A Comparison between string-matching algorithms

Mohamed Samine

May 15, 2021

1 Introduction

The purpose of this paper is to explore the string-matching problem. The problem is to find all occurrences of a pattern in a text.

In this paper, we compare 5 string-matching algorithms*:*

Brute Force, Sunday, Rabin Karp, Finite State Machine and Knuth–Morris–Pratt.

1.1 Brute Force

Brute force also called naive string-matching algorithm achieves its goal of finding the pattern in the text by looping over the text character by character (outer loop) and for each character of the text the algorithm loops over the pattern characters (inner loop) and checks if we have a match.

It does so by using a text iterator "i" (outer loop) and a pattern iterator "j" (inner loop), in each iteration the algorithm compares the $(i + j)^{th}$ character in the text with the j^{th} character in the pattern, if two doesn't match it breaks out of the inner pattern loop and the algorithm goes to the next character in the text, if the inner loop finishes its iteration it means that a match of the pattern was found in the text starting at the position i.

Time complexity

Looping over the pattern characters for each character in the text result in the naive string-matching algorithm having a worst case of $O(n*m)$*,* with m being the length of the pattern and n being the length of the text.

1.2 Sunday

Sunday string-matching algorithm uses a precomputed table to make shifts that are more than one character long (if it is possible) when looping over the text, the precomputed table holds the length of the shifts the algorithm should do for each alphabet that we may encounter in the text.

The Sunday algorithm starts by preparing the precomputed table, first the default value of the table is set to a non-valid value, then for each alphabet in the pattern its value in the table is updated to be the last positions where it is found in the pattern.

The algorithm uses the precomputed table while looping through the text string, once a mismatch is found, the character after the current comparison window is used to define the length of the next shift by using its alphabet value from the precomputed table, if the character simply does not exist in the pattern, the shift length will be pattern length.

Time complexity

Preprocessing is $O(a*m)$ with a being the number of alphabets and m being the length of the pattern, but since a is a constant, we can say that preprocessing takes $O(m)$.

Overall, the algorithm has a worst case of $O(m) + O(n*m)$, n being the length of the text. it happens when the algorithm is doing shifts of one character long all the time, which is the case if the text contains a repetition of the same character.

As of best case, the algorithm performs $O(m) + O(n/m)$.

1.3 Rabin-Karp

Rabin-Karp algorithm also loops over the text one character at time, but unlike the Brute Force algorithm, Rabin-Karp algorithm compares the hash value of the pattern with the hash value of the current sub-string of text, and if the hash values match, then it makes sure of the match by comparing individual characters.

The Rabin-Karp algorithm uses a rolling hash function that avoids recalculating from scratch the hash value for each sub-string of the text, which is not efficient, instead only the hash value of the initial sub-string starting at index 0 of the text is calculated by iterating through all its characters, for all the other future sub-strings the hash value is calculated by a formula that subtracts from the previous hash the character that is no longer going to be part of the next shift and adds the new character that is going to be introduced by the next shift.

Time complexity

The best and average case of the Rabin-Karp algorithm is $O(n+m)$, as for its worst-case time it is $O(nm)$. The worst case of Rabin-Karp algorithm occurs when all characters of the text and the pattern are the same since the hash values of all the sub-strings of the text match with hash value of pattern.

1.4 Finite State Machine

The basic idea of Finite State Machine algorithm is to loop over the text one character at time while comparing it with the pattern, but unlike previously seen string-matching algorithms FSM avoids re-comparing previously read characters from the text in case a mismatch is found, it does so by using a precomputed state table to get the state to which the comparison algorithm should move to when a mismatch is found, so instead of checking the pattern from the beginning, the precomputed state table is used to determine to which point (state) in the pattern the algorithm should go back to.

During the preprocessing stage, the state table is created for the given pattern, the state table is 2 dimensions array that holds the state to which the algorithm should move to for each given character in the alphabet and each state, the next state is calculated to be the length of the longest prefix which is also a suffix of the pattern up to the point $k-1$ with k being the given state.

Time complexity

The FSM algorithm have a preprocessing phase for preparing the Pi table which takes $O(m)$ with m being the length of the pattern.

Since the presented FSM algorithm is using the precomputed Pi table that takes $O(m)$ to compute, to generate the state table the time complexity of the precomputation phase sums up to $O(m) + O(m*a)$ with m being the length of the pattern and a being the number of characters used in the alphabets.

As for the search process the time complexity of the algorithm is $O(n)$ with n being the length of the text.

Overall, the time complexity of the algorithm is $O(m) + O(m*a) + O(n)$.

1.5 Knuth–Morris–Pratt

Just like FSM algorithm, Knuth-Morris-Pratt algorithm follows the idea that whenever we detect a mismatch after some matches, we already know part of the next comparison window,

we take advantage of this information to avoid matching the characters that we know will anyway match.

To achieve its goal of avoiding the repetitive comparison of already matched characters, the KMP algorithm uses a precomputed **Pi** table that holds for each sub-pattern that starts at index **0** and ends at index **i** with **i** being from range **0** to **m-1**, the length of the longest proper prefix of the pattern which is also a suffix of the sub-pattern. In case of a mismatch KMP differentiate it self from FSM by calculating the state to which it should fall back on the fly using the precomputed **Pi** table.

Time complexity

The KMP algorithm have a preprocessing phase for preparing the **Pi** table of **O(m)** time complexity with **m** being the length of the pattern.

As for the searching phase the algorithm has **O(n)** time complexity with **n** being the length of the text.

Altogether, the KMP algorithm has a time complexity of **O(m) + O(n).**

2 Methodology

The algorithms were implemented using the JavaScript programming language.

The comparison was conducted on a increasing text length starting at 1000 characters up to 56000 characters, using two patterns, a small one of 5 characters and a large one of 317 characters.

Each algorithm was tested on both patterns, and each test was repeated 100 times, the average time of the repeated tests was logged into a .csv file for each algorithm.

3 Results

The chart in Figure 1 shows the average time it took each algorithm to find all occurrences of a 317 characters pattern in an increasing text. It can be noticed from this chart that FSM algorithm is the slowest among them all. The slowness of the FSM algorithm is due to the preprocessing stage, as the algorithm needs to prepare a state table, the state table holds a state to which the algorithm can fall back in case of a mismatch, and as it needs to save a state for each possible character in our text and each state in which we may be, hence, the time needed to prepare the state table depends on the size of the pattern. As a result, the state table has a time complexity of **O(m*a)** with **m** being the length of the pattern and **a** being the

number of characters in our alphabets, and since the algorithm needs to precompute the **Pi** table as well, which also depends on the size of the pattern**,** the overall complexity of the FSM preprocessing stage is **O(m) + O(m*a)**.

Second thing that can be observed from the chart in Figure 1 is that the fastest algorithm in the list is by far Sunday, which is then followed by Rabin Karp, KMP and then Brute Force. This proves that with its ability to do shifts larger than one character long in case of a mismatch, Sunday excels in human readable text, where this situation is most likely to happen.

Figure 2, the chart shows us a representation of the time it took each algorithm to find all occurrences of a 5 characters long pattern in an increasing text. The same trend observed in Figure 1 with a large pattern can be seen in Figure 2, except that this time FSM is the second fastest algorithm in the list, because unlike with a large pattern where FSM algorithm was taking so much time in the preprocessing phase, with a small pattern the algorithm's precomputation is taking way less time. Additionally, the precomputed state table gives FSM an advantage later in the search phase over KMP, because it doesn't need to do any computation on the fly to decide the state to which the algorithm has to fall back to in case of a mismatch, as it is all saved in the precomputed state table. It's the same reason why it is performing a bit better than Rabin-Karp, as the latter is using the rolling hash function during the search phase to calculate the hash value for every new sub-string.

In both experiments, we can observe that for a text of small size (<7000) Brute Force algorithm is performing better than it is doing for larger text.

4 Conclusion

Based on the test conditions presented above, we can conclude that with a human readable text the Sunday algorithm is by far the fastest for both large and small patterns. Additionally, we deduce that the Finite State Machine algorithm should be avoided with large patterns.

Figure 1: Comparison of the average time each algorithm took to find the same pattern (317 characters length) in an increasing text.

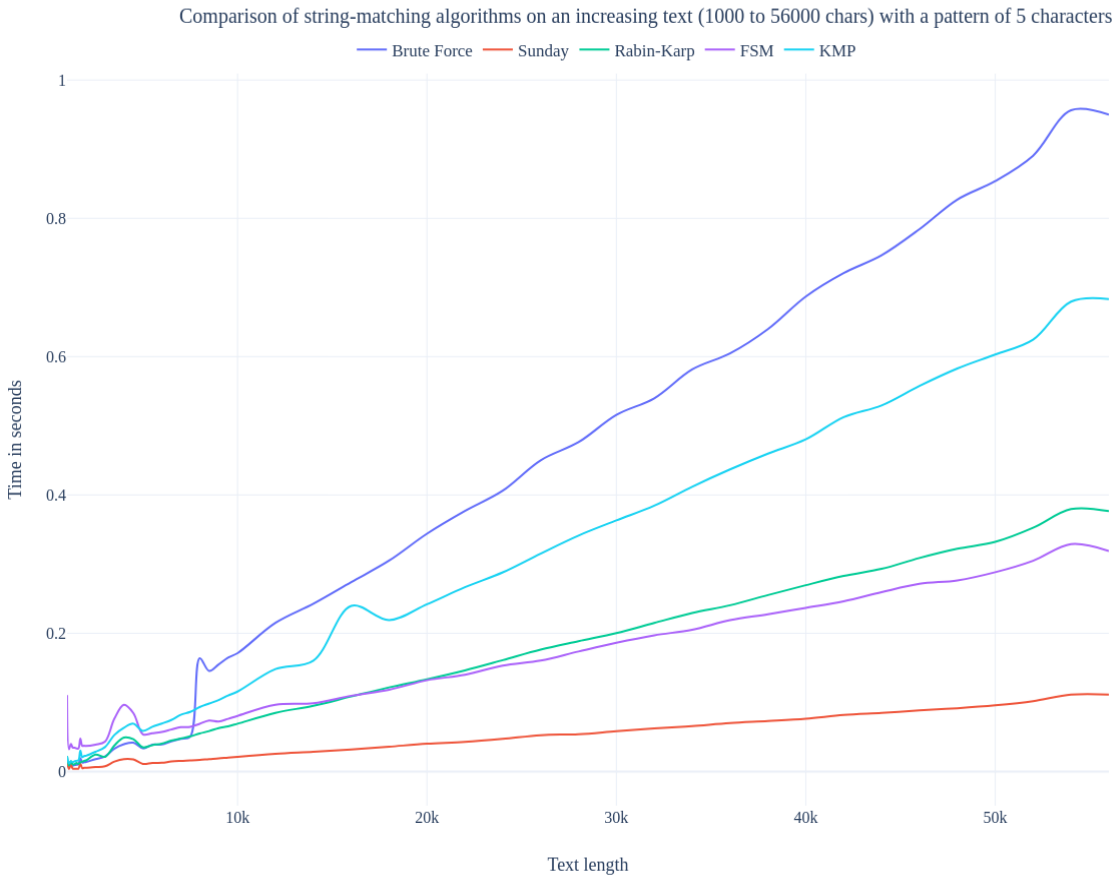

Figure 2: Comparison of the average time each algorithm took to find the same pattern (5 characters length) in an increasing text.