Boyer Moore

```
T = XPBCTBXABPQXCTBPQ
        X||||
P =   TPABXAB
        |     X
        TPABXAB        // The non matching character existed on position 1 
of the pattern
               TPABXAB // The non matching character did not exist
```

Bad Character Rule: Every time a match fails the algorithm looks in the pattern if the character that didn't match exists in the pattern. If yes shift the pattern to align the non-matching character with the corresponding one in the pattern.

Before the algorithm some preprocessing is necessary to find out the information what character is on what position. We build a table with all characters in the text??? and its right most position:

Character	Position
T	1
P	2
A	6
B	7
X	5
Q	0

When matching this table is used to find an occurence of the non-matching character.

Function R(i,C) finds the rightmost position i of character C inn pattern Examples:

- R(4,A) = 3
- R(7,A) = 6

How is the performance of this algorithm?

Three different approaches:

- trivial: poor running time
- better: |P|*|E|
- best: |P|

Better Approach: Store the position of all characters occuring in the patter and the position in the pattern:

Character	Position
T	1
P	2
A	3,6
B	4,7
X	5

Question: Would it take too much time to go through the list?

- At most we spent twice as much as characters in the pattern.
- *While matching we're doing at least the matching work, so the time is not wasted*

Works very well for large alphabets and infrequent Characters. Question: "Can we tweak the Boyer Moore algorithm to do well in all situations?"

1. Approach: After a mismatch occurs: Can we find a position to which we can shift the pattern to so that it matches the already observed character sequence?

Example:

```
ooooooooooooooooooooooooABCDEoooooooooooo
                        X||||
            oooooBCDEooooBCDE
                         ||||
                    oooooBCDEooooBCDE
```

Good Suffix Rule: After we find a mismatch we want to find a sequence in the pattern that matches the sequence in the text that was just observed but which has a different character to the left (since it previously caused the mismatch)

The running time is 4*n. In general, however, the performance is much better but it is difficult how the bad character rule can enhance performance.

For every i we store a value L(i), which is the rightmost position in P s.t. P[i..n] matches suffix of P[1..L(i)]

```
oooooooABCDEFGoooooooooooZBCDEFG
             |            |
            L(i)          i
```

L'(i) -> L(i) and P[i-?) != P[L(i)-|P|+i-2]]

Use approach of Z-Boxes to enhance performance.

L(K)=L(n-Z(i)+1)

How do we find the longest prefix/suffix that mathches using the Z-Values?

P(i) = length of longest prefix of pattern P that matches suffix of pattern P.

We are looking for a Z[j]???

```
1    k    h                    m
oooooooooooooooooooooooooooooooo
          |
      ooooooo
      1   i n
```

R(T[h]) : Tells us how much we can shift

R(T[h]) -> K' = K + i - R(T[h])

Take max of

- K' tells us what to do with the bad character rule // Jump to the position in the pattern that matches the section of the text
- K = *K + n - L'(i) + 1 // The sequence in the pattern that failed to match occurs in the beginning of the pattern*

```
k=n
while k<=m
   i=n
   h=k
   while i>0 AND P[i]=T[h] // i should not go beyond the end of the pattern
and ...
      i --j h=j
   if i= 0 -> match // perfect match
```