

# Boyer Moore Algorithm for Pattern Searching - GeeksforGeeks

**Source:** <https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/>

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund Boyer Moore Algorithm for Pattern Searching Last Updated : 23 Jul, 2025 Pattern searching is an important problem in computer science. When we do search for a string in a notepad/word file, browser, or database, pattern searching algorithms are used to show the search results. A typical problem statement would be- " Given a text `txt[0..n-1]` and a pattern `pat[0..m-1]` where  $n$  is the length of the text and  $m$  is the length of the pattern, write a function `search(char pat[], char txt[])` that prints all occurrences of `pat[]` in `txt[]`. You may assume that  $n > m$ . " Examples: Input: `txt[] = "THIS IS A TEST TEXT"` `pat[] = "TEST"` Output: Pattern found at index 10 Input: `txt[] = "AABAACAADAABAABA"` `pat[] = "AABA"` Output: Pattern found at index 0 Pattern found at index 9 Pattern found at index 12 In this post, we will discuss the Boyer Moore pattern searching algorithm. Like KMP and Finite Automata algorithms, Boyer Moore algorithm also preprocesses the pattern. Boyer Moore is a combination of the following two approaches. Bad Character Heuristic Good Suffix Heuristic Both of the above heuristics can also be used independently to search a pattern in a text. Let us first understand how two independent approaches work together in the Boyer Moore algorithm. If we take a look at the Naive algorithm, it slides the pattern over the text one by one. KMP algorithm does preprocessing over the pattern so that the pattern can be shifted by more than one. The Boyer Moore algorithm does preprocessing for the same reason. It processes the pattern and creates different arrays for each of the two heuristics. At every step, it slides the pattern by the max of the slides suggested by each of the two heuristics. So, it uses greatest offset suggested by the two heuristics at every step. Unlike the previous pattern searching algorithms, the Boyer Moore algorithm starts matching from the last character of the pattern. In this post, we will discuss the bad character heuristic and the Good Suffix heuristic in the next post.

**Bad Character Heuristic** The idea of bad character heuristic is simple. The character of the text which doesn't match with the current character of the pattern is called the Bad Character. Upon mismatch, we shift the pattern until - The mismatch becomes a match. Pattern  $P$  moves past the mismatched character.

**Case 1 – Mismatch become match** We will lookup the position of the last occurrence of the mismatched character in the pattern, and if the mismatched character exists in the pattern, then we'll shift the pattern such that it becomes aligned to the mismatched character in the text  $T$ . Explanation: In the above example, we got a mismatch at position 3. Here our mismatching character is "A". Now we will search for last occurrence of "A" in pattern. We got "A" at position 1 in pattern (displayed in Blue) and this is the last occurrence of it. Now we will shift pattern 2 times so that "A" in pattern get aligned with "A" in text.

**Case 2 – Pattern move past the mismatch character** We'll lookup the position of last occurrence of mismatching character in pattern and if character does not exist we will shift pattern past the mismatching character. Explanation: Here we have a mismatch at position 7. The mismatching character "C" does not exist in pattern before position 7 so we'll shift pattern past to the position 7 and eventually in above example we have got a perfect match of pattern (displayed in Green). We are doing this because "C" does not exist in the pattern so at every shift before position 7 we will get mismatch and our search will be fruitless.

**Implementation:** In the following implementation, we pre-process the pattern and store the last occurrence of every possible character in an array of size equal to alphabet size. If the character is not present at all, then it may result in a shift by  $m$  (length of pattern). Therefore, the bad character heuristic takes  $O(n/m)$  time in the best case.

Below is the implementation of the above idea: C++ /\* C++ Program for Bad Character Heuristic of Boyer Moore String Matching Algorithm \*/ #include <bits/stdc++.h> using namespace std ; #define NO\_OF\_CHARS 256 // The preprocessing function for Boyer Moore's // bad character heuristic void badCharHeuristic ( string str , int size , int badchar [ NO\_OF\_CHARS ] ) { int i ; // Initialize all occurrences as -1 for ( i = 0 ; i < NO\_OF\_CHARS ; i ++ ) badchar [ i ] = -1 ; // Fill the actual value of last occurrence // of a character for ( i = 0 ; i < size ; i ++ ) badchar [ ( int ) str [ i ] ] = i ; } /\* A pattern searching function that uses Bad Character Heuristic of Boyer Moore Algorithm \*/ void search ( string txt , string pat ) { int m = pat . size () ; int n = txt . size () ; int badchar [ NO\_OF\_CHARS ] ; /\* Fill the bad character array by calling the preprocessing function badCharHeuristic() for given pattern \*/ badCharHeuristic ( pat , m , badchar ) ; int s = 0 ; // s is shift of the pattern with // respect to text while ( s <= ( n - m ) ) { int j = m - 1 ; /\* Keep reducing index j of pattern while characters of pattern and text are matching at this shift s \*/ while ( j >= 0 && pat [ j ] == txt [ s + j ] ) j -- ; /\* If the pattern is present at current shift, then index j will become -1 after the above loop \*/ if ( j < 0 ) { cout << "pattern occurs at shift = " << s << endl ; /\* Shift the pattern so that the next character in text aligns with the last occurrence of it in pattern. The condition s+m < n is necessary for the case when pattern occurs at the end of text \*/ s += ( s + m < n ) ? m - badchar [ txt [ s + m ] ] : 1 ; } else /\* Shift the pattern so that the bad character in text aligns with the last occurrence of it in pattern. The max function is used to make sure that we get a positive shift. We may get a negative shift if the last occurrence of bad character in pattern is on the right side of the current character. \*/ s += max ( 1 , j - badchar [ txt [ s + j ] ] ) ; } } /\* Driver code \*/ int main () { string txt = "ABAAABCD" ; string pat = "ABC" ; search ( txt , pat ) ; return 0 ; } // This code is contributed by rathbhupendra C /\* C Program for Bad Character Heuristic of Boyer Moore String Matching Algorithm \*/ #include <limits.h> #include <stdio.h> #include <string.h> #define NO\_OF\_CHARS 256 // A utility function to get maximum of two integers int max ( int a , int b ) { return ( a > b ) ? a : b ; } // The preprocessing function for Boyer Moore's // bad character heuristic void badCharHeuristic ( char \* str , int size , int badchar [ NO\_OF\_CHARS ] ) { int i ; // Initialize all occurrences as -1 for ( i = 0 ; i < NO\_OF\_CHARS ; i ++ ) badchar [ i ] = -1 ; // Fill the actual value of last occurrence // of a character for ( i = 0 ; i < size ; i ++ ) badchar [ ( int ) str [ i ] ] = i ; } /\* A pattern searching function that uses Bad Character Heuristic of Boyer Moore Algorithm \*/ void search ( char \* txt , char \* pat ) { int m = strlen ( pat ) ; int n = strlen ( txt ) ; int badchar [ NO\_OF\_CHARS ] ; /\* Fill the bad character array by calling the preprocessing function badCharHeuristic() for given pattern \*/ badCharHeuristic ( pat , m , badchar ) ; int s = 0 ; // s is shift of the pattern with // respect to text while ( s <= ( n - m ) ) { int j = m - 1 ; /\* Keep reducing index j of pattern while characters of pattern and text are matching at this shift s \*/ while ( j >= 0 && pat [ j ] == txt [ s + j ] ) j -- ; /\* If the pattern is present at current shift, then index j will become -1 after the above loop \*/ if ( j < 0 ) { printf ( " \n pattern occurs at shift = %d" , s ) ; /\* Shift the pattern so that the next character in text aligns with the last occurrence of it in pattern. The condition s+m < n is necessary for the case when pattern occurs at the end of text \*/ s += ( s + m < n ) ? m - badchar [ txt [ s + m ] ] : 1 ; } else /\* Shift the pattern so that the bad character in text aligns with the last occurrence of it in pattern. The max function is used to make sure that we get a positive shift. We may get a negative shift if the last occurrence of bad character in pattern is on the right side of the current character. \*/ s += max ( 1 , j - badchar [ txt [ s + j ] ] ) ; } } /\* Driver program to test above function \*/ int main () { char txt [] = "ABAAABCD" ; char pat [] = "ABC" ; search ( txt , pat ) ; return 0 ; } Java /\* Java Program for Bad Character Heuristic of Boyer Moore String Matching Algorithm \*/ class AWQ { static int NO\_OF\_CHARS = 256 ; // A utility function to get maximum of two integers static int max ( int a , int b ) { return ( a > b ) ? a : b ; } // The preprocessing function for Boyer Moore's // bad character heuristic static void badCharHeuristic ( char [] str , int size , int badchar [] ) { // Initialize all occurrences as -1 for ( int i = 0 ; i < NO\_OF\_CHARS ; i ++ ) badchar [ i ] = - 1 ; // Fill the actual value of last occurrence // of a character (indices of table are ascii and // values are index of occurrence) for ( int i = 0 ; i < size ; i ++ ) badchar [ ( int ) str [ i ] ] = i ; } /\* A pattern searching function that uses Bad Character Heuristic of Boyer Moore Algorithm \*/ static void search ( char txt [] , char pat [] ) { int m = pat . length ; int n = txt . length ; int badchar [] = new int [ NO\_OF\_CHARS ] ; /\* Fill the bad character array by calling the preprocessing function badCharHeuristic() for given pattern \*/ badCharHeuristic ( pat , m , badchar ) ; int s = 0 ; // s is shift of the pattern with // respect to text // there are n-m+1 potential alignments while ( s <= ( n - m ) ) { int j = m - 1 ; /\* Keep reducing index j of pattern while characters of pattern and text are matching at this shift s \*/ while ( j >= 0 && pat [ j ] == txt [ s + j ] ) j -- ; /\* If the pattern is present at current shift, then index j will become -1 after the above loop \*/ if ( j < 0 ) { System . out . println ( "Patterns occur at shift = " + s ) ; /\* Shift the pattern so that the next character in text aligns with the last occurrence of it in pattern. The condition s+m < n is necessary for the case when pattern occurs at the end of text \*/ //

```

txt[s+m] is character after the pattern // in text s += ( s + m < n ) ? m - badchar [ txt [ s + m ] ] : 1 ; } else
/* Shift the pattern so that the bad character in text aligns with the last occurrence of it in pattern. The
max function is used to make sure that we get a positive shift. We may get a negative shift if the last
occurrence of bad character in pattern is on the right side of the current character. */ s += max ( 1 , j -
badchar [ txt [ s + j ] ] ); } } /* Driver program to test above function */ public static void main ( String []
args ) { char txt [] = "ABAAABCD" . toCharArray (); char pat [] = "ABC" . toCharArray (); search ( txt , pat
); } } Python # Python3 Program for Bad Character Heuristic # of Boyer Moore String Matching
Algorithm NO_OF_CHARS = 256 def badCharHeuristic ( string , size ): ''' The preprocessing function
for Boyer Moore's bad character heuristic ''' # Initialize all occurrence as -1 badChar = [ - 1 ] *
NO_OF_CHARS # Fill the actual value of last occurrence for i in range ( size ): badChar [ ord ( string [ i
])] = i # return initialized list return badChar def search ( txt , pat ): ''' A pattern searching function that
uses Bad Character Heuristic of Boyer Moore Algorithm ''' m = len ( pat ) n = len ( txt ) # create the bad
character list by calling # the preprocessing function badCharHeuristic() # for given pattern badChar =
badCharHeuristic ( pat , m ) # s is shift of the pattern with respect to text s = 0 while ( s <= n - m ): j = m
- 1 # Keep reducing index j of pattern while # characters of pattern and text are matching # at this shift s
while j >= 0 and pat [ j ] == txt [ s + j ]: j -= 1 # If the pattern is present at current shift, # then index j will
become -1 after the above loop if j < 0 : print ( "Pattern occur at shift = { } " . format ( s ) ) ''' Shift the
pattern so that the next character in text aligns with the last occurrence of it in pattern. The condition
s+m < n is necessary for the case when pattern occurs at the end of text ''' s += ( m - badChar [ ord ( txt
[ s + m ] )] if s + m < n else 1 ) else : ''' Shift the pattern so that the bad character in text aligns with the
last occurrence of it in pattern. The max function is used to make sure that we get a positive shift. We
may get a negative shift if the last occurrence of bad character in pattern is on the right side of the
current character. ''' s += max ( 1 , j - badChar [ ord ( txt [ s + j ] )] ) # Driver program to test above
function def main (): txt = "ABAAABCD" pat = "ABC" search ( txt , pat ) if __name__ == '__main__':
main () # This code is contributed by Atul Kumar # (www.facebook.com/atul.kr.007) C# /* C# Program
for Bad Character Heuristic of Boyer Moore String Matching Algorithm */ using System ; public class
AWQ { static int NO_OF_CHARS = 256 ; // A utility function to get maximum of two integers static int
max ( int a , int b ) { return ( a > b ) ? a : b ; } // The preprocessing function for Boyer Moore's // bad
character heuristic static void badCharHeuristic ( char [] str , int size , int [] badchar ) { int i ; // Initialize
all occurrences as -1 for ( i = 0 ; i < NO_OF_CHARS ; i ++ ) badchar [ i ] = - 1 ; // Fill the actual value of
last occurrence // of a character for ( i = 0 ; i < size ; i ++ ) badchar [ ( int ) str [ i ] ] = i ; } /* A pattern
searching function that uses Bad Character Heuristic of Boyer Moore Algorithm */ static void search (
char [] txt , char [] pat ) { int m = pat . Length ; int n = txt . Length ; int [] badchar = new int [
NO_OF_CHARS ] ; /* Fill the bad character array by calling the preprocessing function
badCharHeuristic() for given pattern */ badCharHeuristic ( pat , m , badchar ); int s = 0 ; // s is shift of
the pattern with // respect to text while ( s <= ( n - m ) ) { int j = m - 1 ; /* Keep reducing index j of pattern
while characters of pattern and text are matching at this shift s */ while ( j >= 0 && pat [ j ] == txt [ s + j ] )
j -- ; /* If the pattern is present at current shift, then index j will become -1 after the above loop */ if ( j < 0
) { Console . WriteLine ( "Patterns occur at shift = " + s ); /* Shift the pattern so that the next character in
text aligns with the last occurrence of it in pattern. The condition s+m < n is necessary for the case
when pattern occurs at the end of text */ s += ( s + m < n ) ? m - badchar [ txt [ s + m ] ] : 1 ; } else /* Shift
the pattern so that the bad character in text aligns with the last occurrence of it in pattern. The max
function is used to make sure that we get a positive shift. We may get a negative shift if the last
occurrence of bad character in pattern is on the right side of the current character. */ s += max ( 1 , j -
badchar [ txt [ s + j ] ] ); } } /* Driver program to test above function */ public static void Main () { char [] txt
= "ABAAABCD" . ToCharArray (); char [] pat = "ABC" . ToCharArray (); search ( txt , pat ); } } // This
code is contributed by PrinciRaj1992 JavaScript < script > /* Javascript Program for Bad Character
Heuristic of Boyer Moore String Matching Algorithm */ let NO_OF_CHARS = 256 ; // A utility function to
get maximum of two integers function max ( a , b ) { return ( a > b ) ? a : b ; } // The preprocessing
function for Boyer Moore's // bad character heuristic function badCharHeuristic ( str , size , badchar ) { //
Initialize all occurrences as -1 for ( let i = 0 ; i < NO_OF_CHARS ; i ++ ) badchar [ i ] = - 1 ; // Fill the
actual value of last occurrence // of a character (indices of table are ascii and values are index of
occurrence) for ( i = 0 ; i < size ; i ++ ) badchar [ str [ i ] . charCodeAt ( 0 ) ] = i ; } /* A pattern searching
function that uses Bad Character Heuristic of Boyer Moore Algorithm */ function search ( txt , pat ) { let
m = pat . length ; let n = txt . length ; let badchar = new Array ( NO_OF_CHARS ); /* Fill the bad
character array by calling the preprocessing function badCharHeuristic() for given pattern */
badCharHeuristic ( pat , m , badchar ); let s = 0 ; // s is shift of the pattern with // respect to text // there

```

are  $n-m+1$  potential alignments while (  $s \leq (n - m)$  ) { let  $j = m - 1$  ; /\* Keep reducing index  $j$  of pattern while characters of pattern and text are matching at this shift  $s$  \*/ while (  $j \geq 0$  &&  $pat[j] == txt[s + j]$  )  $j--$  ; /\* If the pattern is present at current shift, then index  $j$  will become -1 after the above loop \*/ if (  $j < 0$  ) { document . write ( "Patterns occur at shift = " +  $s$  ); /\* Shift the pattern so that the next character in text aligns with the last occurrence of it in pattern. The condition  $s+m < n$  is necessary for the case when pattern occurs at the end of text \*/  $txt[s+m]$  is character after the pattern in text  $s += (s + m < n) ? m - badchar[txt[s + m].charCodeAt(0)] : 1$  ; } else /\* Shift the pattern so that the bad character in text aligns with the last occurrence of it in pattern. The max function is used to make sure that we get a positive shift. We may get a negative shift if the last occurrence of bad character in pattern is on the right side of the current character. \*/  $s += \max(1, j - badchar[txt[s + j].charCodeAt(0)])$  ; } } /\* Driver program to test above function \*/ let  $txt = "ABAAABCD"$  . split ( "" ); let  $pat = "ABC"$  . split ( "" ); search (  $txt$  ,  $pat$  ); // This code is contributed by unknown2108 < /script> Output pattern occurs at shift = 4 Time Complexity :  $O(m*n)$  Auxiliary Space:  $O(1)$  The Bad Character Heuristic may take  $O(m*n)$  time in worst case. The worst case occurs when all characters of the text and pattern are same. For example,  $txt[] = "AAAAAAAAAAAAAAAAAAAA"$  and  $pat[] = "AAAAA"$ . The Bad Character Heuristic may take  $O(n/m)$  in the best case. The best case occurs when all the characters of the text and pattern are different. Boyer Moore Algorithm | Good Suffix heuristic Comment Article Tags: Article Tags: Strings Pattern Searching DSA