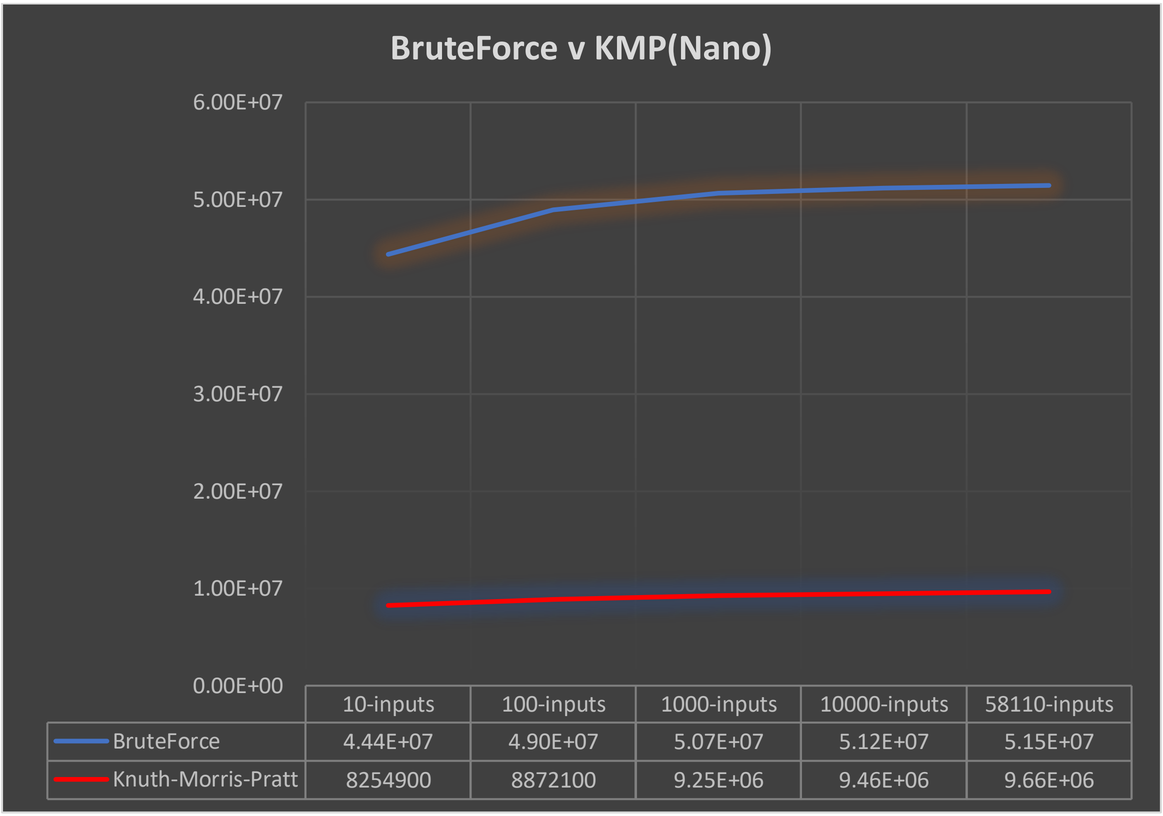
2. KMP - As we can see the linear relationship between KMP and its results graphed.

Comparing Graphs



Brute force (blue line) has a big jump in time consumption from 10inputs – 100inputs

But then the time usage maintains a linearly for the next 4 searches



<u>Unline Brute Force, KMP is guaranteed linear time for each possible outcome. This algorithm is beautifully written and its savings on time usage is elagent.</u>

Q.1) What would you say the complexity of the Brute Force substring search algorithm is?

Quadratic in the worst case. 0(n^2)

Q.2) What would you say the complexity of the KMP algorithm is?

Guaranteed to be linear in the worst case. O(N)