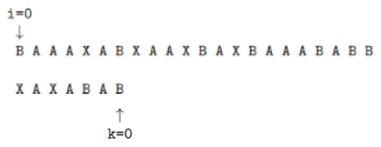
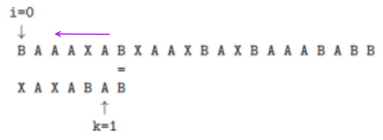
Boyer-Moore - introduction

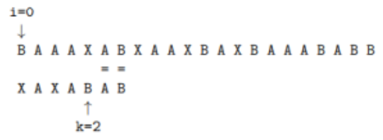
Boyer-Moore searches for the matchings of the pattern (P[0..m-1]) in the text (T[0..n-1]) at different starting positions i (just as KMP and the Naive search algorithm). It is different because we test for the matching going on both the pattern and the portion of the text from right to left.

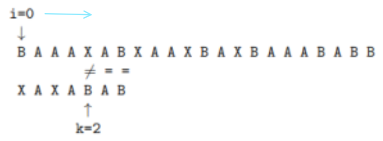
i = the starting position of the current possible match in the text T (i is moved from left to right)
k = the position of the currently compared character in the pattern (k is moved from right to left)

E.g.:





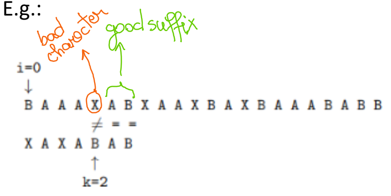




In case of a mismatch between the current characters in the text and in the pattern, the index i is incremented using two types of information:

1. The bad character (= the first character from the text that didn't match) => the bad character rule
2. The good suffix (= the part from the text that matched the pattern) => the good suffix rule

E.g.:

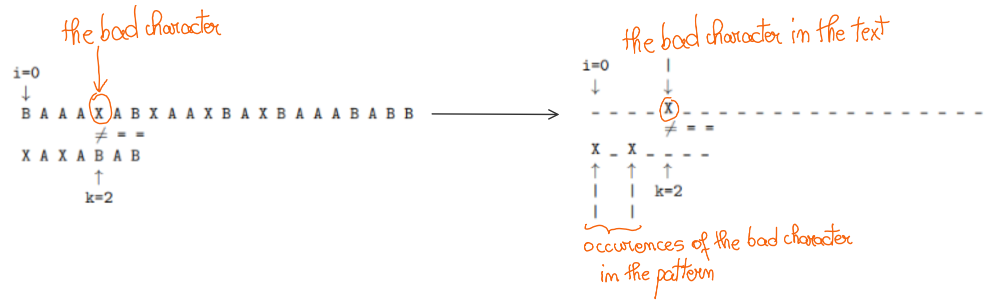


We choose to increment i by the biggest offset between the offset given by the bad character rule and the offset given by the good suffix rule

Boyer-Moore - Bad character rule

The bad character rule tells us the offset by which to increment index i,
such that there is not a mismatch between the bad character (from the text) and the pattern.

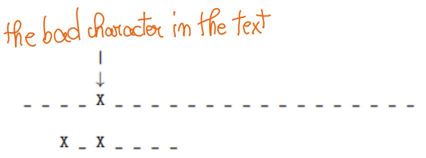
We only keep track of:
- the position of the bad character in the text
- the occurrences of the bad character in the pattern


the bad character
i=0
B A A A X A B X A A X B A X B A A A B A B B
X A X A B A B
k=2
the bad character in the text
occurrences of the bad character in the pattern

The bad character rule (general case):

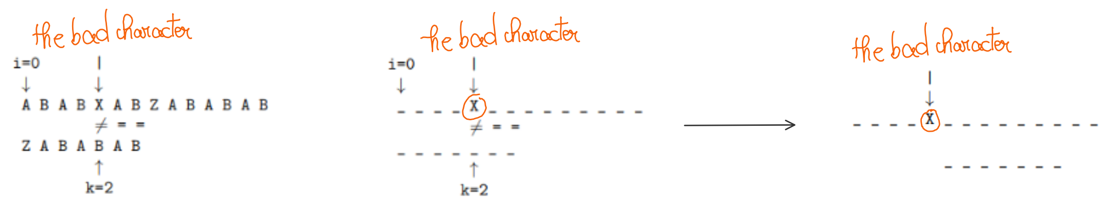
We move the pattern such that the last occurrence of the bad character in the pattern is aligned with the position of the bad character in the text

(we don't want to miss other matches)



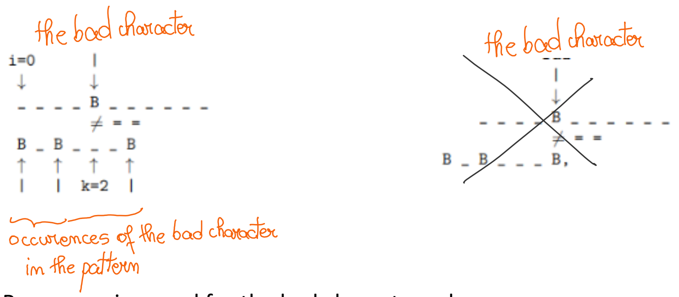
Corner case 1:

If the pattern doesn't contain the bad character we move the pattern after the bad character.
(We can also say that the last occurrence of the bad character in the pattern is on position -1).



Corner case 2:

If the bad character rule moves the pattern in the opposite direction (so the algorithm is regressing, not advancing),
we just move the pattern forward with one position.



Preprocessing used for the bad character rule:

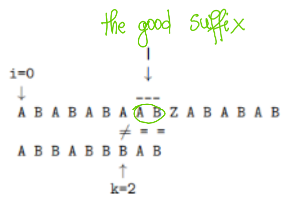
We precompute the values for BC[0... l-1], where l = size of alphabet
BC[c] = the last position where character c appears in the pattern P

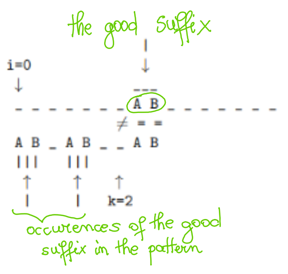
```
for (int i = 0; i < l; ++i) {
  BC[i] = -1;
}

for (int i = 0; i < m; ++i) {
  BC[P[i]] = i;
}
```

Boyer-Moore - Good suffix rule

The good suffix rule tells us the offset by which to increment index i, such that the good suffix continues to be matched.

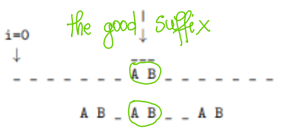
We only keep track of:
- the position of the good suffix in the text
- the occurrences of the good suffix in the pattern



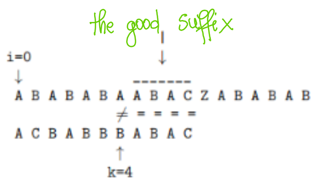


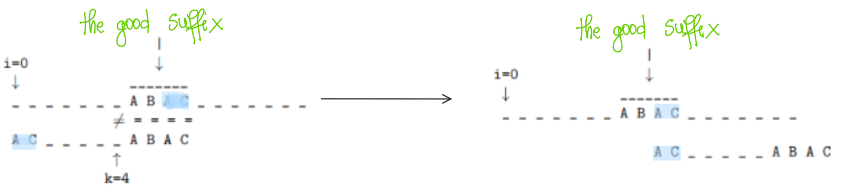
The good suffix rule (general case):

We move the pattern such that the second to last occurrence of the good suffix in the pattern is aligned with the position of the good suffix in the text

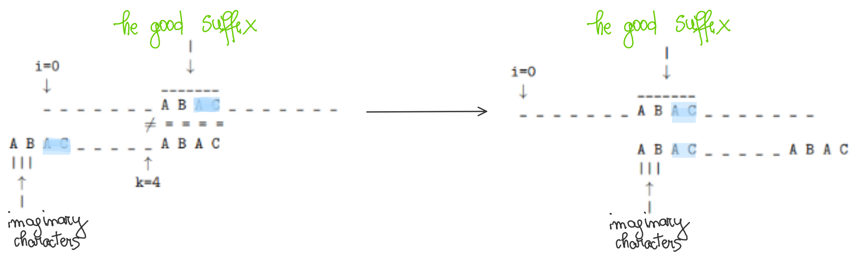


Corner case 1:
If the pattern doesn't contain the good suffix more than once, we move the pattern to align the biggest prefix of the pattern, that is also a suffix of the good suffix.





This can also be visualized as:



Preprocessing to compute the good suffix rule:

We determine GS[0..m-1], where:

GS[i] = the second to last position in the pattern where P[i..m-1] appears

E.g.:

```
i     =  0  1  2  3  4  5  6  7  8
P[i]  =  A  B  A  B  A  C  A  B  A
GS[i] = -6 -5 -4 -3 -2 -1  2  3  6
```

```
i     =  0  1  2  3  4  5  6  7  8
P[i]  =  A  B  A  B  A  C  A  B  A
GS[i] = -6 -5 -4 -3 -2 -1  2  3  6
                     A  C  A  B  A
                    -2 -1  0  1  2
```

Obs: There is an index in the pattern from where the suffixes do not appear anymore in the pattern (except for the last position).

A, BA, ABA appear in the pattern more than once -> positive indices (general case)
CABA, ACABA.. etc., do not appear in the pattern more than once -> negative indices (corner case 1)

We compute GS in 2 steps, using the prefix function f from KMP:

1. <u>Computing the negative values for GS</u>

```
i     =  0  1  2  3  4  5 | 6  7  8
P[i]  =  A  B  A  B  A  C | A  B  A
f[i]  = -1  0  0  1  2  3 | 0  1  2  3

GS[i] = -6 -5 -4 -3 -2 -1 | 0  1  2
```

f[m]=3

not final GS values

```
for (int i = 0; i < m; ++i) {
  gs[i] = f[m] - (m - i);
}
```

length of the suffix P[i..m-1]

length of the biggest border of the entire pattern P[0...m-1]

f[i] = the length of the biggest border of P[0…i-1]

2. <u>Computing the positive values for GS</u>

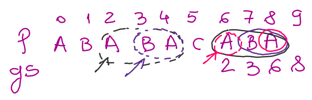
```
i     =  0  1  2  3  4  5  6  7  8
R[i]  =  A  B  A  C  A  B  A  B  A      //P reversed
g[i]  = -1  0  0  1  0  1  2  3  2  3   //failure function for R

P[i]  =  A  B  A  B  A  C  A  B  A
h[i]  =  3  2  3  2  1  0  1  0  0 -1   //g reversed
```

```
for (int i = 0; i < m; ++i) {
  int len = h[i];
  gs[m - len] = i;
}
```

h[i] = length of the biggest prefix of P[i..m-1] that is also a suffix

GS[m-h[i]] = i (i is the second to last position where P[m-h[i] .. m-1] appears)
(GS[i] = the second to last position in the pattern where P[i..m-1] appears)

(e.g. i=0, GS[6] = 0
  i=1, GS[7] = 1
  i=2, GS[6] = 2
  i=3, GS[7] = 3
  i=4, GS[8] = 4
  i=5, GS[9] = 5
  i=6, GS[8] = 6
  i=7, GS[9] = 7
  i=8, GS[9] = 8

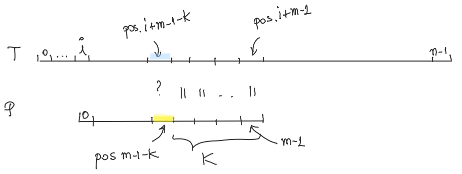


Boyer-Moore - Implementation

Algorithm:

i = the starting position of the current possible match in the text T (i is moved from left to right)
k = the number of matched characters (starting from the right side)

```
i = 0;
k = 0;                                       //k<m ⇔ we haven't matched the whole pattern
                                             // i <= n- m  ⇔ we still have chances to match the pattern
 while (k < m && i <= n - m)
 {
   if (T[i + m - 1 - k] == P[m - 1 - k]) { // we compare the current ch in the pattern with the current ch in the text
     k++;                                   // if equal, move to the left
   } else {
     shiftbc = m - k - 1 - BC[T[i + m - 1 - k]];   // else, determine shiftbc & shiftgs offsets
     shiftgs = m - k - GS[m - k];
     i += max(shiftbc, shiftgs);                  // move the pattern forward by the maximum offset
     k = 0;                                       // start comparing ch. from the right side
   }
 }
```

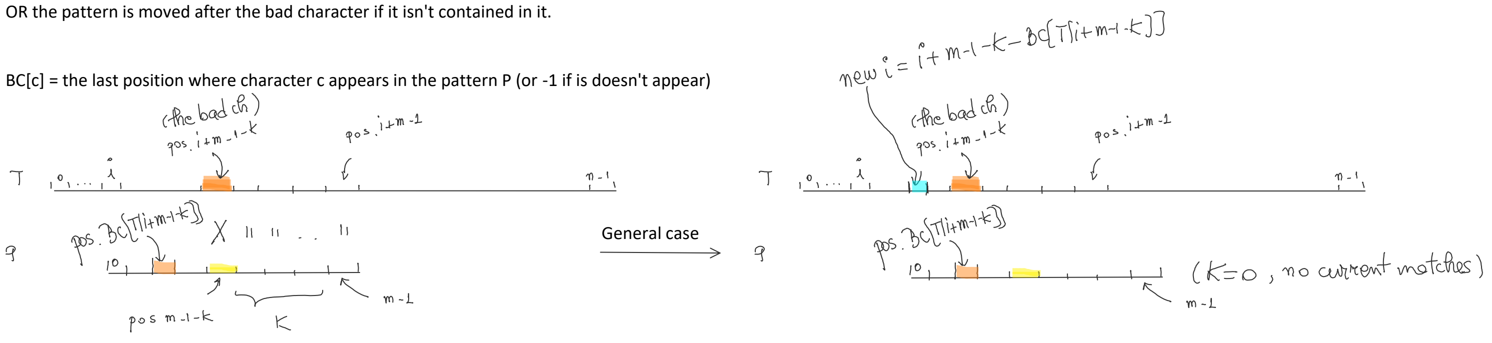


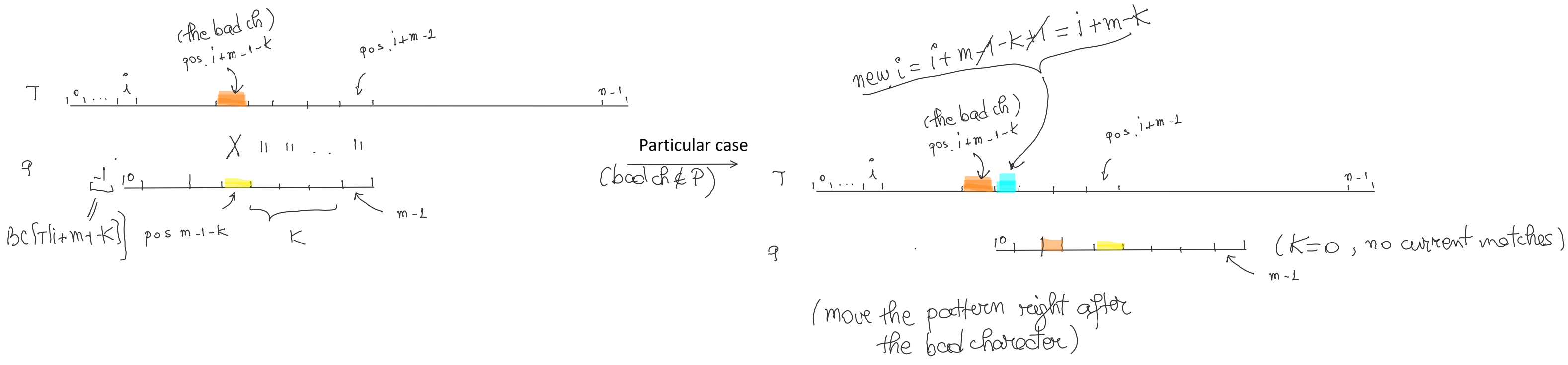
Explanation for shiftbc (=offset given by the Bad Character rule):

Bad character rule:

We move the pattern such that the last occurrence of the bad character in the pattern is aligned with the position of the bad character in the text
OR the pattern is moved after the bad character if it isn't contained in it.

BC[c] = the last position where character c appears in the pattern P (or -1 if is doesn't appear)





Explanation for shiftgs (=offset given by the Good Suffix rule)::

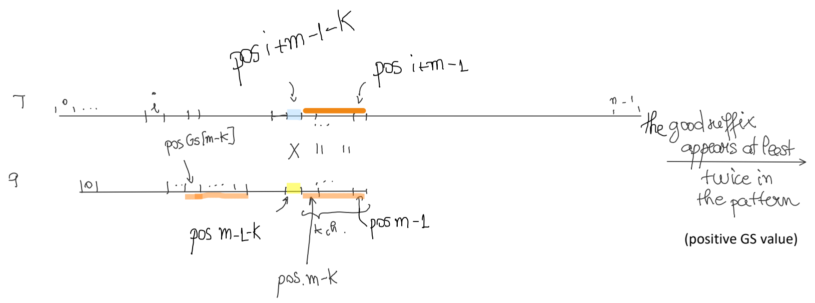
Good Suffix rule:

We move the pattern such that the second to last occurrence of the good suffix in the pattern is aligned with the position of the good suffix in the text
OR if the pattern doesn't contain the good suffix more than once, we move the pattern to align the biggest prefix of the pattern, that is also a suffix of the good suffix

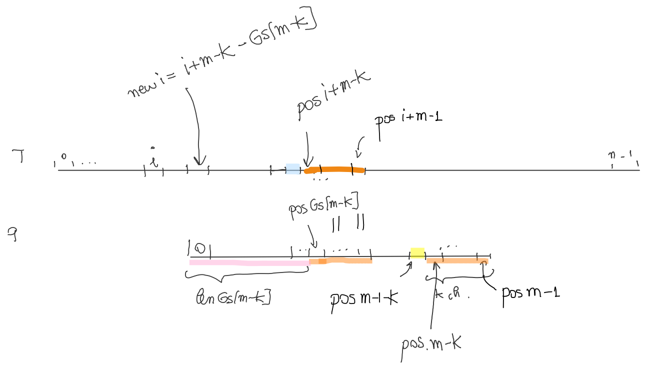
GS[i] = the second to last position in the pattern where P[i..m-1] appears

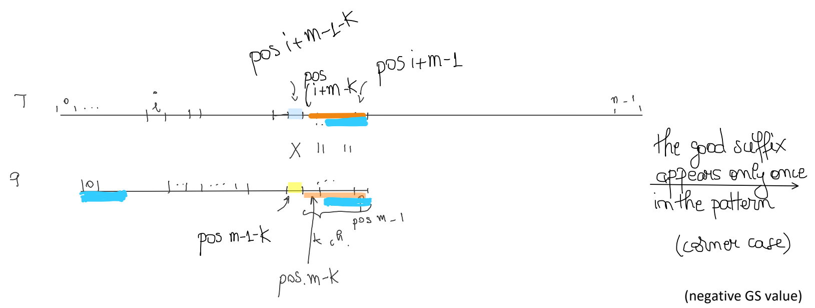
(GS[m-k] = the second to last position in the pattern where P[m-k..m-1] appears)

Boyer Moore - Implementation (drawings)

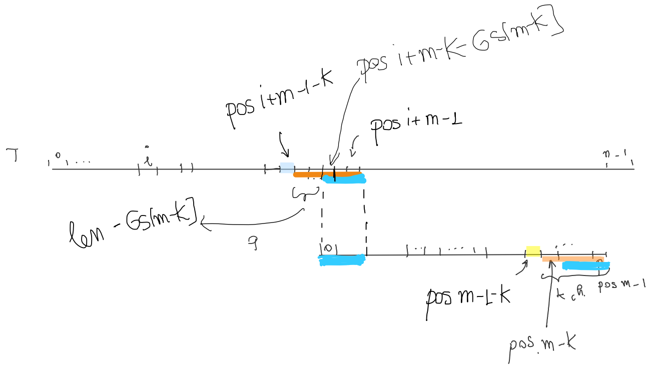


(positive GS value)





(negative GS value)



pos i+m-K-GS[m-K]
pos i+m-1-K
pos i+m-1
T
0, ...
i
n-1
len - GS[m-K]
q
0
pos m-1-K
k ch.
pos m-1
pos.m-K