String Matching Algorithms

Author	Enes Kızılcan
Link	https://github.com/eneskzlcn
Publishing/Release Date	@November 11, 2021

String Matching Algorithms

In this work, I will search and implement (if possible) the string matching, string searching algorithms in below.

- Boyer Moore Algorithm
  - How this algorithm works?
  - Implementation Of Algorithm
- Naive Algorithm
  - How this algorithm works?
  - Implementation Of Algorithm
- KMP Algorithm
  - How this algorithm works?
  - Implementation Of Algorithm
- Rabin-Karp Algorithm
  - How this algorithm works?
  - Implementation Of Algorithm
- Aho-Corasick Algorithm
  - How this algorithm works?
  - Implementation Of Algorithm
- Z Algorithm
  - How this algorithm works?
  - Implementation Of Algorithm

Naive Algorithm

How this algorithm works?

This algorithms works in a brute force approach. Let think you have a text and pattern like in example below:

> Text: A B C D E D G H
>
> Pattern: D G

The algorithm compares the first character in the text with the first character in pattern firstly.

If any match, then second character of text and pattern will be comparised. This operation will go over an over for each character in pattern. If all the character matches, then you found this pattern at this index of the text.

If any mismatch, then you shift pattern 1 index and continue to compare operation again. Let show in example above.

1.Step: Mismatch A ≠ D . So shift the pattern

> A B C D E D G H
> D G

2.Step: Mismatch B ≠ D . So shift the pattern

> A B C D E D G H
> D G

3.Step: Mismatch C ≠ D . So shift the pattern

> A B C D E D G H
> D G

4.Step: Match In D= D. So look next character. But next is mismatch E ≠ G . So shift the pattern

> A B C D E D G H
> D G

5.Step: Match In E≠D So shift the pattern .

> A B C D E D G H
> D G

6.Step: Match In D = D . And all the other characters in pattern matches too. So you found a match at that index. It is okey to continue to found other ones in exactly the same way.

> A B C D E D G H
> D G

Implementation of algorithm in javascript

```
function naive_string_matching(pattern, text) // 'text' is the string that we search for given 'pattern' string.
{
    //the time complexity of naive string matching algorithm is  O( m * (n - m + 1)) where n is the text length and m is the pattern length

    //we need to find the length of pattern and text firstly.
    //This variables for just for understanding and reading clearly. String in js have a length property simply.
    //You do not need to declare them in your code.
    var text_length = text.length;
    var pattern_length = pattern.length;

    let pattern_found_indexes = [];
    //we need to loop from 0th index of text to the ( text_length - pattern_length)th index. This is the last index can pattern start from.
    let last_possible_patern_index = (text_length - pattern_length);
    let matched_character_count;

    for (let i = 0; i <= last_possible_patern_index; i++) {
        matched_character_count = 0;
        for (let j = 0; j < pattern_length; j++) {
            if (text.charAt(i + j) != pattern.charAt(j)) break; //if any mismatch directly break.
            matched_character_count++;
        }
        if (matched_character_count == pattern_length) // if there is no break in upper loop that means fullmatch. So push it.
```

```
            pattern_found_indexes.push(i);
        }
        return pattern_found_indexes;
    }
```

Boyer Moore Algorithm

How this algorithm works?

This algorithms have 2 aproach named **bad suffix heuristic** and **good suffix heuristic**.

Bad Suffix Herustic

In the bad suffix, you create a table and evaluate every character in the pattern (every different character). This values are shifting values for any mismatch happens. As you see in below example , if you got a mismatch ,you are looking at the bad character that causes this mismatch (the character in the text , not in pattern). If the character is already in the table we created with pattern characters, we get the value for that character and shift the pattern by this amount. If the character not in table you just carry all the string after this bad character (if the character causes mismatch not in table so not one of pattern characters, that means you should pass directly this character to not make mismatch over and over for nothing. Just skip this character by shifting the pattern and continue to search.

Creating Bad Suffix Shifting Table

text = A B C D E F G H E Z H G A H Z G A S

pattern = H Z G A | pattern | = 4

$$formula = max(1, paternLength - charIndexInPattern - 1);$$

value H : max(1, 4-0-1) = 3

value Z: max(1, 4-1-1) = 2

value G: max(1, 4-2-1) = 1

value A: max(1, 4-2-1) = 1

table: | H | Z | G | A | * | * (Other any char out of pattern characters)

value: | 3 | 2 | 1 | 1 | 4 |

text = A B C D E F G H E Z H G A H Z G A S

pattern = H Z G A

Step1:**Start from last character which is A. There is a mismatch and the bad character of this mismatch is 'D' .so shift amount of D value in table. D is not in table so shift it amount of pattern length.(4)**

A B C **D** E F G H E Z H G A H Z G A S

H Z G **A**

Step 2: There is a mismatch again. Bad character here is 'H'. The value of H in table is 3. So shift the pattern 3 times.

A B C D E F G H E Z H G A H Z G A S

H Z G A

Step 3: There is a mismatch again. Bad character here is 'H'. So shift pattern 3 times.

A B C D E F G H E Z H G A H Z G A S

H Z G A

Step 4: Same with step 3. Shift 3 times.

A B C D E F G H E Z H G A H Z G A S

H Z G A

Step 5: In here you catch a match. Save the index you found and continue to next step.

A B C D E F G H E Z H G A H Z G A S

H Z G A

Step 6: This is the last index you can be shifted. And there is mismatch. So the algorithm finished its work.

A B C D E F G H E Z H G A H Z G A S

H Z G A

Implementation of algorithm in javascript

Implementation Of Creating A Bad Suffix Table

```
//this functions find the bad suffix value for each character in pattern by the bad suffix table formula
export function boyer_moore_creating_bad_suffix_table(pattern) {
    var table = []; // en empty table. this table will contain objects in this type ex : {letter: 'A', value: 5}

    for (let i = 0; i < pattern.length; i++) { // loop to reach every char in pattern string
        let letter = pattern.charAt(i); // char at ith index
        let value = Math.max(1, (pattern.length - i - 1)); // this is the bad suffix table formula  val = max(1,(len-index-1))

        let is_letter_added = table.find(r => r.letter == letter); //if the letter already in array do not push the same letter again. Just
        if (is_letter_added) is_letter_added.value = value;
        else table.push({
            "letter": letter,
            "value": value
        });
    }
    return table;
}
```

Implementation Of Boyer More Bad Suffix Searching Algorithm

```
function boyer_moore_string_matching(pattern, text) {

    var pattern_found_indexes = [];
    //create table for pattern
    var table = boyer_moore_creating_bad_suffix_table(pattern);

    // loop from 0 th index to (text length - pattern length) 'th index
    let i = 0;
    while (i <= (text.length - pattern.length)) {
        let j = pattern.length - 1;

        while (j >= 0 && (pattern.charAt(j) == text.charAt(i + j)))
            j--;

        if (j < 0) // that means the inner while loop completed succesfully. so the pattern found at i index
        {
            pattern_found_indexes.push(i); //save this index and shift i one time
            i++;
        } else { // that means there is a mismatch in i+j index. Find the bad character
            var bad_match_char = text.charAt(i + j); // bad character
            var is_char_in_table = table.find(r => r.letter == bad_match_char); // control if the char in table( in pattern)
            if (is_char_in_table) { // if it is , take its value as shift amount and shift i by this amount.
                var shift_amount = is_char_in_table.value;
                i += shift_amount;
            } else { // if the bad character not in table, just pass all the way of it so shit i pattern.length times.
                i += pattern.length;
            }
        }
    }
    return pattern_found_indexes;
}
```