# **BOYER MOORE ALGORITHM WITH BAD CHARACTER HEURISTIC**

Pattern matching problems are very popular among string problems. There are some well-known algorithms to solve them, among them the KMP algorithm, Rabin-Karp algorithm, and the Boyer Moore algorithm, which are the most efficient and most used. We will discuss the Boyer Moore algorithm in this article. Similar to the KMP algorithm, we also do some pre-processing in the Boyer Moore algorithm.

In this algorithm, we start to check characters from the right and then move to the left. We compare characters of the pattern (let us call it P) and text (let us call it T). We get a match then we check the next characters of both P and T, and if we get a mismatch then we check whether the character in T which got the mismatch (let us call it ‘c’) is present anywhere in P. If it is present then we shift P until we get c occurrence in P aligned with T, and if it is not present then we completely shift P past that character in T. This type of shift avoids a comparison of needless characters again and again,  improving the running time of the algorithm. More specifically, this is bad character heuristics, which we will discuss in detail. There is also a different version of this strategy known as good suffix heuristics, in which efficiency further improves, and it can also be used with the bad character heuristic for better implementation.

We use two different strategies (formally called heuristics) in this algorithm, which work independently or together, which reduces the search.

1. Bad character heuristics
2. Good suffix heuristics

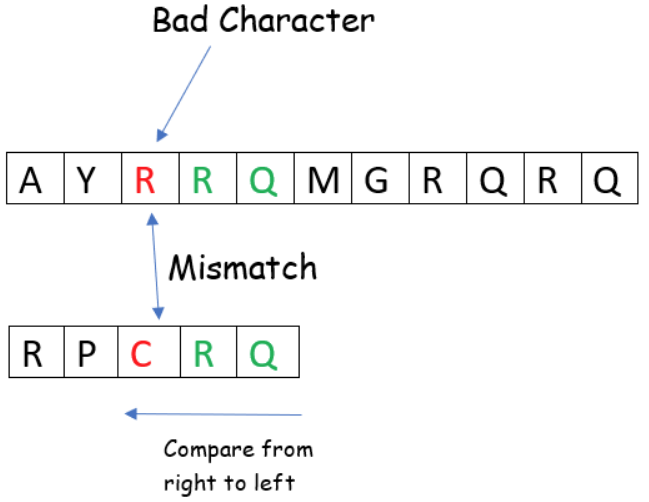
We will discuss bad character heuristics which is used more and is easier to implement as compared to good suffix heuristics, which is more efficient but is harder to implement.

BAD CHARACTER HEURISTICS

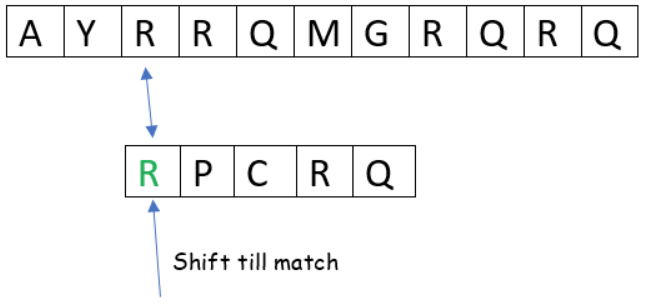
Here we consider two cases, and we call the character of the text which is not matching with the pattern character a **bad character**.

**Case 1:** The mismatched character of text T is present in pattern P.

In this case, we will shift the pattern P until it gets aligned to the mismatched character of T.

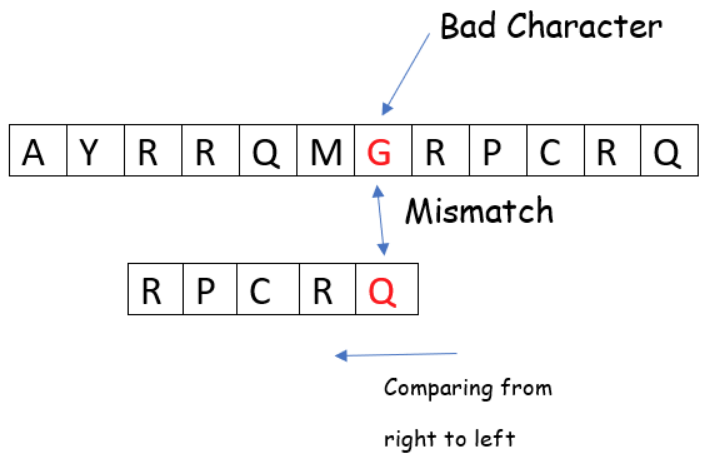


Since we got a mismatch between the ‘R’ of text (the bad character) and ‘C’ of the pattern, but we also know that ‘R’ is present in the pattern, so it is the first case, then we will shift the pattern until it matches with ‘R’ (bad character) of the text.

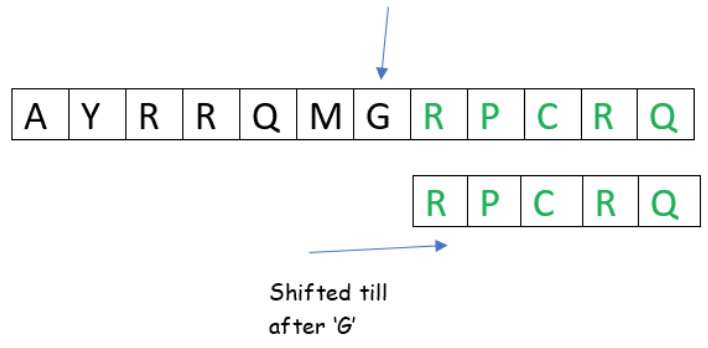


Here is what we got after shifting the pattern. We shifted it because it might be the case, in some situations, that we may get a matching pattern from that position.

**Case 2**: The mismatch character of text T is not present in pattern P.  
In this case, we will shift the pattern P until it gets past that mismatched character of T.



Here we got a mismatch, but “G” is not present in the pattern so there is no point in comparing the pattern again to any previous one, so we will shift pattern P until after “G” of text.



After the shift, we got our pattern in the above example, but there may be a case in which we didn’t get any matches. In that case we would return -1.

We are using a preprocessing function to fill the bad\_character array to get the last occurrence of a character which will take O(n).

In the above code, n indicates the size of text T and m indicates the size of pattern P. We are using the while loop (looping till n-m) in which we are using variable ‘idx’, which indicates how much the pattern P is shifted, and j which is used to find a mismatch. There will be two conditions, one when j becomes -1, which indicates pattern P is present at the current shift so we will print it. Also, we will shift P such that the next character in text T is aligned with the last occurrence of it in pattern P so we can get the next position where pattern P is present if it is present more than once in text T. As we stated (idx+m<n) condition is used to cover the case when P occurs at the end of T. Another one is when j != -1. In this case, we will simply shift P as per the previous rule, but we need only positive shift, not negative, so we are using max function.

Bad character heuristics worst-case time complexity is O(mn) which may occur when both text T and pattern P have all characters the same, like in this case T=”ppppppppp” and P=”pppp”.  
In the best-case scenario, the bad character heuristic takes O(n/m) which may occur when all characters of pattern P and text T are different.

Doc Link : <https://www.topcoder.com/thrive/articles/boyer-moore-algorithm-with-bad-character-heuristic>

Visualization : <https://dwnusbaum.github.io/boyer-moore-demo/>