Genome Assembly

officialprofile

2021-09-14

Contents

1 Preface 5

1.1 Prerequisites 6

2 Introduction 7

3 Basic string manipulations 9

3.1 Random DNA sequence 9

3.2 Find cDNA 10

3.3 Reverse complementary 11

3.4 Frequency of the nucleobases 11

3.5 Longest common prefix 12

3.6 Exact matching 13

3.7 Generate all possible k-mers 14

4 More advanced problems 17

4.1 Read fasta file 17

4.2 Count k-mers 18

4.3 Skew 19

4.4 Index 20

5 Graph theory 21

6 De Bruijn graph 23

7 Burrows-Wheeler transform **25**
7.1 Introduction 25
7.2 Burrows-Wheeler matrix 25
7.3 Inverse transform 27

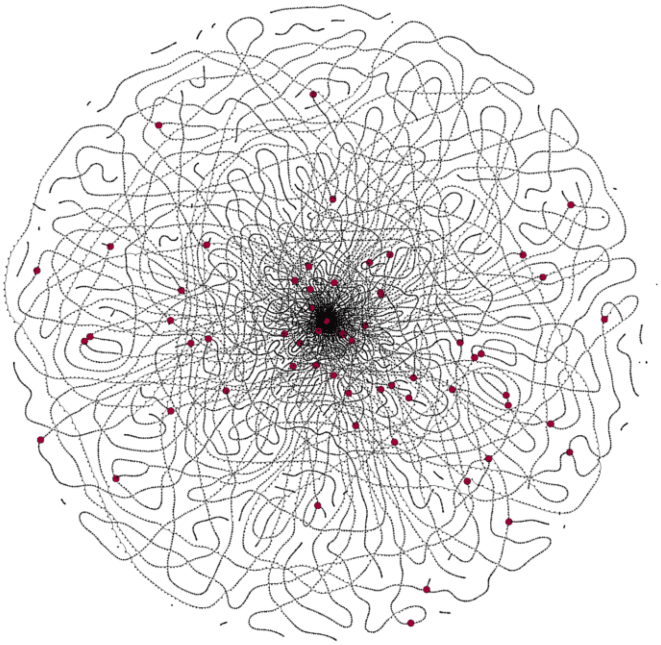
Chapter 1

Preface

This mini textbook describes selected algorithms that play a fundamental role in genome assembly. The premise of this book is to construct the algorithms from the very bottom and explain step by step main ideas that stand behind them.

We will try to avoid as much as possible ready-to-use implementations, which are of course available and of good quality, but their use wouldn't serve the educational purpose. Naturally, many applications are included as well.

By default the code is written in R, but at certain points python is also being mentioned.



A HiFi De Bruijn graph for a pile of reads from Drosophila genome sequencing. Each dot represents a k-mer (k=23), the edges denote neighboring k-mers. The larger red dots mark the head of heterozygous bubbles. Source: pacb.com.

1.1 Prerequisites

It is assumed that the reader:

1. Has a basic understanding of genetics;
2. Has some experience with programming in R (is familiar with loops, data structures, etc.).
3. Had some contact with higher mathematics, e.g. statistics, graph theory. Expertise in this field is by no means required though.

Throughout the book the following libraries are being used and it is assumed that the reader has them loaded.

```
library(stringr)
library(dplyr)
library(igraph)
```

Chapter 2

Introduction

Genome assembly has been regarded as one of the most important and most challenging problems in bioinformatics, perhaps at least since the late 80's, when the Human Genome Project was announced. Due to exponential growth of technology this topic is still relevant and there are still quite a few problems we have to face.

Genomes tend to be almost inconceivably long. Human DNA for example is approximately thousand times longer than the Bible (letter-wise), and some species have their genomes even orders of magnitude longer. Yet the algorithmic part of the problem arises not mainly because of the length. This was definitely the case for the biotechnologicians, who few decades ago were constrained within low-thorughput sequencing techniques. For bioinformaticians though the core of the problem is located a bit deeper.

Good analogy, that explains the nature of our challenge, is a shredded newspaper (some authors call it the newspaper problem, others use book as an example). Namely, genome assembly resembles reconstruction of the original document from a set of unarranged newspaper pieces, like on the figure below.

As one can imagine the problem is difficult, but not because the newspaper is twenty or forty pages long. With just a single sheet the task still wouldn't be easy. The length of the genome, although far from being irrelevant, for us is more of a secondary issue. On the other hand, repetitive patterns, which for biotechnologicians are not a problem at all, will complicate our journey substantially.

One should also underline the fact that transcriptome assembly is not the same as the genome assembly. Read coverage of the latter is relatively uniform, whereas transcriptome can be differentially expressed and therefore the frequency of its reads varies a lot. In genome assembly non-uniform abundance of the reads simply indicates presence of repetitive patterns. In the case of transcription product it is much less straightforward. Methods that overcome this issue exist but they are beyond of the scope of this textbook.

Chapter 3

Basic string manipulations

Before we dive into the main topics let's warm up with few basic exercises. Some of these problems may be regarded as a form of general familiarization with string manipulations. Having them in the bioinformatics repertoire is here in a sense required. From a perspective of the algorithms that will be discussed later some of the problems, like generating k-mers or suffixes, are absolutely essential and one has to know them like the back of one's hand.

3.1 Random DNA sequence

Let's start with writing a function that, for a given integer $n > 0$, returns randomly generated DNA sequence.

```
random_sequence <- function(n){
  return(sample(c('A','C','G','T'), n, replace = TRUE))
}

random_sequence(10)
#>  [1] "G" "A" "C" "T" "A" "T" "G" "A" "T" "G"
```

If we want our returned sequence to be in a form of a single string we can use `paste()` function with collapse parameter equal to `''`.

```
paste(random_sequence(10), collapse = '')
#> [1] "TGCCGCATTG"
```

Unfortunately, there is no convenient way to retrieve a substring. If a `DNA` variable is equal to `'ACGTG'` then we won't get a substring `'CG'` simply by writing `DNA[2:3]` (one could to this in python). One could do this by employing the `substr()` function, i.e. `substr(DNA, 2, 3)`. On the other hand, if DNA is a vector `c('A', 'C', 'G', 'T', 'G')` then `DNA[2:3]` will return `c('C', 'G')` that in turn can be merged into `'CG'` by `paste()`. Either way, one must use a certain function in order to glue the letters or to split the sequence.

3.2 Find cDNA

To find the complementary sequence one must know that C (cytosine) is complementary to G (guanine), and A (adenine) is complementary to T (thymine). Also, before creating the cDNA we will at first construct a list (in Python it would be a dictionary) of complementary nucleobases.

```
c_bases <- list('A' = 'T', 'C' = 'G', 'G' = 'C', 'T' = 'A')

c_bases['A'][[1]]
#> [1] "T"
```

This data structure allows us to directly access complementary bases. Let's build a function that utilizes this.

```
complementary_sequence <- function(sequence){
  complementary = c()
  for (base in sequence){
    complementary = c(complementary, c_bases[base][[1]])
  }
  return(complementary)
}
```

Please note that we append complementary nucleobase in a seemingly non-optimal way, namely by writing `sequence = c(sequence, new_nucleobase)`. One could argue that this should be done with `append()`, i. e. `sequence = append(sequence, new_nucleobase)`, but in fact the append in R is regarded as a relatively slow function and our solution is usually more recommended (in python using append is fine).

```
seq <- random_sequence(20)
cat(' DNA =', seq, '\n')
#>  DNA = C C G T T G T A A C T C A C A G G G C A
cseq <- complementary_sequence(seq)
cat('cDNA =', cseq)
#> cDNA = G G C A A C A T T G A G T G T C C C G T
```

There is also a neater way to compute, and in a way avoid, these kind of loops. One can achieve this though functions like `apply()`, `sapply()` or `lapply()`.

```
complementary_sequence <- function(sequence){
  sapply(seq, function(base) c_bases[base][[1]])
}
```

The clear downside of using functions like the one above is loss of legibility. Applying them can also be not as straightforward and intuitive as writing a conventional loop. However, more advanced programmers find these functions very handy, fast and as readable as standard for or while loop. For that reason we will try to balance these two approaches out. We don't want our code to be too hermetic, but also we should not limit ourselves by deliberately avoiding legitimate solutions.

3.3 Reverse complementary

Obtaining reverse complementary is very similar to the previous excercise. The only difference is that instead of appending complementary nucleobases we will prepend them.

```
reverse_complementary <- function(sequence){
  reverse = c()
  for (base in sequence){
    reverse = c(c_bases[base][[1]], reverse)
  }
  return(reverse)
}
```

```
seq <- random_sequence(20)
cat(' DNA =', seq, '\n')
#>  DNA = C T C G C C A A T T T T A G G A G T G C
rcseq <- reverse_complementary(seq)
cat('cDNA =', rcseq)
#> cDNA = G C A C T C C T A A A A T T G G C G A G
```

3.4 Frequency of the nucleobases

```
frequency <- function(sequence, percents = FALSE){
  counts <- table(sequence)
  divide_counts <- ifelse(percents, sum(counts), 1)
  return (counts/divide_counts)
}
```

```
frequency(seq, percents = TRUE)
#> sequence
#>    A    C    G    T
#> 0.20 0.25 0.25 0.30
```

3.5 Longest common prefix

```
longest_common_prefix <- function(sequence1, sequence2){
  min_length = min(length(sequence1), length(sequence2))
  index = 0
  if (sequence1[1] != sequence2[1]){
    return ('No common prefix')
  }
  for (i in 2:min_length){
    if (sequence1[i] != sequence2[i]){
      return (sequence1[1:i-1])
    }
  }
  return (sequence1[1:min_length])
}
```

```
longest_common_prefix(c('A','C','T','G'), c('A', 'C', 'T', 'T', 'T'))
#> [1] "A" "C" "T"

longest_common_prefix(c('C','C','T','G'), c('A', 'C', 'T', 'T', 'T'))
#> [1] "No common prefix"
```

Imputing strings that are in fact consisted of vectors of characters may not be particularly convenient. We will therefore add a functionally that detects type of inputs and converts them if it is needed.

Unfortunately, R does not differentiates between `c('a','c','t','g')` and `'ACTG'`, e.g.

```
typeof(c('A', 'C', 'G', 'T'))
#> [1] "character"
typeof('ACGT')
#> [1] "character"
```

```
length(c('A', 'C', 'G', 'T'))
#> [1] 4
length('ACGT')
#> [1] 1
```

On this account we will simply add new boolean argument **`glued`** to our function (by default equal to FALSE). This solution is far from perfect, but for now is good enough.

```
longest_common_prefix <- function(sequence1, sequence2, glued = FALSE){
  if (glued){
    sequence1 <- strsplit(sequence1, '')[[1]]
    sequence2 <- strsplit(sequence2, '')[[1]]
  }
  min_length = min(length(sequence1), length(sequence2))
  index = 0
  if (sequence1[1] != sequence2[1]){
    return ('')
  }
  for (i in 2:min_length){
    if (sequence1[i] != sequence2[i]){
      return (sequence1[1:i-1])
    }
  }
  return (sequence1[1:min_length])
}
```

```
longest_common_prefix('ACTG', 'ACCC', glued = TRUE)
#> [1] "A" "C"

longest_common_prefix('TCTG', 'ACCC', glued = TRUE)
#> [1] ""
```

3.6 Exact matching

We will construct a naive algorithm for exact pattern matching

```
exact_matching <- function(pattern, template, glued = FALSE){
  if (glued){
    pattern <- strsplit(pattern, '')[[1]]
    template <- strsplit(template, '')[[1]]
  }
  positions = c()
  for (i in 1:(length(template) - length(pattern) + 1)){
    match = TRUE
    for (j in 1:length(pattern)){
      if (pattern[j] != template[j + i - 1]){
        match = FALSE
        break
      }
    }
    if (match){
      positions = c(positions, i)
    }
  }
  return(positions)
}
```

```
exact_matching('ACTG', 'ACTGTTGACTGACTGGGGGGGGACTGAACTG', glued = TRUE)
#> [1]  1  8 12 23 28
```

3.7 Generate all possible k-mers

```
all_kmers <- function(k){
  kmers <- c('A', 'C', 'G', 'T')
  for (i in 1:(k-1)){
    kmers_new = c()
    for (j in 1:length(kmers)){
      kmers_new <- c(kmers_new,
                     paste(kmers[j], 'A', sep = ''),
                     paste(kmers[j], 'C', sep = ''),
                     paste(kmers[j], 'G', sep = ''),
                     paste(kmers[j], 'T', sep = ''))
    }
    kmers = kmers_new
  }
  return(kmers)
}
```

```
all_kmers(3)
#>  [1] "AAA" "AAC" "AAG" "AAT" "ACA" "ACC" "ACG" "ACT" "AGA"
#> [10] "AGC" "AGG" "AGT" "ATA" "ATC" "ATG" "ATT" "CAA" "CAC"
#> [19] "CAG" "CAT" "CCA" "CCC" "CCG" "CCT" "CGA" "CGC" "CGG"
#> [28] "CGT" "CTA" "CTC" "CTG" "CTT" "GAA" "GAC" "GAG" "GAT"
#> [37] "GCA" "GCC" "GCG" "GCT" "GGA" "GGC" "GGG" "GGT" "GTA"
#> [46] "GTC" "GTG" "GTT" "TAA" "TAC" "TAG" "TAT" "TCA" "TCC"
#> [55] "TCG" "TCT" "TGA" "TGC" "TGG" "TGT" "TTA" "TTC" "TTG"
#> [64] "TTT"
```

Chapter 4

More advanced problems

This chapter is devoted to a little more specific problems, like managing fasta or fastq files. Throughout the section we will operate on three different files which are available on NCBI, but are also included in the `files` directory:

1. MK028861.fasta - Zika virus
2. NC_001416.fasta - Enterobacteria phage lambda

4.1 Read fasta file

Fasta file consist of two parts - header, that describes the sequence, and sequence itself. In order to load the first file we will use a very basic `leadLines()` function.

```
zika <- readLines('files/MK028861.fasta')

zika[1:4]
#> [1] ">MK028861.1 Zika virus isolate Zika virus/H.sapiens-tc/Panama/2015/259359 polyprotein ge
#> [2] "TGACTAAGACTGCGACAGTTCGAGTTTGAAGCGAAAGCTAGCAACAGTATCAACAGGTTTTATTTTGGAT"
#> [3] "TTGGAAACGAGAGTTTCTGGTCATGAAAAACCCAAAAAAGAAATCCGGAGGATTCCGGATTGTCAATATG"
#> [4] "CTAAAACGCGGAGTAGCCCGTGTGAGCCCCTTTGGGGGCTTGAAGAGGCTGCCAGCCGGACTTCTGCTGG"

length(zika)
#> [1] 151
```

As one can see at this point we have a vector of 151 elements. Each entry represents a single line in from the loaded file. We are going to remold our output to more convenient form, namely to a list of two elements (header and sequence).

```r
zika <- list('header' = substr(zika[1], 2, nchar(zika[1])),
             'sequence' = do.call(paste0, as.list(zika[2:length(zika)])))
```

Let's talk trough `substr(zika[1], 2, nchar(zika[1]))` and `do.call(paste0, as.list(zika[2:length(zika)]))`. We create a header on the basis of initial line of the fasta file, i.e. first element of our vector. One could simply write `'header' = zika[1]` but we also want to remove `>` character from the very beginning. Therefore we take a substring that starting from the second character and ends at the end (position equal to the number of characters).

The second part, which is `do.call(paste0, as.list(zika[2:length(zika)]))`, needs a little more explanation. As we know one can use the paste0 function to concatenate strings. Unfortunately, imputing a vector of strings won't concatenate all element together. For paste0 this vector is simply a one element.

```r
paste0('A', 'B', 'C')
#> [1] "ABC"

paste0(c('A', 'B', 'C'))
#> [1] "A" "B" "C"
```

This is the reason for employing `do.call()`. The function takes two arguments - name of function to execute and **list** of elements.

Fortunately, there is a smoother to solve this exercise - `str_c()` function the the stringr package.

```r
stringr::str_c(c('A', 'B', 'C'), collapse = '')
#> [1] "ABC"
```

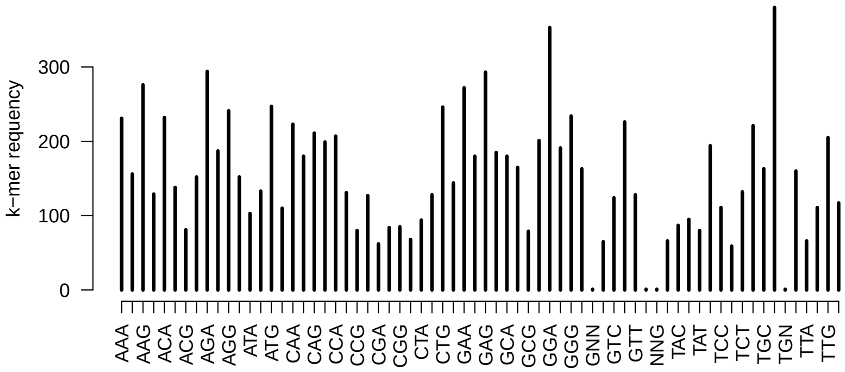
```r
zika$header
#> [1] "MK028861.1 Zika virus isolate Zika virus/H.sapiens-tc/Panama/2015/259359 polyp

substr(zika$sequence, 1, 100)
#> [1] "TGACTAAGACTGCGACAGTTCGAGTTTGAAGCGAAAGCTAGCAACAGTATCAACAGGTTTTATTTTGGATTTGGAAAC
```

4.2 Count k-mers

```r
count_kmers <- function(sequence, k, glued = TRUE){
  if (glued){
    sequence <- strsplit(sequence, '')[[1]]
  }
  kmer_count <- c()
  for (i in 1:(length(sequence)-k+1)){
    kmer_count <- c(kmer_count, paste(sequence[i:(i+k-1)], collapse = ''))
  }
  return(kmer_count)
}
```

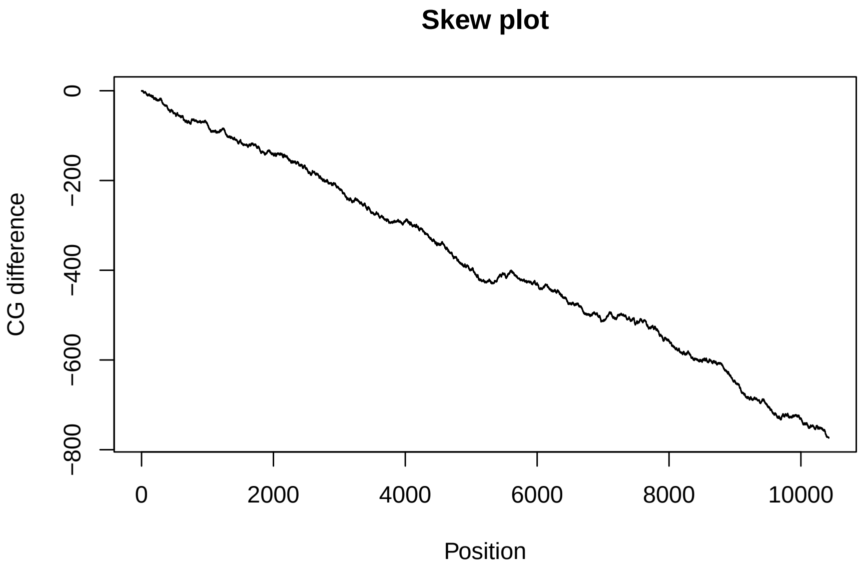
```
plot(table(count_kmers(zika$sequence, 3)),
     ylab = 'k-mer requency', type = 'h', lwd = 3, las = 2)
```



4.3 Skew

```
plot_skew <- function(sequence, glued = TRUE){
  kmer_skew <- 0
  if (glued){
    sequence <- strsplit(sequence, '')[[1]]
  }
  for (i in 2:length(sequence)){
    kmer_skew[i] = ifelse(sequence[i] %in% c('A', 'T'), kmer_skew[i-1],
                   ifelse(sequence[i] == 'C', kmer_skew[i-1]+1, kmer_skew[i-1]-1))
  }
  return(plot(1:length(sequence), kmer_skew, type = 'l',
              ylab = 'CG difference', xlab = 'Position',
                 main = 'Skew plot'))
}
```

```
plot_skew(zika$sequence)
```



4.4 Index

Chapter 5

Graph theory

Chapter 6

De Bruijn graph

Chapter 7

Burrows-Wheeler transform

The Burrows-Wheeler transform is one of the most effective lossless text compression method available. It provides a reversible transformation for text that makes it easier to compress. Of course, one may wonder what text compression has to do with genome assembly. As a matter of fact these two issues are closely related. But we should to be more precise here - text compression is closely related to pattern matching which in turn is crucial for the genome assembly. In a broad sense compression algorithms look for patterns and try to remove repetitions. We want to take advantage of this feature, especially because repetitive patterns tend to be very abundant in genomic sequences.

It is worth mentioning that the Burrows-Wheeler transform is also closely related to suffix trees and suffix arrays, which are commonly used within pattern matching. This relationship will be studied later but perhaps reader should already keep the trivia in mind. (Donald Adjeroh, 2008)

7.1 Introduction

The Burrows-Wheeler transform method is often referred to as “block sorting”, because it takes a block of text and permutes it. By permuting a block of text we mean rearranging the order of its symbols. Once again, we should be more precise here because Burrows-Wheeler transform performes a specific type of permutation, namely *circural shift permutation*: all of the characters are moved one position to the left, and first character moves to the last position.

7.2 Burrows-Wheeler matrix

Consider the following sequence:

```
sequence <- 'GATTACA'
```

In order to create the Burrows-Wheeler matrix, from which the transform itself can be obtained, for the given string we at first add the dollar sign $ at the end of the sequence.

```
sequence  <- str_c(sequence, '$')
```

Afterwards we perform a series of circular shift permutations.

```
sequences <- c(sequence)
n         <- nchar(sequence)

for (i in 1:(n-1)){
  sequence <- str_c(str_sub(sequence, 2, n),
                    str_sub(sequence, 1, 1))

  sequences <- c(sequences, sequence)
}

cat(sequences, sep = '\n')
#> GATTACA$
#> ATTACA$G
#> TTACA$GA
#> TACA$GAT
#> ACA$GATT
#> CA$GATTA
#> A$GATTAC
#> $GATTACA
```

Then we sort these sequences with the assumption that the dollar sign precedes lexicographically every other symbol.

```
sequences <- sort(sequences)
cat(sequences, sep = '\n')
#> $GATTACA
#> A$GATTAC
#> ACA$GATT
#> ATTACA$G
#> CA$GATTA
#> GATTACA$
#> TACA$GAT
#> TTACA$GA
```

For our convenience let's split these permutations into vectors of single characters.

```
bw.matrix            <- data.frame(matrix(, n, n))
colnames(bw.matrix) <- 1:n

for (i in 1:n){
  bw.matrix[i, ] <- strsplit(sequences[i], split = '')[[1]]
}

knitr::kable(bw.matrix)
```

1	2	3	4	5	6	7	8
$	G	A	T	T	A	C	A
A	$	G	A	T	T	A	C
A	C	A	$	G	A	T	T
A	T	T	A	C	A	$	G
C	A	$	G	A	T	T	A
G	A	T	T	A	C	A	$
T	A	C	A	$	G	A	T
T	T	A	C	A	$	G	A

Thus we have created the **Burrows-Wheeler matrix**. Sequence in the last column is called the **Burrows-Wheeler transform**.

```
transform <- paste(bw.matrix[,n], collapse = '')

cat('The Burrows-Wheeler transform of',
    sequence, 'is', transform)
#> The Burrows-Wheeler transform of $GATTACA is ACTGA$TA
```

7.3 Inverse transform

As we said at the very beginning the transform is reversible. Having only the transformed sequence we are going to reconstruct the Burrows-Wheeler matrix and initial sequence itself.

Firstly let's sort the characters of the transformed sequence.

```
first.sequence <- strsplit(transform, split = '')[[1]] %>% sort
paste(first.sequence, collapse = '')
#> [1] "$AAACGTT"
```

Note that this string is equivalent to the first column of the Burrows-Wheeler transform.

```
bw.inverse           <- data.frame(matrix(, n, 2))
colnames(bw.inverse) <- c(n, 1)

bw.inverse[, 1] <- strsplit(transform, split = '')[[1]]
bw.inverse[ ,2] <- first.sequence

knitr::kable(bw.inverse)
```

8	1
A	$
C	A
T	A
G	A
A	C
$	G
T	T
A	T

Also keep in mind that the characters from last and the first column are adjacent. In other words, at this point we have a set of 2-mers.

```
kmers <- apply(bw.inverse, 1,
               function(x) paste(x, collapse = ''))
kmers
#> [1] "A$" "CA" "TA" "GA" "AC" "$G" "TT" "AT"
```

The reconstruction process strictly relies on the fact that Burrows-Wheeler matrix is sorted lexicographically. This property will allow us to retrieve the remaining columns.

```
kmers <- sort(kmers)
kmers
#> [1] "$G" "A$" "AC" "AT" "CA" "GA" "TA" "TT"
```

The 2-mers (k-mers in general) that we sorted lexicographically represent first two columns of the Burrows-Wheeler matrix. We can extract last character of each 2-mer in the following way:

```
sapply(kmers, function(x) str_sub(x, 2, 2),
       simplify = TRUE, USE.NAMES = FALSE)
#> [1] "G" "$" "C" "T" "A" "A" "A" "T"
```

By inserting this set of characters we obtained the second column, and by iterating the proccess of building substrings, sorting them, and retrieving last characters we can fill the whole Burrows-Wheeler matrix.

```r
for (i in 2:(n-1)){
  kmers              <- apply(bw.inverse, 1,
                              function(x) paste(x, collapse = ''))
  kmers              <- sort(kmers)
  bw.inverse[, i+1] <- sapply(kmers, function(x) str_sub(x, i, i),
                               simplify = TRUE, USE.NAMES = FALSE)
  colnames(bw.inverse)[i+1] = i
}
knitr::kable(bw.inverse)
```

8	1	2	3	4	5	6	7
A	$	G	A	T	T	A	C
C	A	$	G	A	T	T	A
T	A	C	A	$	G	A	T
G	A	T	T	A	C	A	$
A	C	A	$	G	A	T	T
$	G	A	T	T	A	C	A
T	T	A	C	A	$	G	A
A	T	T	A	C	A	$	G

Finally we move first column to the very end

```r
bw.inverse[,n+1] <- bw.inverse[, 1]
bw.inverse       <- bw.inverse[,2:(n+1)]
colnames(bw.inverse)[n] = n

knitr::kable(bw.inverse)
```

1	2	3	4	5	6	7	8
$	G	A	T	T	A	C	A
A	$	G	A	T	T	A	C
A	C	A	$	G	A	T	T
A	T	T	A	C	A	$	G
C	A	$	G	A	T	T	A
G	A	T	T	A	C	A	$
T	A	C	A	$	G	A	T
T	T	A	C	A	$	G	A

One can also verify that bw.matrix and bw.inverse are in fact the same.

```r
knitr::kable(bw.inverse == bw.matrix)
```

1	2	3	4	5	6	7	8
TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE

Additionally we can encapsulate the Burrows-Wheeler transform in a form of a single function.

```r
BWT <- function(sequence){
  sequence  <- str_c(sequence, '$')
  sequences <- c(sequence)
  n         <- nchar(sequence)

  for (i in 1:(n-1)){
    sequence <- str_c(str_sub(sequence, 2, n),
                      str_sub(sequence, 1, 1))
    sequences <- c(sequences, sequence)
  }
  sequences <- sort(sequences)

  bw.matrix           <- data.frame(matrix(, n, n))
  colnames(bw.matrix) <- 1:n

  for (i in 1:n){
    bw.matrix[i, ] <- strsplit(sequences[i], split = '')[[1]]
  }
  return(paste(bw.matrix[,n], collapse = ''))
}
```

```r
BWT('GATTACA')
#> [1] "ACTGA$TA"
```

One can verify that this output is equal to result we obtained earlier.

Out of pure curiosity lets check the Burrows-Wheeler transform for a longer sequence.

```r
BWT('ATGCTCGTGCCATCATATAGCGCGCGCGATCTCTACGCGCG')
#> [1] "GTTTCCG$TCGGGGGAGGGTTGTCCTCCCCCCATCCAAACCAGA"
```

Please note that the input string has no identical characters at adjacent positions, whereas in the transformed sequence such situation appears quite often.

These substrings of identical characters will allow us represent the sequence in a more condensed manner and expediate pattern matching.

Bibliography

Donald Adjeroh, Tim Bell, A. M. (2008). *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer.