Pattern matching with the Burrows-Wheeler Transform

Student: Meinhard Dam, 201303531
Supervisor: Christian Storm Pedersen

April 23, 2019

Abstract

In Bioinformatics, researchers have a need to analyze DNA, RNA, and protein sequences. A method to perform this analysis is by searching for specific patterns in the sequence. The project investigates three distinct exact pattern matching algorithms - the Naïve algorithm, the Knuth-Morris-Pratt algorithm, and an algorithm which makes use of the Burrows-Wheeler Transform. The report calls the last of the three algorithms BW-Match. Even though these three algorithms already are well-established, it is always useful to acquire more empirical data on how algorithms work in practice. The project implements the three algorithms and uses the implementations to run three distinct experiments - one experiment that verifies the correctness of the implementations, one experiment which indicates that the runtime of the implementations in practice match the theoretical runtime of the algorithms, and one experiment which runs the implementations of the KMP algorithm and the BW-Match algorithm side by side with the same text and patterns to investigate how many short patterns one would need to search for in a text, for it to be worth it to construct a C-table and an O-table for the BW-Match algorithm. The third and last experiment concludes that it would be worth it to construct a C-table and an O-table for the BW-Match algorithm, if one wanted to search for more than 10 short patterns in a text.
# Contents

1 Introduction 1

2 The Algorithms 3
   2.1 Exact Pattern Matching 3
   2.2 The Naive Algorithm 3
   2.3 Knuth-Morris-Pratt 4
      2.3.1 Border Arrays 4
      2.3.2 The KMP Algorithm 6
   2.4 Burrows-Wheeler Match 7
      2.4.1 Suffix Arrays 7
      2.4.2 The C-table 8
      2.4.3 The O-table 8
      2.4.4 The BW-Match Algorithm 9

3 Implementation 13
   3.1 The Naive Algorithm 13
   3.2 The KMP Algorithm 13
   3.3 BW-Match 15
      3.3.1 Suffix Array 15
      3.3.2 The C-table 16
      3.3.3 The O-table 17
      3.3.4 The BW-Match Algorithm 18

4 Experiments 21
   4.1 The Three Experiments 21
   4.2 Verification Experiment 21
   4.3 Runtime vs Complexity Experiment 22
      4.3.1 The Naive Algorithm 23
      4.3.2 The KMP Algorithm 24
      4.3.3 The BW-Match Algorithm 26
   4.4 KMP vs BW-Match Experiment 28

5 Conclusion 33

References 34
1 Introduction

In Bioinformatics, researchers have a need to analyze DNA, RNA, and protein sequences. A method to perform this analysis is by searching for specific patterns in the sequence. One can use a pattern matching algorithm to identify the locations of a pattern’s occurrences in a DNA, RNA, or protein sequence. There are multiple pattern matching algorithms available, where some are more useful in some scenarios than others. There are also some that perform exact pattern matching while others perform approximate pattern matching. The scope of this project is to investigate three distinct exact pattern matching algorithms - One algorithm which takes a straightforward naive approach, one called the Knuth-Morris-Pratt algorithm, and one which uses the Burrows-Wheeler transform. Since the algorithms can also be used for other kinds of strings than DNA, RNA, and protein sequences, the terminology used in the report will be that the algorithms are used to search for a pattern occurring in a text.

These are all already well-established algorithms, but it is always useful to acquire more empirical data on how algorithms work in practice. This is why all three algorithms will be implemented in this project, and experiments will be run on them. First though, the report will contain information about how the algorithms work, and why they are sound. Both the Knuth-Morris-Pratt algorithm and the one using the Burrows-Wheeler transform, which we are calling the Burrows-Wheeler Match, or BW-Match for short, have multiple parts that need to be explained in order for them to make sense. The Knuth-Morris-Pratt algorithm, also known as KMP, uses a border array, which needs to be explained, while BW-Match has a preprocessing, where a suffix array and two tables called the C-table and the O-table are used.

The report can be an additional explanation of the algorithms and be added as additional empirical evidence that they provide correct result and in practice match the theoretical runtime. Additionally, the report includes a section, which compares the KMP algorithm and BW-Match to see when it is worth it to construct the C- and O-tables and use BW-Match, instead of just using KMP. This can be useful when choosing which algorithm to use in what scenario.

To create the new empirical data and to make it possible to compare the two algorithms KMP and BW-Match, the three algorithms, the Naive algorithm, KMP, and BW-Match will be implemented. When the algorithms have been implemented, three separate experiments will be run on the implementations. The first will assess if the implementations are producing correct outputs with various inputs, i.e., it will verify the correctness of the implementations. The second experiment will run the algorithms to see how the runtimes of the implementations compare with the complexity of the algorithms. The third and last experiment will take the implementation of KMP and the implementation of BW-Match and run them side by side with the same text and patterns to see how many patterns at a specific length you need to search for, for it to be worth it to construct the C-table and the O-table at different text lengths. It makes sense to do the third experiment this way since the preprocessing of BW-Match is only done once per text, while the preprocessing for KMP is needed for every pattern searched for in the text.

What might the result be? How many patterns will it take? The report will give you the answer.
2 The Algorithms

2.1 Exact Pattern Matching

Exact Pattern Matching is comparatively simple and does not lose any data, since you find the locations of the exact wanted patterns.

An Exact Pattern Matching algorithm takes a pattern and a text as an input and outputs the indexes of the text, where the pattern has been found. The characters of the pattern must be from the same alphabet as those of the text.

2.2 The Naive Algorithm

A naive way to locate the indexes for where the input pattern occurs in the input text, is to start matching the characters of the pattern with the characters of the text one by one. For every index of the text, the algorithm checks if the first character in the text from the current index is the same as the first character of the pattern, then the second character of the text and the second character of the pattern and so on, until there is a mismatch or it has gone through the whole pattern. If the character in the text and the character in the pattern do not match, the algorithm stops comparing characters and starts over from the next index of the text. If the algorithm manages to compare all the characters of the text from an index and forward the length of the pattern, with the characters of the pattern without the occurrence of a mismatch, the algorithm will report the relevant index as a match, and continue to the next index of the text.

A pseudo code for this algorithm could be [1]:

```
for i = 1 to length(x):
    for j = 1 to m+1:
        if x[i+j-1] != p[j]: break
    if j == m+1: report i as a match
```

Here $x$ is the text, $i$ is the index of the text, $j$ is the index of the current character compared, and $m$ is the length of the pattern. The running time of the naive algorithm is $O(n \times m)$.

What makes the algorithm naive is the fact that it sometimes compares strings that we know will not match. If the pattern is something like "aab", where not every character is the same, we know that if the pattern matches at $i$, then it cannot match at $i+1$ or $i+2$. In this scenario, the Knuth-Morris-Pratt algorithm is more efficient.

2.3 Knuth-Morris-Pratt

To understand the Knuth-Morris-Pratt (KMP) algorithm, one needs a basic understanding of borders and border arrays.

A border of a string is any proper prefix \(^1\) of a string that equals the suffix of the same length. So for example in the string "abaabaab", there are the borders "ab" and "abaab" and the empty border. A border array $B$ is an array, where the entry $B[i]$ is the length of the longest border in the

\(^1\)A proper prefix is one, which does not include the last character. So "abaab" is not a proper prefix of "abaab". The longest proper prefix for "abaab" is "abaab".
prefix $x[1..i]$, where $x$ is a string. We can take the same string as before as an example. The border array of the string 'abaabaab' would be $\{0, 0, 1, 1, 2, 3, 4, 5\}$.

$$
x=gcggcacttaactgattagacagtaagac...
p=acttaactcgc
\uparrow
x=gcggcacttaactgattagacagtaagac...
p=acttaactcgc
\uparrow
p=acttaactcgc
\uparrow
$$

Figure 1: An example from slides of the course called string algorithms [1]. The green characters are the prefix of the pattern, which has been matched with the text, and the part of the text, which it matched with. The red characters are at the location in the pattern and the text, where a mismatch happened. The blue characters are the longest border of the matched prefix in the pattern.

Now thinking about the times in the previous section 2.2 where we tried to match strings, which we already knew would not match, there is an observation to be had. If a prefix of some length $l$ of a pattern matches with a text at some index $i$, then the pattern will definitely not match at any index of the text between and including the indexes $i$ and $i + l$, unless the prefix has a border. So if the pattern’s prefix which matched with the text at an index $i$ did not have a border, then the algorithm will continue scanning from where a mismatch was found in the text by matching it with the first character of the pattern and so on. But if the prefix does have a border, like it does in figure 1, then we again do not move the index of the text from where a mismatch was found, but the index of the pattern will not be moved all the way to the first character. It will instead be moved to the index with the value of the length of the border plus one, i.e. $B[l] + 1$. This is because we already know that the prefix border matches with the suffix border in the text, so we can continue matching the pattern with the text from right after the prefix border [1]. Because of this observation we will need to construct the border array for the pattern in the preprocessing of KMP.

### 2.3.1 Border Arrays

Now that we know what borders and border arrays are, and why we need a border array for KMP, let us go through how border arrays can be constructed. There are different ways to construct a border array. To construct one, one needs to find the length of the longest border for every prefix of a string, even the non-proper prefix, which is the full string. The pseudo code shown below deals with the task of finding the longest border of one of the prefixes or just some string: [2]:

```
B = 0
for i = 1 to n-1 do
    if x[1..i] = x[n-i+1..n] then
        B = i
return B
```
This pseudo code starts by initiating the border’s length as zero, then it evaluates one prefix at a
time by comparing it to the suffix of the same length to see if they match. If they do, then it has a
new longest border, which will be assigned to $B$. When the for loop has gone through all the proper
prefixes, $B$ will have the longest border length assigned to it.

This particular method of identifying the longest border length, is, however, not very time-efficient.
To find the longest border of one prefix you compare all of its prefixes with its suffixes of the same
length, and you have to do that with all the prefixes of the string that you want to construct a border
array for. That would make the running time $O(n^2)$. There is a faster way to do it, which is also the
one used in this project’s implementation.

There is a way to compute $B[i+1]$ from $B[i]$, where $B$ is the border array, and $i$ is the index.
$B[i]$ returns the length of the longest border of the prefix, which has a length corresponding to the
value of $i$. $B[1]$ is always 0, since the only border of a string of one character is the empty border.

One observation we can make is that if $B[i+1] = b$, where $b$ is a specific border length, then
$B[i] >= b - 1$, since if you remove the last character of the suffix border, then the resulting suffix
still matches with the prefix of its new length. So how can we compute $B[i+1]$ from $B[i]$? We
check if $x[B[i]+1] = x[i+1]$, where $x$ is the whole string, which we are making the border array
from. Also $x[i]$ returns the character of the string $x$ at the index $i$. So if $x[B[i]+1] = x[i+1]$, then
$B[i+1] = B[i] + 1$. However, if they do not match, then we try the same comparison with the second
longest border, which is also the longest border of the longest border, i.e. $B[B[i]]$ is the second longest
border of $x[1..i]$, where $B[i]$ is the longest. So now we can check if $x[B[B[i]]+1] = x[i+1]$, and if it is
true, then $B[i+1] = B[B[i]] + 1$, but if not, then we try the same with the third longest border and
so on. That idea is where the following pseudo code has its inspiration from [2]:

```
for i=1 to n-1 do
  b = B[i]
  while b > 0 and x[i+1] != x[b+1] do
    b = B[b]
  if x[i+1] = x[b+1] then
    B[i+1] = b+1
  else
    B[i+1] = 0
```

The algorithm starts by setting $B[1]$ to be 0 as we know it to be. Then it starts a loop that goes
through all the proper prefixes of a string. The loop starts off by assigning $b$ as the longest border
previously added to the border array, which is what each iteration will use to compute the next one
longest border. Then comes the previously mentioned part where we compare $x[B[i]+1]$ with $x[i+1]$.
Here we first check in the while-loop’s condition if they match or not. If they do not match, we take a
look at the second longest border of the current prefix, by setting $b$ as $B[b]$, which in the first iteration
would be the same as $B[B[i]]$. The while-loop continues like this until $x[b+1]$ and $x[i+1]$ match or
$b = 0$. When the while-loop is done, the algorithm must check if $x[b+1] = x[i+1]$, since it is not
obvious which condition stopped the while loop. If $x[b+1] = x[i+1]$, $B[i+1]$ is set as seen in the
pseudo code. If not, then $B[i+1]$ must be 0. This continues until the whole border array is constructed.

The running time of this algorithm is $O(n)$ plus the total time for the while-loop. The while-loop can
have pretty different total times depending on which texts the algorithm is used on.
2.3.2 The KMP Algorithm

Now that we know how to construct a border array, we can look at the actual KMP algorithm [1]:

Preprocessing:
build border array B
B'[1] = 0
for j = 2..m+1:
    B'[j] = B[j-1]+1

Main:
i = 1; j = 1
while i <= |x|-m+j:
    i,j = match(i, j, m)
    if j == m+1: report match at i-m
    if j == 1: i = i+1
    else: j = B'[j]

Help routine:
match(i, j, m):
    while x[i] == p[j] and j <= m:
        i = i+1
        j = j+1
    return i, j

It can be seen in the pseudo code that the preprocessing builds a border array and then modifies it afterwards. The border array is constructed from the pattern, not the text. When the border array B has been constructed, it is modified to make B'. B' is initiated with 0 as its first entry, and all the entries of B are then taken in order one by one, incremented by 1, and are added to B'. Because of that the size of B' is m + 1, where m is the length of the pattern. This procedure is a direct result of the observation previously mentioned, where when we have a mismatch, we let the index on the text stay at where the mismatch occurred, and change the index on the pattern to be B'[l] + 1, i.e. the location right after the prefix border of the longest border of the matched prefix of the pattern. The example used previously can be found in figure 1. This means that B' will give the location, where the index of the pattern should be moved to after a mismatch.

The help routine, called match, is where the actual matching happens. Its input is i and j, which are the indexes of the text and the pattern respectively, and m, which is the length of the pattern. Its output is the new i and j after the matching is done. If there is a mismatch, then i and j are the indexes of where the mismatch happened. If the whole pattern matches, then i is at the location right after the last matching character of the text, while j is m + 1, which is part of why we need the size of B' to be m + 1. The while-loop only increments the two indexes by one, while its condition checks if the two characters that the indexes point to match, and if j has not become greater than m, which would mean that the whole pattern has matched. That is why j ends as m + 1, when a full match has been found.

The main part of the algorithm uses B' and the help routine, to fulfill the idea, which was had with the previous observation. It just needs to initiate the two indexes at 1 and start a while-loop, which continues until it has gone through the whole text. The while loop starts by using the help
routine to do the matching, then using the output, it can know if there has been a match, by checking if \( j \) is \( m + 1 \), which was mentioned before to only have that value, when there has been a full match. After it either has or has not reported a match, it must assess which value should be assigned to \( j \) after that, i.e. where should the index of the pattern point to. If \( j \) is 1, which would mean that there was a mismatch at the first character of the pattern, then it just increments \( i \) by one, i.e. moves the pointer of the text one forward. This means that the first entry of \( B' \) is only used in the preprocessing, never in the main part of the algorithm. If \( j \) is greater than 1, then it uses \( B' \) to get the new value of \( j \). After the while loop is over, all the existing matches have been reported.

The running time of the KMP algorithm is \( O(n + m) \), which makes it a lot faster than the Naive algorithm - at least in a worst-case scenario.

### 2.4 Burrows-Wheeler Match

The algorithm chosen for comparison of the KMP algorithm is one that uses the Burrows-Wheeler Transform (BWT). When explaining the Naive algorithm and the KMP algorithm, the index for first the character of a string and the index for the first entry of an array has been 1. Now when explaining the BW-Match algorithm and its preprocessing, it will be 0 instead. For BW-Match we need to construct a suffix array, a C-table and an O-table. Each of these will be explained in the following.

#### 2.4.1 Suffix Arrays

A suffix array is an array of suffix indexes sorted alphabetically in regards to the suffixes. When making a suffix array, you can put a special character at the end, which will be the only one of its kind in the text. We will use \$\). The special character \$\) is set to be lexicographically smaller than any other character when constructing a suffix array. To give an example of a suffix array, the on for 'banana' would be \{6, 5, 3, 1, 0, 4, 2\} since the suffixes would be sorted like this:

<table>
<thead>
<tr>
<th>( i )</th>
<th>( S(i) )</th>
<th>( Suf(S(i)) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>$</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>a$</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>ana$</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>anana$</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>banana$</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>na$</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>nana$</td>
</tr>
</tbody>
</table>

Here \( i \) is the index of the suffix array, \( S \) is the suffix array, and \( Suf(i) \) is the suffix \( x[i..n - 1] \), where \( x \) is a string, \( n \) is the length of that string, and \( i \) is the index of the string.

A suffix array is useful, because it can be a good tool, when doing efficient exact pattern matching. An observation that can be made is that if a pattern \( w \) occurs at \( i \) in a text \( x \), then \( w \) is a prefix of the suffix \( Suf(i) \). Also, suffixes with a common prefix form a consecutive block in the suffix array bounded by \( L(w) = \min\{k | w \text{ is a prefix of } Suf(S(k))\} \) and \( R(w) = \max\{k | w \text{ is a prefix of } Suf(S(k))\} \).

What is interesting about this is, that if \( L(w) \) and \( R(w) \) exist for a pattern \( w \), which would mean that \( w \) is the prefix of some suffix of \( x \), then \( w \) occurs at the positions \( \{S(k) | L(w) \leq k \leq R(w)\} \) [3]. So if you were searching for the pattern 'ana', then you could see that \( L(ana) = 2 \) and \( R(ana) = 3 \), so the
occurrences of 'ana' in 'banana' would be \{1, 3\}.

Let us also observe that \(L(w)\) and \(R(w)\) divide the suffix array into 3 parts: \(LT(w) = \{S(k)|0 \leq k < L(w)\}\), \(EQ(w) = \{S(k)|L(w) \leq k \leq R(w)\}\), and \(GT = \{S(k)|R(w) < k \leq n - 1\}\). So with that we can see that we need to find \(EQ(w)\) to know where \(w\) occurs in \(x\), i.e. locate \(L(w)\) and \(R(w)\). The key idea in locating \(L(w)\) and \(R(w)\) for a specific \(w\), is that if we know \(L(w)\) and \(R(w)\) for some \(w\), then we can compute \(L(aw)\) and \(R(aw)\) for any character \(a\), e.g. \(L(nana)\) from \(L(ana)\). This means that it is possible to start with the empty string \(\varepsilon\), since \(L(\varepsilon) = 0\) and \(R(\varepsilon) = n - 1\), and then compute \(L(w)\) and \(R(w)\), by iteratively adding one character at a time to the empty string. So we would first add the character \(w[m - 1]\), then \(w[m - 2]\), then \(w[m - 3]\) and so on, until we have the full \(w\). But how do we compute \(L(aw)\) and \(R(aw)\) efficiently? To answer this question, having an understanding of C- and O-tables is a powerful tool.

### 2.4.2 The C-table

The C-table can be computed in time \(O(n)\) by counting the occurrences of each symbol and summing them together in the right way. An entry in the C-table \(C(a)\), where \(a\) is some character, is the number of characters in \(x[0..n - 1]\) that are lexicographically smaller than \(a\). That is excluding '\$' [3]. The C-table for 'banana' would e.g. be:

\[
x[0..6] = \text{banana}\$
\]

\[
C(\ ) = \{0, 0, 3, 4\}
\]

Since '\$' is not included when counting characters in the C-table, the entry for "a" is 0, just like the entry for "\$". The entry for "b" is 3 since there are 3 a's in 'banana', while the entry for "n" is 4 since there are 3 a's and 1 'b' in 'banana'. From this example it can also be seen that you can find the entry for 'n' by counting the number of b's in 'banana' and adding that number to the entry for 'b'.

### 2.4.3 The O-table

So now that we know what a C-table is, what does an O-table look like? An entry of the O-table \(O(a, i)\) is the number of times the character \(a\) occurs immediately to the left of one of the \(i\) lexicographically smallest suffixes \(Suf(S(0)), ..., Suf(S(i))\). This can be computed in time \(O(|\Sigma|n)\), where \(\Sigma\) is the alphabet, by using the BWT of \(x\).

The BWT is a string \(b\) which is a permutation of \(x\), where \(b[i] = x[S(i) - 1]\) if \(S(i) > 0\), and \(b[i] = \$\) if \(S(i) = 0\). So the BWT of "banana\$" is "anb$aa". Look to the suffix array of "banana" at 2.4.1 for confirmation. The BWT is useful for getting the 'immediately to the left of' part of the definition for the O-table when constructing it, which is why it is used for the construction.

When the BWT of the text has been constructed, it can be used to make every entry of the O-table by counting the occurrences of a certain character in a certain prefix of the BWT. If \(O(a, i)\) is an entry in the O-table, where \(a\) is a character and \(i\) is an index, then the entry is found by counting the occurrences of the character \(a\) in the prefix \(b[0..i]\) where \(b\) is the BWT. The O-table for "banana" would look like the following:
\[ b[0..6] = \text{annb}a \]

\[
\begin{array}{ccccccc}
 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
0(\$, ) & = & [0, 0, 0, 1, 1, 1] \\
0(a, ) & = & [1, 1, 1, 1, 1, 2, 3] \\
0(b, ) & = & [0, 0, 0, 1, 1, 1, 1] \\
0(n, ) & = & [0, 1, 2, 2, 2, 2, 2] \\
\end{array}
\]

2.4.4 The BW-Match Algorithm

Now that we know how what C-tables, and O-tables are, and how they can be constructed, we can continue where we left off at section 2.4.1. The algorithm which uses the C-table and the O-table first searches for the \( w[m - 1] \) interval, then the \( w[m - 2..m - 1] \) interval and so on, until it gets to the full \( w[0..m - 1] \) interval. Because of that nature, it is called backward search \[4\], even though it is called BW-Match in this report. This way the algorithm can find \( L(w) \) and \( R(w) \), which bound the suffixes that have the pattern \( w \) as a prefix.

As mentioned before, the suffix array can be separated into three intervals: \( LT(w) \), \( EQ(w) \), and \( GT(w) \). When moving the \( L(w) \) pointer to \( L(aw) \), it uses information from the \( LT(w) \) section. When moving the \( R(w) \) pointer to \( R(aw) \), it uses information from both the \( LT(w) \) and the \( EQ(w) \) section.

We can e.g. try to find the pattern 'ana' in 'banana'. Here is the suffix array for 'banana' again, but with the BWT included:

\[
\begin{array}{cccc}
i & S(i) & \text{Suf}(S(i)) & b[i] \\
0 & 6 & $ & a \\
1 & 5 & a$ & n \\
2 & 3 & ana$ & n \\
3 & 1 & anana$ & b \\
4 & 0 & banana$ & $ \\
5 & 4 & na$ & a \\
6 & 2 & nana$ & a \\
\end{array}
\]

This illustration of the suffix array with the BWT will be a guide in the explanation of the BW-Match algorithm.

Let us look at the part of the algorithm’s process, where it computes \( L("na") \) and \( R("na") \) from \( L("a") \) and \( R("a") \), and the part where it computes \( L("ana") \) and \( R("ana") \) from \( L("na") \) and \( R("na") \), since they are more interesting than the part, where the algorithm computes \( L("a") \) and \( R("a") \) from \( L("a") \) and \( R("a") \).

When the \( L \) and \( R \) for 'a' have been computed, they are \( L("a") = 1 \) and \( R("a") = 3 \), which means that \( EQ("a") = \{S(1), S(2), S(3)\} \), and \( LT("a") = S(0) \). After that the algorithm will try to compute \( L("na") \) and \( R("na") \). To compute the first of the two, it will use the formula:

\[
L("na") = C("n") + O("n", L("a") - 1) + 1
\]

where \( C \) is the C-table, and \( O \) is the O-table. The part of the equation that uses the C-table and the +1 will help the algorithm to move the \( L \) pointer to the first suffix that starts with 'n', because \( C("n") \) returns the number of characters in the text, which are lexicographically smaller than 'n' excluding...
'$. That number is 4, since there are three a's and one 'b' in 'banana'. That moves the L to the suffix right before the one that starts with an 'n'. In this case that is the one which starts with 'b'. The +1 then moves the L to the first suffix of the suffix array, which has 'n' as its first character.

The part, which uses the O-table, will help the L move from there to the first of the suffixes with 'n' as their first character, which has 'a' as its second character. It performs this task by looking at how many suffixes in LT("a") have an 'n' directly to the left of it. The −1 makes sure that it does not include the suffix, which L("a") was pointing at. It moves the L from the first suffix of EQ("a") to the last suffix of LT("a"), so that we only look at the part of the O-table, which is associated with LT("a"). In this case the O-table returns 0, which is why L("na") = C("n") + O("n", L("a") − 1) + 1 = 4 + 0 + 1 = 5, which is exactly the first suffix with 'na' as a prefix. The formula for R, which is:

\[ R("na") = C("n") + O("n", R("a")) \]

uses the C-table in the same was as the previous formula. The part which uses the O-table looks at how many of the suffixes of sections LT("a") and EQ("a") have 'n' directly to the left of them, which is why R("a") is used as is instead of being added to or subtracted from. EQ("a") is included, so that the R pointer can be moved to the end of the suffixes which have 'a' as their second character. The part of the algorithm, which adds the last 'a' to w demonstrates this even better.

The part where the algorithm computes L("ana") and R("ana") from L("na") and R("na"), is more interesting since the resulting interval does not include all the suffixes that start with 'a'. The part of the formula with the C-table and the +1 again moves the new L to the first suffix that starts with 'a' by seeing that there are 0 characters lexicographically smaller than 'a', since "$" is excluded. Now, the interesting part is where the O-table part of the formula moves the new L one forward, since Suf(S(1)) = "a$" and Suf(S(2)) = "ana$". It looks at how many suffixes in LT("na") have an 'a' immediately to the left of it, since none of the suffixes in that section have 'na' as a prefix and are definitely lexicographically smaller than 'na'. The resulting number will be exactly the number of suffixes which start with 'a', but do not have 'na' directly to the right of said 'a' and are lexicographically smaller than 'ana'. In this case the number is 1, which is why it moves the new L one forward to exactly the first suffix in the suffix array, which has 'ana' as a prefix. The part of the formula for R("ana") which uses the C-table, again does the same as the one for L("ana"). Here the part which uses the O-table again includes the section associated with EQ("na") in addition to the one associated with LT("na"). The suffixes of LT("na") with 'a' to the left of it again move the new R one forward for the same reasons as previously, while the suffixes of EQ("na") move it two forward since both suffixes have and 'a' immediately to the left of it. This is why the general formulas for finding new L's and R's are:

\[ L(aw) = C(a) + O(a, L(w) − 1) + 1 \]

\[ R(aw) = C(a) + O(a, R(w)) \]

Where a is the new character added to the pattern and w is the previous pattern.

Now that these two formulas have been explained, we can take a look at the whole BW-Match algorithm. The following is the pseudo code for the whole algorithm [3]:

```plaintext
1. Initialize the suffix array with the suffixes of the pattern.
2. For each character a in the pattern, do:
   a. Move the L pointer to the suffix array entry with 'a' as its second character.
   b. Move the R pointer to the suffix array entry with 'a' as its second character.
   c. If L pointer is before R pointer, then:
      i. Move the L pointer forward by calculating L(aw) using the general formula.
      ii. Move the R pointer forward by calculating R(aw) using the general formula.
3. If the R pointer is before the end of the suffix array, then:
   a. Move the R pointer to the next suffix that starts with 'a'.
   b. Move the L pointer to the suffix that is immediately after the R pointer.
4. If the R pointer is at the end of the suffix array, then:
   a. Stop the algorithm.
```
L = 0, R = n - 1 // the empty string occurs at all positions in x
i = m - 1
while i >= 0 and L <= R:
    // compute L(w[i...m]) from L(w[i+1...m])
    L = C(w[i]) + O(w[i], L - 1) + 1
    // compute R(w[i...m]) from R(w[i+1...m])
    R = C(w[i]) + O(w[i], R)
    i = i - 1
if i < 0 and L <= R:
    report that w occurs in x at positions \{S(k)|L <= k <= R\}
else: report that w does not occur in x

Since the algorithm starts with the empty string and adds a character at a time to the front of it, we start with L at the start of the suffix array and with R at the end. For the same reason we initiate i as the index of the last character of the pattern searched for, since that is the character which we add first.

The while-loop which comes after continues until i = -1, which means that it has added the last character of the pattern, or until L is greater than R, which would mean that the pattern does not occur in the text. The two first assignments in the while-loop are the formulas, which were explained previously. After the new L and R have been computed with the two formulas, the while-loop decrements i by one, so that it is ready to add the next character to the pattern. The condition after the while-loop checks if the while-loop went through the whole pattern by checking if i is less than 0, and it also checks if L <= R to make sure that the pattern occurs in the text. If the condition is met, the algorithm can report that w occurs in x at \{S(k)|L <= k <= R\}, since it has found L(w) and R(w) like we wanted it to. If the condition is not met, then the pattern w does not occur in x, and that would be reported.

The running time for the BW-Match algorithms is \(O(m + |output|)\), since it only needs to go through the characters of the pattern. It also reports at which locations the pattern occurs, which is why the length of the output is included.
3 Implementation

In the previous section, the naive algorithm, the KMP algorithm with preprocessing, and the BW-Match algorithms with preprocessing, have been explained. That explanation has set a base for understanding the implementations of those three algorithms and the necessary preprocessing. In this section we will take a look at all the implementations, which were made and used in this project. The programming language used for the implementations is Java.

The following is a link to the Java code: https://github.com/MeinhardD/exact_pattern_matching.git

3.1 The Naive Algorithm

The implementation here is basically performing the same tasks as the pseudo code in the theoretical section 2.2. Figure 2 below provides an overview of the implementation of the naive algorithm in this project:

![Figure 2: The implementation of The Naive Algorithm](image)

In this implementation, the indexes of the first character of the two strings are 0 instead of 1, since that is how Java works. Because of that difference, the results are incremented by 1, so that they resemble the results that would be reported in the pseudo code. In this Java implementation there has also been added a new condition, which prevents $i + j$ from being out of bounds. The pseudo code did not concern itself with that, but it is necessary in Java.

3.2 The KMP Algorithm

The KMP Algorithm has a preprocessing part and a part which uses the preprocessing. There are two methods with the same name, where one calls the preprocessing method and uses what it returns, when calling the other method of the same name. The method is depicted on figure 3. The preprocessing

![Figure 3: Method calling both the preprocessing and the KMP algorithm](image)
method is shown on figure 4. The preprocessing method calls borderArray, which constructs the border array of a given string and returns it. The method which constructs a border array is depicted on figure 5. The other search_KMP method uses the preprocessing and searches for the locations where a specific pattern can be found in a text. That method is shown on figure 6. It calls a private method called match, which is the method which actually compares characters, just like the helper routine in the pseudo code in section 2.3.2. It is depicted on figure 7.
3.3 BW-Match

3.3.1 Suffix Array

There is a suffix array class, which has a constructor which uses private methods for constructing a suffix array as an array of integers. The constructor can be seen on figure 8. It takes the text string as a parameter and it starts by adding the special character '$' to the string and stores it as a field variable. Next it calls makeArrayOfCharacters(), which constructs an array with the alphabet used in the current text and returns it, and stores the returned array as a field variable. That method is depicted on figure 9. makeArrayOfCharacters is an array containing all the different characters that occur in the text in lexicographical order. The special character '$' is always the first character in the array. The array is used when making the C-table and the O-table. When making makeArrayOfCharacters Collections.sort is used to sort it, but then the '$' is removed and added back to the start of the array, so that it always is seen as the character of lowest value no matter which other characters are in the text. Lastly the constructor calls makeSuffixArray, which constructs the suffix array and returns it, and stores the returned suffix array as a field variable.

The suffix array class contains a private class called Index, which is used by the method makeSuffixArray. The private class Index is a comparable class, which is used as a wrapper around the different indexes of the text. This wrapper can be sorted with methods like Arrays.sort(), since it is comparable. The Index class and its compareTo method are depicted figure 10. The compareTo method contains a naive approach to sorting a suffix array. makeSuffixArray first initiates an array and stores the returned suffix array as a field variable.

```java
private static int match(String x, String p, int l, int j) {
    int m = p.length();
    int inc = 0;
    while (j < n & x.charAt(i) == p.charAt(j)) {
        j++;
        inc++;
    }
    return inc;
}
```

Figure 7: The match method used in the KMP algorithm

```java
public SuffixArray(String text) {
    this.text = text + '$';
    arrayCharacters = makeArrayOfCharacters();
    suffixArray = makeSuffixArray(this.text);
}
```

Figure 8: The constructor for the SuffixArray class

```java
// Constructs an array with the alphabet used in the current text and returns it
public List<String> makeArrayOfCharacters() {
    List<String> result = new ArrayList<String>();
    for (int i = 0; i < text.length(); i++) {
        String currentCharacter = text.substring(i, i+1);
        if (result.contains(currentCharacter)) {
            result.add(currentCharacter);
        } else {
            result.add(currentCharacter);
        }
    }
    return result;
}
```

Figure 9: The method makeArrayOfCharacters

```java
// Constructs an array with the alphabet used in the current text and returns it
public List<String> makeArrayOfCharacters() {
    List<String> result = new ArrayList<String>();
    for (int i = 0; i < text.length(); i++) {
        String currentCharacter = text.substring(i, i+1);
        if (result.contains(currentCharacter)) {
            result.add(currentCharacter);
        } else {
            result.add(currentCharacter);
        }
    }
    return result;
}
```

Figure 10: The Index class and its compareTo method
for Index objects and iteratively creates an Index object for each index of the text while adding that Index object to the array. Next it uses Arrays.sort() to sort the array containing the Index objects and wraps the result with a for loop before returning it. The method can be seen on figure 11. Since Array.sort() uses the naive CompareTo method of the private Index class, the construction of the suffix array in this implementation is non-linear.

The SuffixArray class also contains a method for computing the BWT for a specific character in the text. The BWT method is shown on figure 12. This method is in the SuffixArray class, because it contains the field variable with the text.

### 3.3.2 The C-table

The CTable class only uses one method in its constructor and saves what that method returns in a field variable called cTable. The method used is called makeCTable and it constructs the C-table and
returns it. It is depicted on figure 13. The CTable class also has a get method, which returns the
integer in the C-table related to the given key and is shown on figure 14.

```
// Constructs the C-table and returns it
private Map<String, Integer> makeCTable(String text, SuffixArray suffixArray) {
    List<String> arrayOfCharacters = suffixArray.getArrayOfCharacters();
    Map<String, Integer> result = new HashMap<>();
    result.put('$', 0);
    result.put(arrayOfCharacters.get(1), 0);
    int value = 0;
    for (int i = 2; i < arrayOfCharacters.size(); i++) {
        for (int j = 0; j < text.length(); j++) {
            if (text.charAt(j) == arrayOfCharacters.get(i).charAt(0)) {
                value = i;
            }
        }
        result.put(arrayOfCharacters.get(1), value);
    }
    return result;
}
```

Figure 13: The method which constructs the C-Table

When making the the C-table, the method on figure 13 gets the array called arrayOfCharacters from
the suffix array, which is given to the constructor as an argument. Then it starts by making a new
hashmap and putting the two first elements in it. The first has "$" as the key and 0 as the value, while
the second one has the first character after $ as the key and also 0 as the value so that it can be like
in the explanation in section 2.4.2. After that the method can make a local variable, which initially
contains 0 and will be incremented, when characters in the text are counted. For each character in
the text the method counts how many of the character there are in the text by incrementing the local
variable called value. After having counted the number of characters, it stores the value as the value in
the hashmap with the character as the key. The value variable is not reset after each character, but it
continues to count from where it left off and adds a new element to the map for each different character
in the text until there is none left. Then the method ends by returning the resulting hashmap.

3.3.3 The O-table

The OTable class only uses one method in its constructor and saves what that method returns in a field
variable called oTable. The method used to construct the O-table and return it is called makeOTable.
It is depicted on figure 15. The OTable class also has two get methods, one which returns the integer
in the O-table related to the given key and index, and one that returns the row of the O-table related
to the given key. They are shown on figure 16.

```
// Returns the int in the C-table related to the given key
public int get(String key) {
    if (key.length() == 0) {
        System.out.println("Input should be a string of length 1");
        return -1;
    }
    if (!cTable.containsKey(key)) {
        System.out.println("C-table: The text does not contain the input character");
        return -1;
    }
    return cTable.get(key);
}
```

Figure 14: The get method of the CTable class

The method for constructing the O-table starts off the same as the one, which constructs the C-table
by getting the arrayOfCharacters and initiating a new hashmap. It does not add two elements to the
hashmap right away though, like the one for the C-table did. The method instead calls a method
which constructs a BWT array, stores the returned BWT array in a local variable, and uses the array.
The method which makes the BWT array is a private method of the OTable class and is depicted on figure 17.

### 3.3.4 The BW-Match Algorithm

Up until now we have discussed the preprocessing that needs to happen before running the BW-Match algorithm. In the Main class there is a method, which calls the constructors for the SuffixArray, OTable and CTable, stores them in a local variable each, wraps them in a wrapper object called BWPreprocessing, and returns the wrapper object. The method is shown on figure 18. There is also a method which calls the preprocessing method, stores returned the wrapped object in a local variable, and uses it as one of the arguments when calling bwMatch, which is the BW-Match algorithm. It is depicted on figure 19. Lastly there is the algorithm itself, which is implemented as the method called bwMatch and is shown on figure 20. It unwraps the preprocessing out of the wrapper object, stores...
each element of the preprocessing in their own local variable, and uses them to find a given pattern in a given text as explained in section 2.4.4. The result is incremented by 1 in this method, just like in the implementation of the naive algorithm and the KMP algorithm. This way, all three implementations return the same locations for where a pattern occurs in a text, when given equal inputs.
4 Experiments

In the previous section all the implementations were shown, and some parts were explained. This section will go through how the implementations were used to run three separate experiments on the algorithms.

4.1 The Three Experiments

The goal of the first experiment is to verify the correctness of the implementations of the Naive algorithm, the KMP algorithm and BW-match. There are four texts that are used as input for all three algorithms, and there has been chosen a separate pattern to be searched for in each text respectively.

The purpose of the second experiment is to see if the implementations run as well in practice as they should in theory, i.e. how well the runtime of the implementations correlates with the complexity of the algorithms. In the experiment some test data is produced, which is used for all three algorithms. There are three different data sets, one for a worst-case scenario, one for a best-case scenario, and one for a random scenario. Diagrams are plotted from the results of the experiment, and it is discussed how well the results correspond with expectations.

The third and last experiment establishes how many different short patterns one would need to search for in a text, before it is worth it to construct the C- and O-tables for BW-match, instead of just using the KMP algorithm.

All three experiments have methods in Java, which run the experiments using the implementations of the algorithms.

4.2 Verification Experiment

In this first experiment four different texts are chosen to test the three different algorithms: the Naive algorithm, the KMP algorithm, and the BW-Match algorithm. The first two texts only consist of one word each, "banana" and "mississippi". The other two texts are the rhyme of The Ancient Mariner and the narrative poem called The Walrus And The Carpenter. The Walrus And The Carpenter is 3389 characters long, while The Ancient Mariner is 28299 characters long, and they both contain special characters. None of the texts contain dollar signs though, so a dollar sign can still be used as the unique character in the suffix array used for BW-Match. The patterns used for the verification experiment were 'ss' for 'mississippi', 'ana' for 'banana', 'Walrus' for The Walrus And The Carpenter, and 'Albatross' for The Ancient Mariner. Each pattern occurs multiple times at spread out locations in their respective texts.

To verify that the 3 algorithms can perform as expected, there is written a verification method called VerifyByRunningTests. It works by running the 4 texts through each of the algorithms with the patterns mentioned, storing the results, and assessing that the results contain all the correct locations.

The implementations of all three algorithms were verified this way to output correct results, for all four different inputs.
4.3 Runtime vs Complexity Experiment

After the implementations were verified to run correctly, it was possible to do the second experiment, since it would be difficult to if not impossible to run the second experiment with faulty implementations of the algorithms.

In this second experiment there are three different data sets that are used to see if the runtime of the implementations correlates with the complexity of the three algorithms. One data set for the worst-case scenario of the Naive algorithm, one for the best-case scenario, and one randomly generated data set. When talking about the runtime and complexity in this report the three symbols \( T \), \( n \), and \( m \) are used, where \( T \) stands for time, \( n \) stands for the length of the text, and \( m \) represents the length of the pattern.

The dataset for the worst-case scenario has a text of only a’s and a pattern of only a’s, since that would make the pattern occur in almost every location of the text. The dataset for the best-case scenario has a text of only a’s and a pattern of only b’s, since that pattern does not occur at any location in the text. The random dataset has a randomly generated text of a’s, c’s, g’s, and t’s, which is reused for all the implementations of the three algorithms. The patterns used are prefixes of different lengths taken from the text, so that they occur at least once in the text.

The sets of \( n \) and \( m \) used for all three data sets are \( n \in \{1000, 2000, 3000, \ldots, 200000\} \) and \( m \in \{100, 200, 300, \ldots, 1000\} \). When running the experiment, it first runs the relevant algorithm with the first \( n \) and the first \( m \), then the first \( n \) and the second \( m \) and so on, until it has gone through all 10 \( m \)’s. Then it uses the second \( n \) and the first \( m \) again and so on, until it has gone through all the \( n \)’s, or as many as we are willing to wait for. For each different \( n \) a text of length \( n \) of only a’s is generated for the worst and best case scenarios, while a prefix of length \( n \) is used from the randomly generated string for the random scenario. It is works the same way for the different values of \( m \), with the three respective data sets.

For every iteration, i.e. pairing of an \( n \) and an \( m \), the algorithm is run a certain amount of times, with the runtime being measured every time, and then the mean of all the time results is used as data for the experiment. It is done this way, because the runtime results can vary a lot at times, so running each iteration multiple times and using the mean result helps in making the results more stable. For the Naive algorithm it is repeated 50 times, while it is repeated 200 times for the KMP and the BW-Match. For KMP and BW-Match, the runtime is only measured for the algorithm itself, not the preprocessing.

The results of the datapoints have been saved in google sheets and there been plotted into diagrams. Here is a link to the Naive runtime data: https://docs.google.com/spreadsheets/d/1YZKLKOH-Nf8B_-qOyVptI1Xq8Ju5NIMhNNgs_Mq1W0/edit?usp=sharing

Here is a link to the KMP runtime data: https://docs.google.com/spreadsheets/d/1HX3HquFVKBlqtZkiYDDK/edit?usp=sharing

Lastly, here is a link to the BW-Match runtime data: https://docs.google.com/spreadsheets/d/1oJw11d220Y4TX-QQMwu2TvH07SXG-B1uoM3bj9HRVoY/edit?usp=sharing

In each of the three google sheets linked to, only the three last sheets are used in the report.
4.3.1 The Naive Algorithm

The complexity of the Naive algorithm is $O(n \cdot m)$. Because of that, the data that was collected from the experiment for the worst-case scenario of the Naive algorithm is plotted with $n \cdot m$ on the x-axis and $T(n,m)_{n \cdot m}$ on the y-axis. If the runtime correlates correctly with the complexity, we would expect to see the line on the diagram flatten out and approximate a horizontal line. We can see on the diagram

![Figure 21: Diagram for the worst-case runtime of the Naive algorithm](image)

that the line is very shaky on the left of the diagram, and then gradually settles as it progresses to the right. The result looks like what would be expected.

In the best-case scenario, where the pattern will not match with the text anywhere, not even at the first character of the pattern, we would expect the algorithm to go through the dataset in $O(n)$ time. For this reason, the best-case runtime data is plotted with $n$ on the x-axis and $T(n,m)_{n}$ on the y-axis. As before we would expect the line on the diagram to approximate a horizontal line. The line

![Figure 22: Diagram for the best-case runtime of the Naive algorithm](image)

in the diagram seen on figure 22 very quickly begins approximating a horizontal line, so it is clearly as expected.

In the random case, we would expect the runtime to lie between the worst-case and the best-case
for the different values of $n$, i.e. different lengths of the text. Because of that, it has been plotted with $n$ on the x-axis and $Time(n, m)$ on the y-axis, to make it easier to compare the three runtimes side by side. There is no need to divide the time by anything, when we are comparing the three.

![Figure 23: Diagram for the random runtime of the Naive algorithm](image1)

Figure 23 shows a diagram with the random runtime on its own, so we can see what it looks like by itself, while figure 24 shows the random runtime next to the worst-case and best-case runtimes. As can be seen on figure 24, the runtime of the random case is a little higher than the best-case and clearly lower than the worst-case, which means that the results correlate well with the expectations.

**4.3.2 The KMP Algorithm**

The complexity for the KMP algorithm is $O(n + m)$. Because of that, the data that was collected from the experiment for the worst-case scenario of the KMP algorithm is plotted with $n + m$ on the x-axis and $T(n, m)$ on the y-axis. We would again expect the line of the plotted diagram to approximate a horizontal line. As depicted on the diagram in figure 25, the line does approximate a horizontal line even though it is quite shaky. It is though as expected.

![Figure 24: Diagram of all three runtimes of the Naive algorithm. The red line illustrates the runtime of the worst-case scenario, the orange line illustrates the random case runtime, and the green line illustrates the best-case runtime.](image2)
As with the Naive algorithm, here we also expect the best-case scenario to lie within $O(n)$ time. There is though not as big a difference between $O(n)$ and $O(n + m)$ as there is between $O(n)$ and $O(n \times m)$. But still the diagram for this was plotted with $n$ on the x-axis and $T_{n,m}$ on the y-axis. We again expect an approximation of a horizontal line. In the diagram on figure 26 it can be seen that, the line clearly approximates a horizontal line, so it is as expected.

In the random case, we would expect the runtime to lie between the worst-case and the best-case for the different values of $n$. Because of that, it has been plotted with $n$ on the x-axis and $Time(n, m)$ on the y-axis.

Figure 26 shows a diagram with the random runtime on its own, so we can see it by itself, while figure 28 depicts the random runtime beside the worst-case and best-case runtimes.

We can clearly see on figure 28 that the runtime for the random case lies between the worst-case and the best-case. This correlates well with the expectations.
4.3.3 The BW-Match Algorithm

The complexity for BW-Match is $O(m + |output|)$. Because of that, the data that was collected from the experiment for the worst-case scenario of BW-Match is plotted with $m + |output|$ on the x-axis and $\frac{T(n,m)}{m+|output|}$ on the y-axis. We would expect the line in the diagram to gradually approximate a horizontal line. As seen in the diagram of figure 29, the line settles to approximating a horizontal line as expected.

In the best-case scenario, where the algorithm does not find even the first character of the pattern in either the C-table or the O-table, we would almost expect the runtime to be constant since, we should not expect the algorithm to go through the pattern at all after seeing the first character. The runtime has though been plotted with $m$ on the x-axis and $\frac{Time(n,m)}{m}$ on the y-axis, since the runtime seemed to be affected somewhat by the value of $m$ when running it. The length of the output has not been added, since the output obviously is empty every time. It would be expected to see the runtime start to approximate a horizontal line.

The diagram for the best-case runtime of BW-Match can be seen on figure 30. It is seemingly starting
to approximate a horizontal line, which correlates with the expectations.

In the random case, we would expect the runtime to lie between the worst-case and the best-case for the different values of $n$. Because of that, it has been plotted with $n$ on the x-axis and $Time(n, m)$ on the y-axis.

Figure 31 shows a diagram with the random runtime on its own, while figure 32 shows the random runtime beside the worst-case and best-case runtimes. Only a short part of the random case is seen though, since the worst-case does not have as many datapoints as the other two. There are fewer datapoints for the worst-case scenario, because it took so much longer to run it, than it took to run the others.

In figure 32 it can be seen that the runtime for the random case is mostly between the worst and best case runtimes. This correlates fine with the expectations.
4.4 KMP vs BW-Match Experiment

Now that the algorithms have been confirmed to run in practice as the theory suggests, it is possible to make comparisons between two of the algorithms.

In this experiment we compare the KMP algorithm and the BW-Match algorithm to see when which should be used. We want to find out when it is worth it to use the time to construct the C-table and O-table, instead of just using KMP, i.e. how many $k$ patterns of length $m$ would you need to match with a text of length $n$, before it is worth it to construct the tables.

The text used for this experiment is cut from an RNA string of length 1000000. The different lengths of text used in the experiment were all prefixes of that string. The RNA string was chosen to make it resemble a real use case as closely as possible in the time available. The patterns are substrings taken at random indexes from the text, all having the same length. Every time patterns are taken from the text, it is made sure to be from the currently used prefix of the full text. The patterns are taken from the text, so they always occur in the text at least one location. This is to make sure that
the patterns are not solely treated as if they are shorter than they are, since it is very unlikely for there to be many reoccurring patterns of the length that we wanted to use. The different pattern lengths $m$ chosen are 25, 50, 75, and 100. These are chosen since 25 is around the length of patterns searched for in RNA strings, and we want to see how three longer lengths would affect the results. The exact other three lengths are chosen as multiples of 25 since it makes the different lengths uniform.

For each text length $n$ the experiment method runs through the two algorithms KMP and BW-Match multiple times searching for different patterns in the same text. The preprocessing for BW-Match is only done once per text length, while the preprocessing for KMP is done for each different pattern searched for. This is because the preprocessing of BW-Match uses the text, while the preprocessing for KMP uses the pattern. When time is measured in this experiment, the construction of the suffix array is excluded, since the implementation used is non-linear. The time for the BW-Match is measured by first measuring the time it takes to construct the C-table and the O-table, and then adding the time it takes to run the BW-Match algorithm with each pattern. The time for the KMP algorithm is initiated as zero, and then the time it takes to run the whole KMP algorithm with preprocessing is added for each pattern. This means that when the method is about to start running the two algorithms with different patterns, the time for BW-Match, called BWMatchTime, is already at how long it took to construct the C- and O-tables, while the time for KMP, called KMPTime, is at zero.

When acquiring the patterns from the text, they are all added to an array of strings called searchPatterns. A new array is made for every time $n$ changes. When running the two algorithms with different patterns, the patterns are taken from that array, and used with both algorithms. So one pattern is used at a time with both algorithms, where the runtime is measured separately. In each iteration of the algorithms being run with a different pattern, it is checked if $\frac{BWMatchTime}{KMPTime} < 1$, to make sure that it still is faster to just use KMP instead of constructing the C- and O-tables and using BW-Match. When $\frac{BWMatchTime}{KMPTime} < 1$ is true, how many $k$ patterns have been searched for is saved and the loop that runs the algorithms with different patterns is stopped, whereafter the saved datapoint is added to a sum, since this process is repeated 200 times, whereafter the mean of the sum is used as a $k$ in one of the diagrams. Only then does it move to go through the same process with a different value of $n$.

When running the experiment, we ran it enough to fill a whole diagram for each value of $m$, i.e. the value of $m$ is constant in each of the four diagrams that came from the experiment. When $m$ was 25, 50, and 75, the $n$ started at 500 and increased with increments of 500 until it ended at 50000, but when $m$ was 100, the $n$ started at 1000, increased at increments of 1000, and ended at 100000. This means that each diagram has 100 datapoints.

The results of the experiment can be seen on figures 33, 34, 35, and 36. The values of the datapoints can be found in the google sheet that this links to: https://docs.google.com/spreadsheets/d/1wIWaDWIwU2V-tghbAZCrYUabjvB9f1s8yEEWgCL0b0/edit?usp=sharing, on the sheets named $m=25$, $m=50$, $m=75$, and $m=100$ respectively.

From the results, it can be concluded that it is worth it to construct the C- and O-tables and use BW-match over KMP, when you want to search for 10 or more patterns in a text. When looking at the different diagrams, they are mostly similar to each other. The one for $m = 50$ is more wobbly at the smaller lengths of text, but it also settles around the same as the others at the longer lengths. The different pattern lengths most likely do not make much of a difference, because there is not a high probability of there being multiple equal patterns of those lengths. The one full match that each pattern had did not make much of a difference either, since it was only one full match of the different
pattern lengths. It is still interesting to see that all four diagrams show a tendency to settle around a certain interval of $k$ quite early and continue within it until the end. The observation indicates that the resulting values of $k$ would continue to be within the same interval at even higher lengths of text.
Figure 36: Diagram of the experiment with $m = 100$
5 Conclusion

In this report, the three different algorithms have been explained in detail, to give an overview of how they function, and that they are sound. It has even been shown how the different data structures needed for some of the algorithms are constructed, how they work within the algorithms, and why they are necessary. After the explanation of the algorithms, it was shown how they had been implemented in Java. Lastly the three different experiments that were run were discussed. We went through the different data sets chosen for the experiment and why the were chosen. The results of the experiments were also discussed. The following were the conclusions.

The experiments generally confirmed the prior expectations: The first experiment verified that all the implementations worked correctly, i.e. all the algorithms produced correct outputs when given various inputs. The second experiment verified that all three algorithms ran within the expected times. The findings of the third and last experiment were that when comparing the KMP algorithm and BW-Match, the tendency was for the resulting diagrams to quite quickly settle at a narrow interval of the number of patterns needed to search for, before it was worth it to create the C-table and the O-table. The consensus was that it would be worth it to construct the C- and O-tables if one wanted to search for 10 or more different patterns in one text.

Now that these implementations have been made of the algorithms for exact pattern matching, further work could be made to implement versions of the same algorithms which do approximate pattern matching. Similar experiments could be made with those algorithms and the ones for approximate pattern matching could also be compared to the algorithms for exact pattern matching. The aforementioned idea could provide a solid basis for further research, and even provide the base idea for a new thesis.
References


