IBS – 3: Burrows-Wheeler Transform (Group A)

Abijith Pradeep – 2
G Rohith – 26
Lakshaya K – 39
Sanjay B – 56

Acknowledgements

This project would not have been possible without the support and guidance of Mr Sachin Kumar S who has provided us with sufficient knowledge and guidelines even during these unprecedented and trying times.

We would also like to thank our Computer Science and Engineering (Artificial Intelligence) department for giving us this opportunity to nurture and hone our skills.

Furthermore, we would like to thank the Amrita Vishwa Vidyapeetham management for ample resources to avail our project needs and a platform for communication within our team members and online lectures during this lockdown.

Contents

Abstract

The following report details upon the intersection of Bioinformatics and Burrows Wheeler Transform.

It includes the implementation of the Burrows Wheeler Transform in 4 coding languages: JavaScript, Python, Julia, and Scala. The theory behind the usage of the Burrows Wheeler Transform in compression and pattern matching is explained and the respective codes have been attached as well. The idea of using Burrows-Wheeler Transform for genome reconstruction was explored, and applications such as Pattern Matching, and Pattern Counting have been implemented.

Introduction and Problem Statement:

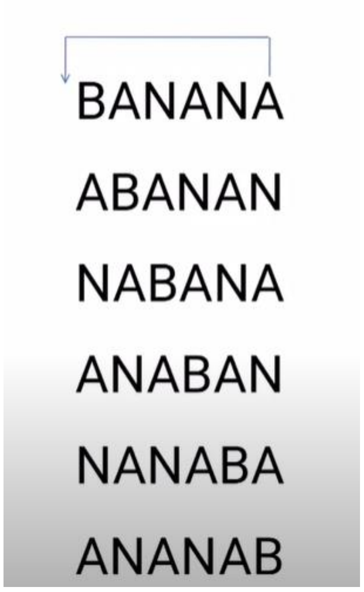
This project focuses upon the Burrows-Wheeler Transform and the applications of it, including pattern matching and pattern counting. The process of Burrows-Wheeler Transform has also been mapped with Bioinformatics, and experiments have been conducted to test the idea of using Burrows-Wheeler Transform for genome reconstruction.

Burrows-Wheeler Transform (BWT):

The Burrows-Wheeler Transform is an algorithm that aids in data transformation so that the transformed data can be compressed more easily. It performs a context-dependant permutation of all the symbols of the input data.

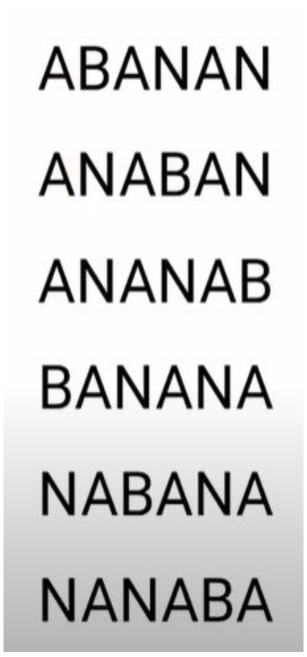
Similar contexts usually adjoin similar sets of symbols, and consequently the permuted data has extensive groupings of similar symbols and a far greater number of runs of single symbols compared to the original data. This is in turn useful for compression as long string sequences can be stored using lesser memory. Some of such further compression techniques include run-length encoding and Huffman encoding.

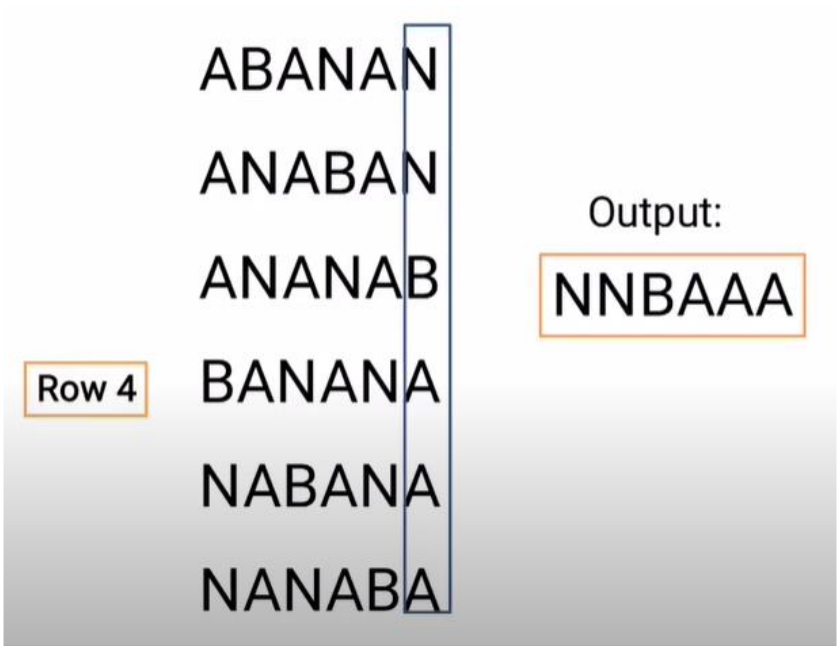
The most important application of BWT is in Bioinformatics wherein genomes do not have many runs but contain a large number of repeats. This is due to the fact that genome sequences are only made up of A, C, G, and T.



The process of Burrows-Wheeler transform begins by doing a cyclic rotation on the given string sequence.

The last letter of the string should be shifted to the front consecutively until a table with the same number of rows as the length is created.


ABANAN
ANABAN
ANANAB
BANANA
NABANA
NANABA



The second step is to sort the table lexicographically. Following which, the Burrows-Wheeler string can be obtained by taking the last column of the sorted table.

As seen from the image above, the original string “BANANA” has been transformed into “NNBAAA”. This transformed output has a greater number of reads (similar characters grouped together).

The transformed data can now be encoded using run-length encoding. Run length encoding is also a form of lossless data compression wherein runs of data are stored as a single data value and count.

For the example taken, NNBAAA will be run length encoded as 3N1B3A.

BANANA → NNBAAA → 3N1B3A

JavaScript Code and Explanation:

As explained in the previous section, the Burrows-Wheeler Transform has been split into sub-functions so as to illustrate the working of the algorithm clearly.

```
function bwt_table(given){
    var table = []
    for (var i = 0; i < given.length; i++){
      // word = last letter + first to last-1 letter
      var word = given.slice(-1) + given.slice(0, -1);
      // assigning given to word so as to perform cyclic rotation
        given = word;
    // adding each word to words_list
    table.push(word)
}
    return table.sort()
}
```

This function named *bwt_table* takes a string as the input parameter and creates the BWT table by doing cyclic rotation. A for loop has been used to perform cyclic rotations, wherein each rotated word is added to the *table* which is then sorted.

Cyclic rotation is performed by placing the last letter of the word before the remaining letters and doing this iteratively for the length of the string.

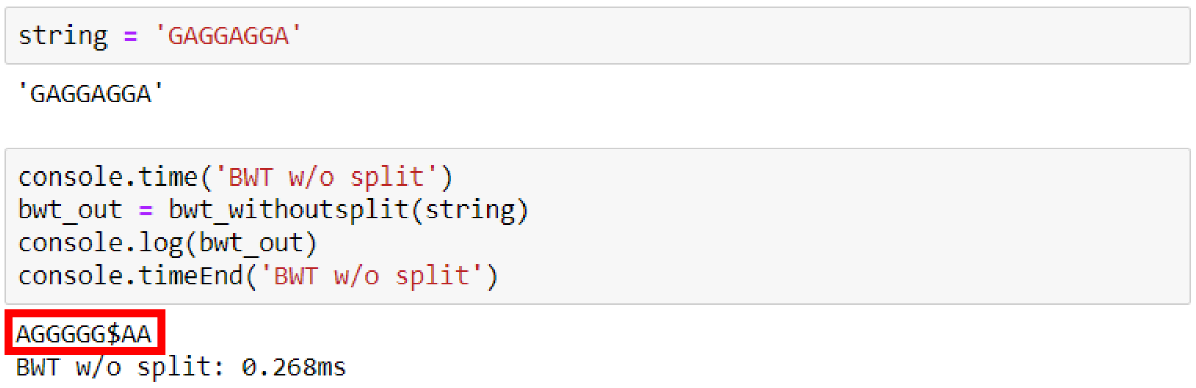
```
function bwt_withoutsplit(given_string){
    // appending $ to the string
    given_string += '$'
    // performing cyclic rotation and obtaining the BWT table
    table = bwt_table(given_string)
    // empty string for containing bwt string
    bwt = ''
    // taking the last element from each word from the sorted words list
    for (var j = 0; j < table.length; j++){
        bwt += (table[j].slice(-1))
    }
    return bwt
}
```

The *bwt_withoutsplit* function contains the previous sub-function and does the entire BWT process. It takes the given string as the input parameter, appends a $ sign which will assist during the inverse BWT, and calls the *bwt_table*. It then adds the last element from each row in the table into an empty string named *bwt*. This output is the final transformed string.

Similar to the previously explained JavaScript code, the Python, Julia, and Scala implementations also use the same algorithm.

To prove that the algorithms are accurate in all the languages, the string "GAGGAGGA" has been given as the input to ensure that the accurate output is being obtained. The output will next be cross verified with the Inverse Burrows Wheeler Transform.

JavaScript Output:



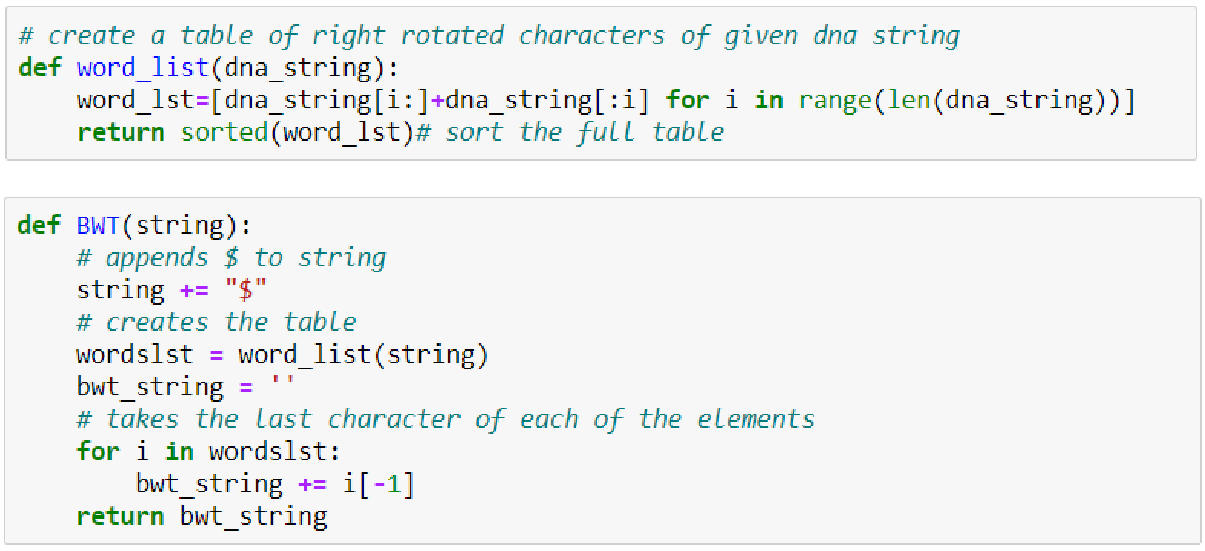
```
string = 'GAGGAGGA'
```

```
'GAGGAGGA'
```

```
console.time('BWT w/o split')
bwt_out = bwt_withoutsplit(string)
console.log(bwt_out)
console.timeEnd('BWT w/o split')
```

```
AGGGGG$AA
BWT w/o split: 0.268ms
```

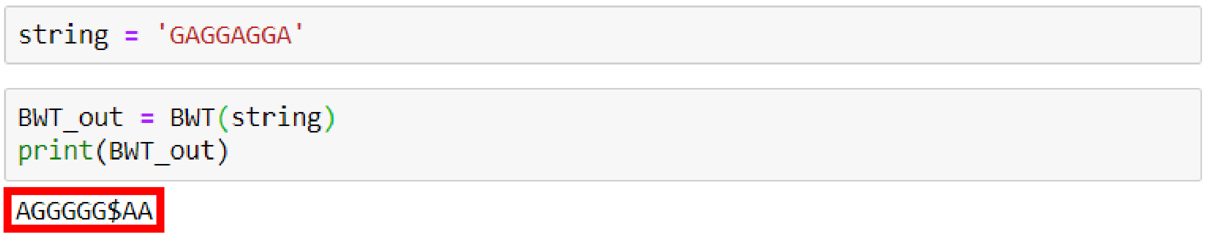
Python Code:



```python
# create a table of right rotated characters of given dna string
def word_list(dna_string):
    word_lst=[dna_string[i:]+dna_string[:i] for i in range(len(dna_string))]
    return sorted(word_lst)# sort the full table
```

```python
def BWT(string):
    # appends $ to string
    string += "$"
    # creates the table
    wordslst = word_list(string)
    bwt_string = ''
    # takes the last character of each of the elements
    for i in wordslst:
        bwt_string += i[-1]
    return bwt_string
```

Python Output:



```python
string = 'GAGGAGGA'
```

```python
BWT_out = BWT(string)
print(BWT_out)
```

```
AGGGGG$AA
```

Julia Code:

```julia
# creating a table of cyclic rotations
function BWT_table(word)
    word = string(word,'#') #adding hashtag sign is to track the last word
    l = length(word)
    T = []
    for i in 1:l
    push!(T,word)
    #Cyclic rotations of the sequence
    word = string(word[end],word[1:end - 1])
    end
return T
end
```

```julia
#function to get BWT of a Sequence
function BWT(string)
    List = BWT_table(string)
    #for sorting the list
    Lexicographical_order = sort!(List)
    #creating a empty string to append the BWT Sequence
    BWT_sequence = ""
    #for loop to append the sorted in BWT_sequence
    for i in 1:length(Lexicographical_order)
        BWT_sequence *= Lexicographical_order[i][end]
    end
    return BWT_sequence
end
```

Julia Output:

```julia
inp_string = "GAGGAGGA";
```

```julia
@time begin #to get time and memory allocation for BWT

out_BWT = BWT(inp_string)

end
```

```
  0.037895 seconds (84.16 k allocations: 4.108 MiB)
```

```
"AGGGGG#AA"
```

Scala Code:

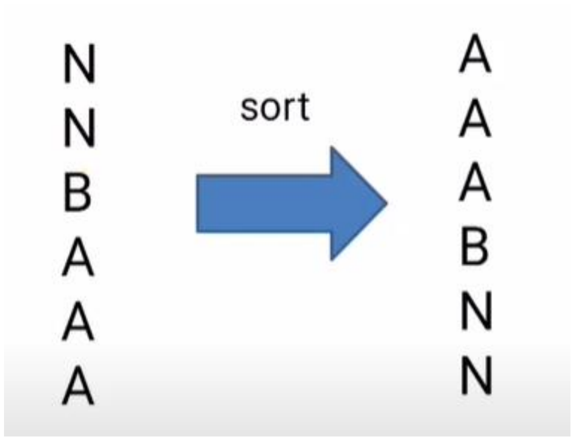
```scala
def BWT_table(Input: String): MutableList[String] = {
    var table = MutableList[String]()

   // create a table of right rotated characters of given string
   for (i <- 0 to Input.length - 1) {
     table += Input.substring(i, Input.length) + Input.substring(0, i)
   }
   table = sorting(table) // sorting the table
   return table
}



 def BWT(Input: String): String = {
   var InputSeq = Input + "$" // Add "$" to the end of the string
   var bwt = ""
   var table = BWT_table(InputSeq)
   for (i <- 0 to table.length - 1) {
     // takes the last character of each of the elements
     bwt += table(i).substring(table(i).length - 1, table(i).length)
   }
   return bwt
 }
```

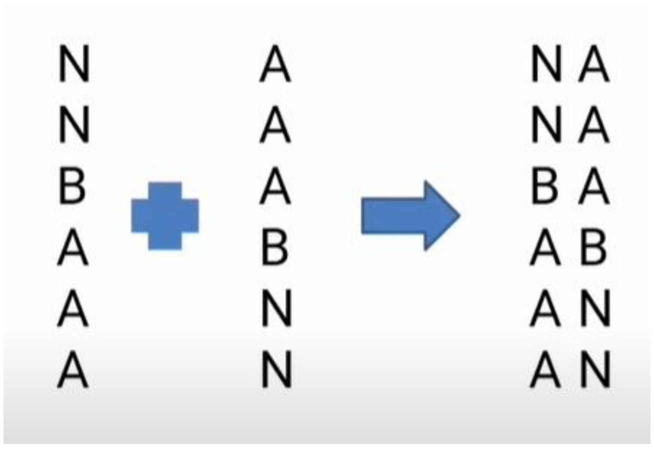
Scala Output:

```
var str = "GAGGAGGA"                                    //> str  : String = GAGGAGGA
var BWT_out = BWT(str)                                  //> BWT_out  : String = AGGGGG$AA
```

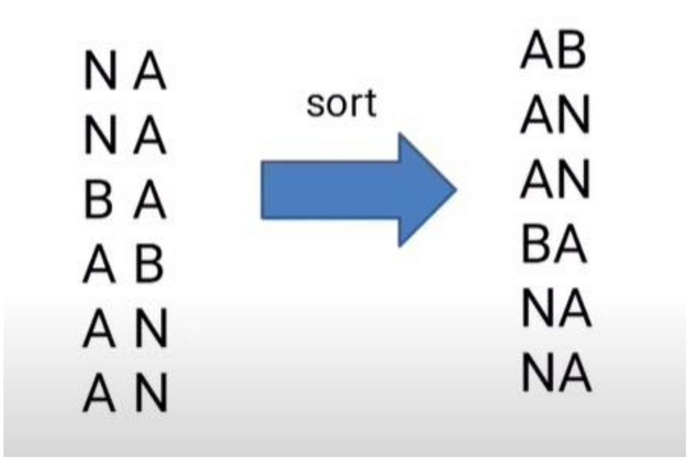
Inverse BWT (IBWT):

The inverse Burrows-Wheeler Transform is used to obtain the original string using the transformed string.

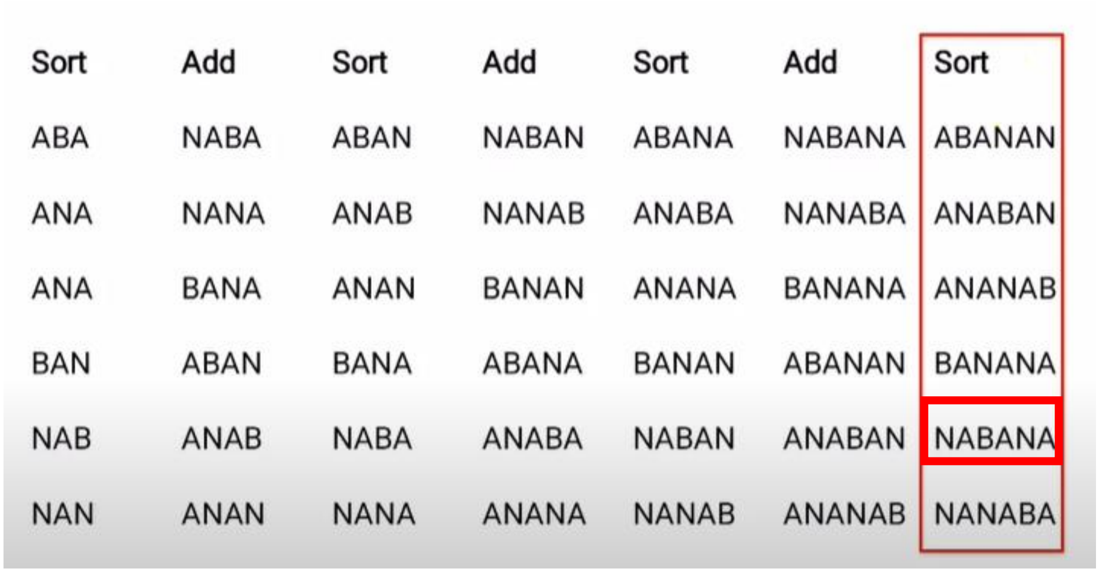




The obtained BWT string should be first be sorted and stored. The BWT string should then again be concatenated with the sorted string.



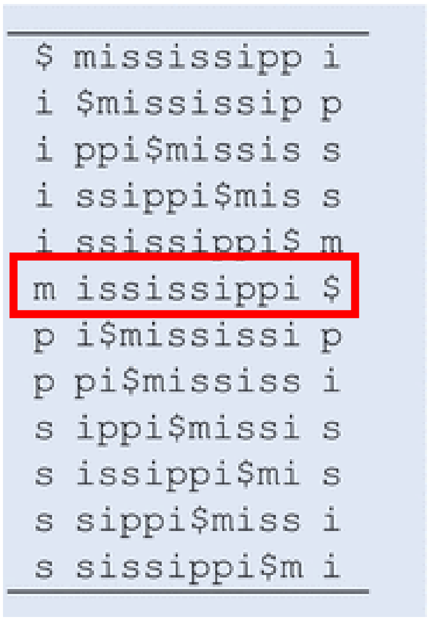
The appending columns should again be sorted.



Sort	Add	Sort	Add	Sort	Add	Sort
ABA	NABA	ABAN	NABAN	ABANA	NABANA	ABANAN
ANA	NANA	ANAB	NANAB	ANABA	NANABA	ANABAN
ANA	BANA	ANAN	BANAN	ANANA	BANANA	ANANAB
BAN	ABAN	BANA	ABANA	BANAN	ABANAN	BANANA
NAB	ANAB	NABA	ANABA	NABAN	ANABAN	NABANA
NAN	ANAN	NANA	ANANA	NANAB	ANANAB	NANABA

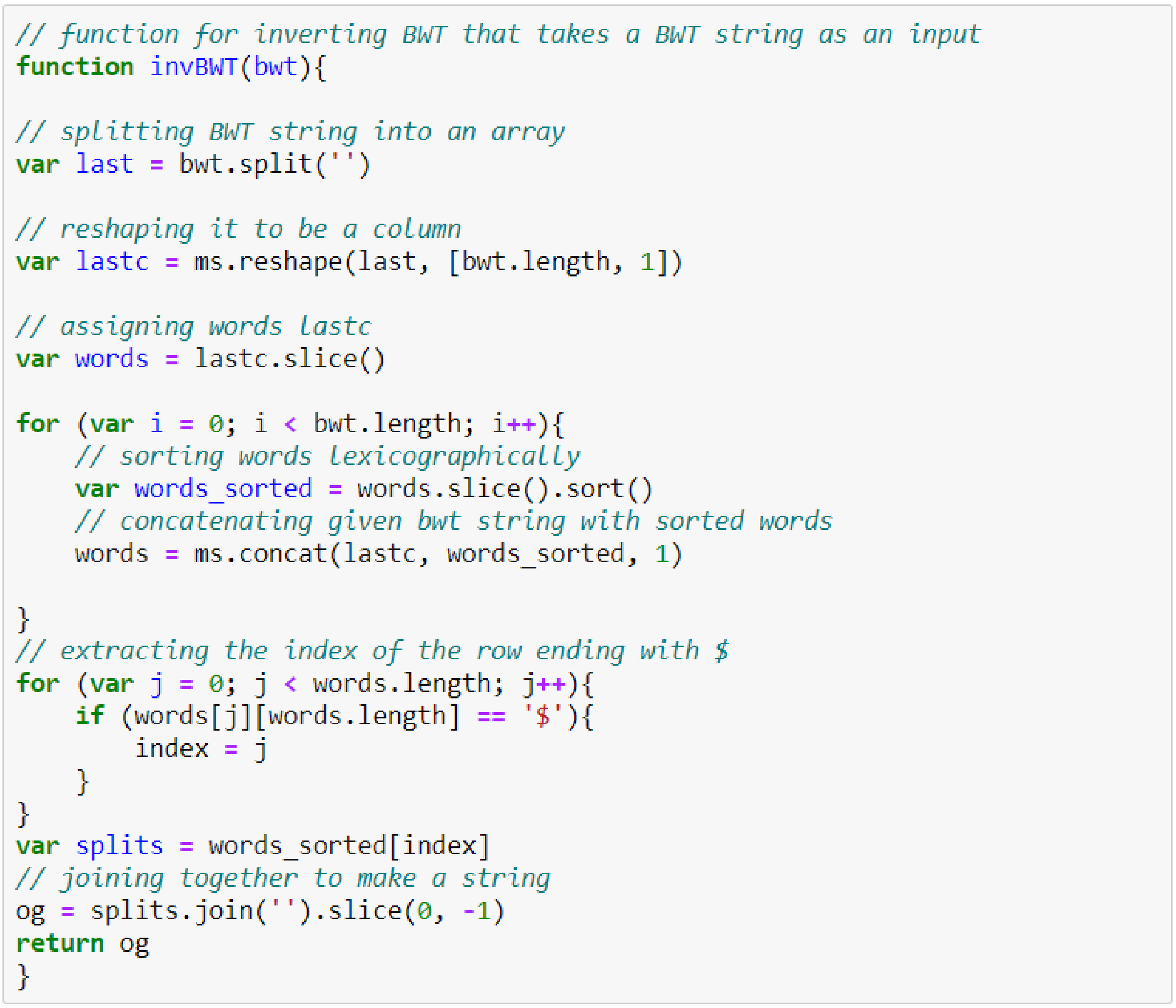
Repeating this process until the number of columns is equal to the length of the string will result in the BWT table that was obtained during the forward BWT process.

In the algorithm implemented in this project, the word with the marker ‘$’ as the last letter of the word will contain the required string. This can be explained using the image below, which shows that the original string “Mississippi” has been obtained by choosing the word wherein ‘$’ is the last symbol.



$	mississipp	i
i	$mississip	p
i	ppi$missis	s
i	ssippi$mis	s
i	ssissippi$	m
m	ississippi	$
p	i$mississi	p
p	pi$mississ	i
s	ippi$missi	s
s	issippi$mi	s
s	sippi$miss	i
s	sissippi$m	i

JavaScript Code and Explanation:



```
// function for inverting BWT that takes a BWT string as an input
function invBWT(bwt){

// splitting BWT string into an array
var last = bwt.split('')

// reshaping it to be a column
var lastc = ms.reshape(last, [bwt.length, 1])

// assigning words lastc
var words = lastc.slice()

for (var i = 0; i < bwt.length; i++){
    // sorting words lexicographically
    var words_sorted = words.slice().sort()
    // concatenating given bwt string with sorted words
    words = ms.concat(lastc, words_sorted, 1)

}
// extracting the index of the row ending with $
for (var j = 0; j < words.length; j++){
    if (words[j][words.length] == '$'){
        index = j
    }
}
var splits = words_sorted[index]
// joining together to make a string
og = splits.join('').slice(0, -1)
return og
}
```

A function named *invBWT* has been created which takes the BWT string as the input parameter. The string is split into a vector and reshaped to be a column. A for loop is then used to consecutively sort and concatenate the BWT string. The resultant BWT table is stored in *words_sorted*. A for loop is used to iterate through the BWT table to find the index of the row which ends with the marker '$'. The original string can then be obtained using this index.

The same logic has been utilised to implement the inverse Burrows Wheeler Transform in the remaining languages.

The output of the BWT 'AGGGGG$AA' has been fed into the implementation in all 4 languages to ensure that the original string of "GAGGAGGA" is being obtained.

JavaScript Output:

```
console.time('IBWT w/o split')

console.log("BWT String:", bwt_out)
ibwt_out = invBWT(bwt_out)
console.log("Reconstructed string:", ibwt_out)

console.timeEnd('IBWT w/o split')
```

```
BWT String: AGGGGG$AA
Reconstructed string: GAGGAGGA
IBWT w/o split: 3.285ms
```

```
// checking if output of IBWT is same as input string
console.log(ibwt_out==string)
```

```
true
```

Python Code:

```
def inverseBWT(bwt):
    # initialize the table from t
    table = ['' for c in bwt]
    for j in range(len(bwt)):
        # insert the BWT as the first column
        table = sorted([c+table[i] for i, c in enumerate(bwt)])
    # return the row that ends with '$'
    given = table[bwt.index('$')]
    return given[:len(given)-1] # returning without $ sign
```

Python Output:

```
print("BWT String:", BWT_out)
IBWT_out = inverseBWT(BWT_out)
print("Reconstructed string:", IBWT_out)
```

```
BWT String: AGGGGG$AA
Reconstructed string: GAGGAGGA
```

```
# Checking if output of IBWT is equal to input string
print(IBWT_out == string)
```

```
True
```

Julia Code:

```
# function to get the IBWT of a BWT sequence
function IBWT(BWT_sequence)
    BWT = []
    # for loop to append BWT array with the elements of BWT_Sequence
    for i in 1:length(BWT_sequence)
    push!(BWT,BWT_sequence[i])
    end

    len = length(BWT)
    # Creating a empty array with the length of the BWT sequence
    IBWT = Array{String}(undef,len)
    # pre-initialising an array temp for BWT
    temp = BWT

    for i in 1:len
    temp_1 = copy(temp)
    # sorting the copy of BWT array lexicographically
    temp_1 = Bub_sort(temp_1)
    for j in 1:len
    # concatenating the bwt string with the sorted words
    IBWT[j] = BWT[j] * temp_1[j]
    end
    # reinitialising the temp array as IBWT
    temp = IBWT
    end

    for i in 1:len
    IBWT[i] = IBWT[i][2:end]
    end
    # to return the first string of the list
    return IBWT[1][2:end]
end
```

Julia Output:

```
println("BWT String: ", out_BWT)
@time begin #to get time and memory allocation for IBWT

out_IBWT = IBWT(out_BWT)
println("Reconstructed String: ", out_IBWT)
end
```

```
BWT String: AGGGGG#AA
Reconstructed String: GAGGAGGA
  0.000269 seconds (189 allocations: 7.375 KiB)
```

Scala Code:

```
def IBWT(BWT_out: String): String = {
  var table = ListBuffer[String]() // Make empty table
  for (i <- 0 to BWT_out.length - 1) { // Add a column
    table += ""
  }
  for (i <- 0 to BWT_out.length - 1) {
    for (j <- 0 to BWT_out.length - 1) {
    // Concatenating given BWT string w sorted words
      table(j) = BWT_out(j) + table(j)
    }
    table = table.sorted
  }
  var inverse_bwt = ""
  for (i <- 0 to table.length - 1) {
  // Finding the word that ends with $
    if (table(i).endsWith("$")) {
      inverse_bwt = inverse_bwt + table(i).substring(0, table(i).length - 1) // Returning w/o $
    }
  }
  return inverse_bwt
}
```

Scala Output:

```
print(BWT_out)                                  //> AGGGGG$AA
var IBWT_out = IBWT(BWT_out)                    //> IBWT_out  : String = GAGGAGGA
print(str == IBWT_out)                          //> true
```

Run-Length Encoding and Decoding:

After the output of the Burrows-Wheeler Transform has been obtained, run-length encoding can be used to compress the data and decoding can be used to retrieve the data. This has been implemented in JavaScript and Python.

The encoding has been done by counting the number of continuous occurrences of a character by using a while loop. The output consists of alternating integer value and symbol pairs wherein the integer denotes the number of that particular symbol that was present in the string.

The decoding iterates through the encodes string and simply repeats the symbols in accordance with its given respective integer value.

JavaScript Implementation:

```
function encode(st){
    n = st.length
    i = 0
    encoded_out = ''
    while (i < n){
        //Count occurrences of current character
        count = 1
        while (i < n-1 && st[i] == st[i + 1]){
            count += 1
            i += 1
        }
        i += 1
        //Print character and its count
        encoded_out += (count+st[i - 1])
    }
    return encoded_out
}
```

```
console.log("BWT string:", bwt_out)
encoded_out = encode(bwt_out)
console.log("Compressed BWT string", encoded_out)
```

```
BWT string: AGGGGG$AA
Compressed BWT string 1A5G1$2A
```

```
function decode(st){
    decoded_out = ''
    num = ''
    for (var i = 0; i < st.length; i++){
        if (isNaN(st[i]) == true ){
            decoded_out += st[i].repeat(parseInt(num))
            num = ''
        }
        else num += st[i]
    }
    return decoded_out
}
```

```
console.log("Encoded String:", encoded_out)
decoded_out = decode(encoded_out)
console.log("Decoded String:", decoded_out)
```

```
Encoded String: 1A5G1$2A
Decoded String: AGGGGG$AA
```

Python Implementation:

The Python implementation also uses the same logic that has been explained previously.

```python
def encode(st):
    n = len(st)
    i = 0
    out = ''
    while i < n:

        # Count occurrences of current character
        count = 1
        while (i < n-1  and st[i] == st[i + 1]):
            count += 1
            i += 1
        i += 1

        # Print character and its count
        out += str(count)+st[i - 1]
    return out
```

```python
print("Original string:", BWT_out)
encoded_bwt = encode(BWT_out)
print("Encoded string:", encoded_bwt)
```

```
Original string: AGGGGG$AA
Encoded string: 1A5G1$2A
```

```python
def decode(s):
    output = ""
    num = ''
    for i in s:
        if not i.isnumeric():
            output+=i*int(num)
            num=""
        else:
            num+=i
    return output
```

```python
print("Encoded string:", encoded_bwt)
decoded_bwt = decode(encoded_bwt)
print("Decoded string:", decoded_bwt)
```

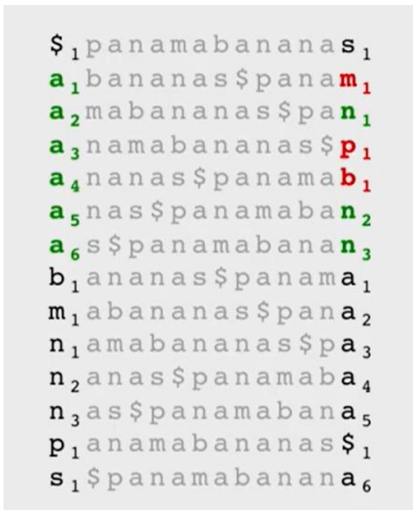
```
Encoded string: 1A5G1$2A
Decoded string: AGGGGG$AA
```

Pattern Matching:

Pattern Matching is the process of finding the position of a particular pattern in the original genome sequence using only the BWT string. The process can be summarised using the following steps:

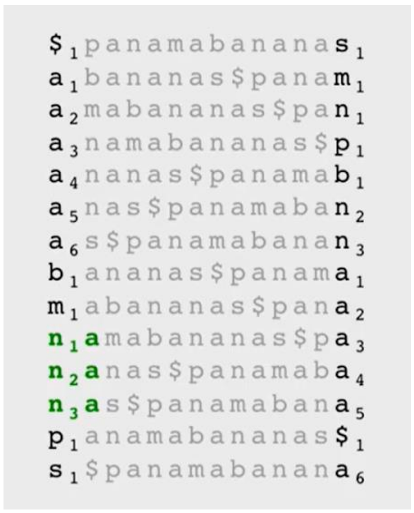
1) Using the BWT string, the BWT table can be constructed using cyclic rotations
2) The index of the last letter of the given pattern in the first column is computed and the algorithm checks if these rows end with the 2nd last letter in the pattern
3) This process is done iteratively for the length of the given pattern

The following example of searching for ‘ana’ in ‘panamabananas’ will be used to explain the algorithm.



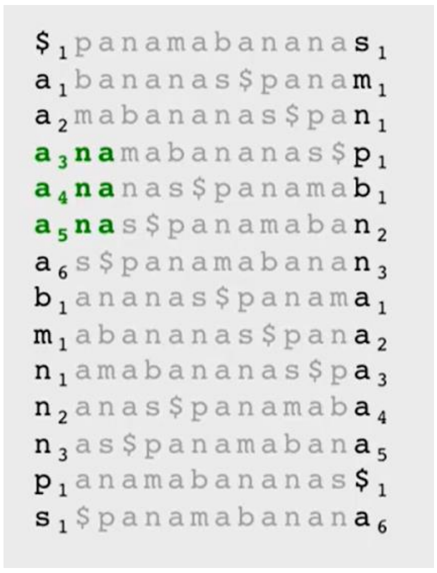
$\$_1$panamabanana$s_1$
$a_1$bananas\$panam_1
$a_2$mabananas\$pan_1
$a_3$namabananas\$$p_1$
$a_4$nanas\$panamab_1
$a_5$nas\$panamaban_2
a_6s\$panamabanan_3
$b_1$ananas\$panama_1
$m_1$abananas\$pana_2
$n_1$amabananas\$pa_3
$n_2$anas\$panamaba_4
$n_3$as\$panamabana_5
$p_1$anamabananas$\$_1$
s_1\$panamabanana_6

Step 1:
Identifying rows beginning with last element of pattern
In this case: the last letter of the given pattern is ‘A’ and so the rows starting with ‘A’ will be highlighted. As we can see we have 6 occurrences of A.



$\$_1$panamabanana$s_1$
$a_1$bananas\$panam_1
$a_2$mabananas\$pan_1
$a_3$namabananas\$$p_1$
$a_4$nanas\$panamab_1
$a_5$nas\$panamaban_2
a_6s\$panamabanan_3
$b_1$ananas\$panama_1
$m_1$abananas\$pana_2
$n_1$amabananas\$pa_3
$n_2$anas\$panamaba_4
$n_3$as\$panamabana_5
$p_1$anamabananas$\$_1$
s_1\$panamabanana_6

Step 2:
The algorithm then checks which of these 6 occurrences ends with the 2nd last letter which is N. The search will now be limited from the entire 14 rows to the highlighted 6 rows to 3 rows.



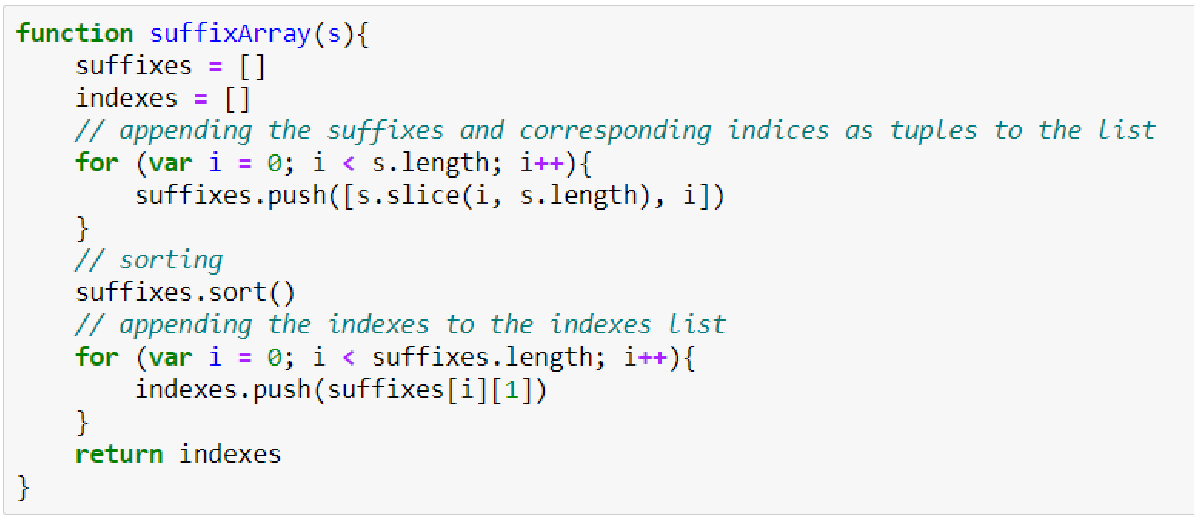
$\$_1$panamabanana$s_1$
$a_1$bananas\$panam_1
$a_2$mabananas\$pan_1
$a_3$namabananas\$$p_1$
$a_4$nanas\$panamab_1
$a_5$nas\$panamaban_2
a_6s\$panamabanan_3
$b_1$ananas\$panama_1
$m_1$abananas\$pana_2
$n_1$amabananas\$pa_3
$n_2$anas\$panamaba_4
$n_3$as\$panamabana_5
$p_1$anamabananas$\$_1$
s_1\$panamabanana_6

Step 3:
Using the 3 rows starting with N, the algorithm checks if they end with A which is the first letter.

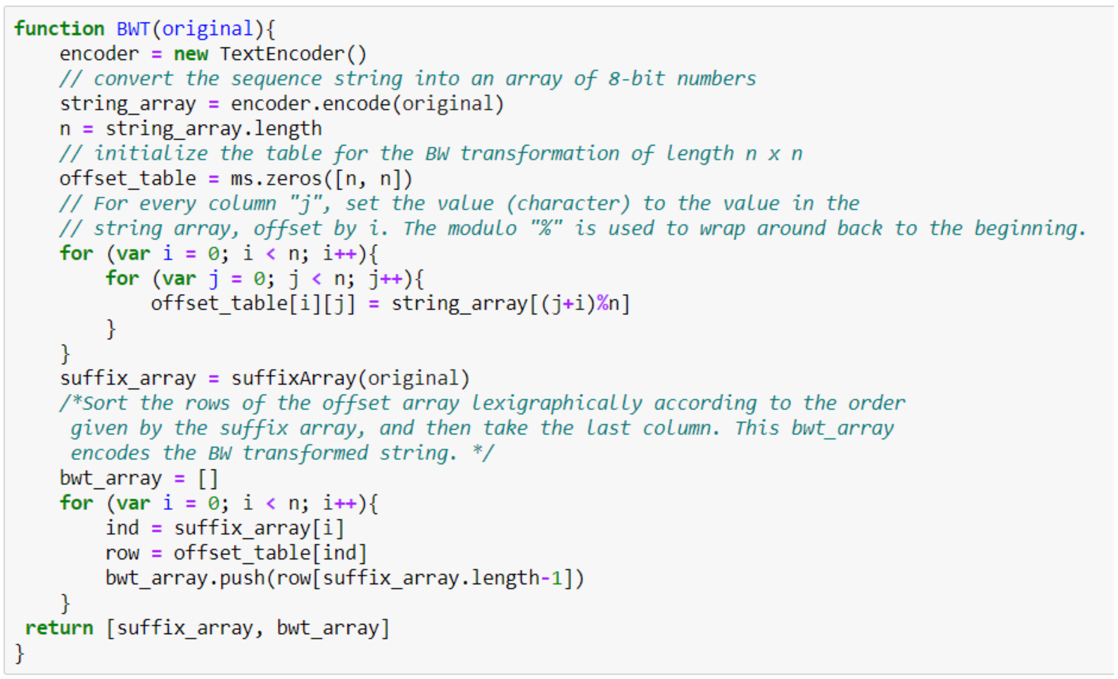
It can then be concluded that the rows containing our pattern are the ones starting with a3, a4 and a5.

This will be extremely useful in the field of bioinformatics as it allows for researchers to find the location of particular kmers in the long genome sequence. Pattern Matching has been implemented using JavaScript and the pattern ‘CG’ has been searched in the sequence ‘ACGTCGTCGT’.

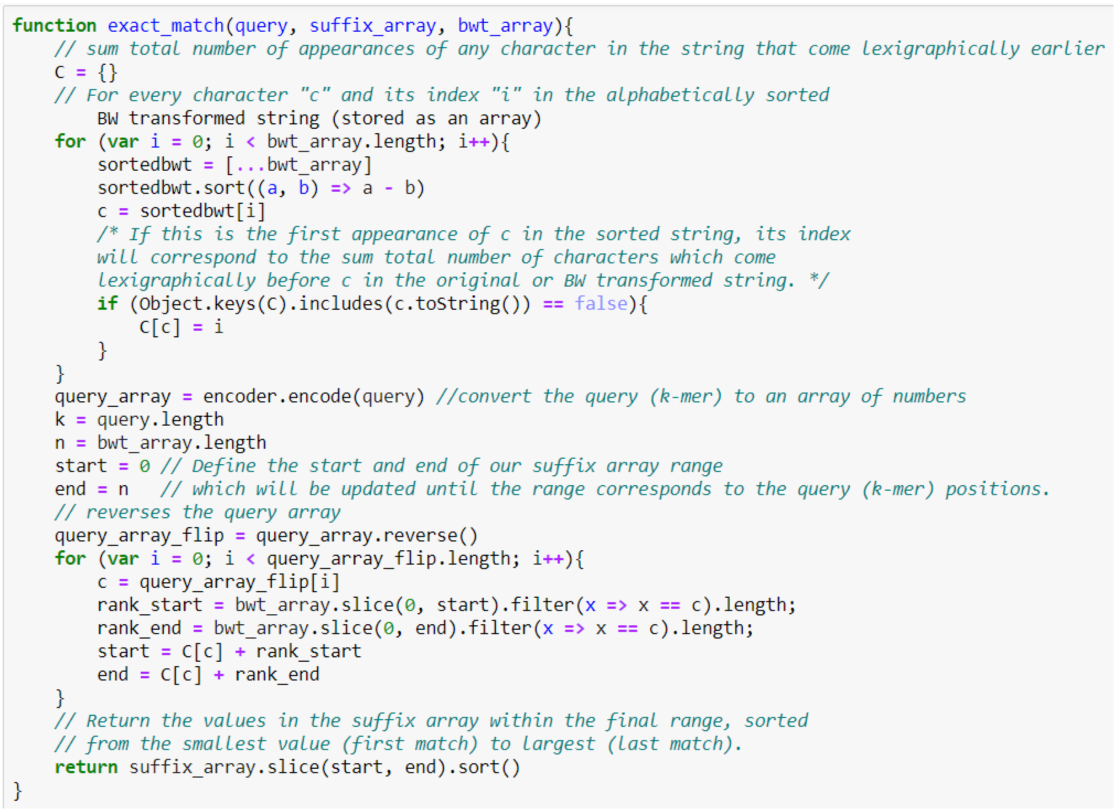
JavaScript Code & Output:



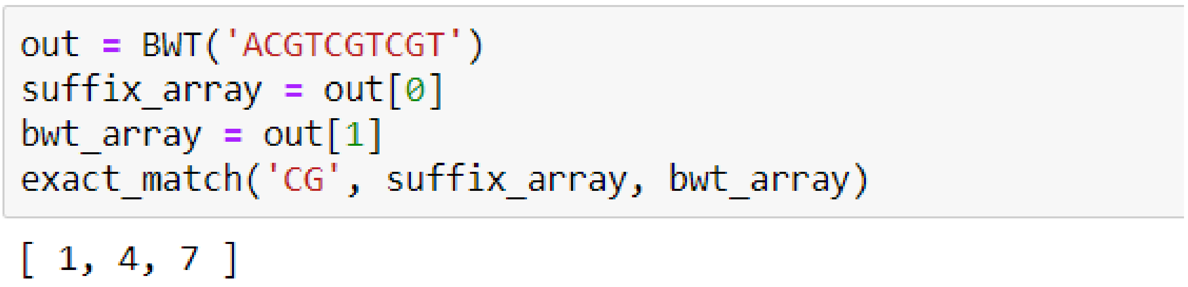
```
function suffixArray(s){
    suffixes = []
    indexes = []
    // appending the suffixes and corresponding indices as tuples to the list
    for (var i = 0; i < s.length; i++){
        suffixes.push([s.slice(i, s.length), i])
    }
    // sorting
    suffixes.sort()
    // appending the indexes to the indexes list
    for (var i = 0; i < suffixes.length; i++){
        indexes.push(suffixes[i][1])
    }
    return indexes
}
```



```
function BWT(original){
    encoder = new TextEncoder()
    // convert the sequence string into an array of 8-bit numbers
    string_array = encoder.encode(original)
    n = string_array.length
    // initialize the table for the BW transformation of length n x n
    offset_table = ms.zeros([n, n])
    // For every column "j", set the value (character) to the value in the
    // string array, offset by i. The modulo "%" is used to wrap around back to the beginning.
    for (var i = 0; i < n; i++){
        for (var j = 0; j < n; j++){
            offset_table[i][j] = string_array[(j+i)%n]
        }
    }
    suffix_array = suffixArray(original)
    /*Sort the rows of the offset array lexigraphically according to the order
     given by the suffix array, and then take the last column. This bwt_array
     encodes the BW transformed string. */
    bwt_array = []
    for (var i = 0; i < n; i++){
        ind = suffix_array[i]
        row = offset_table[ind]
        bwt_array.push(row[suffix_array.length-1])
    }
 return [suffix_array, bwt_array]
}
```



```
function exact_match(query, suffix_array, bwt_array){
    // sum total number of appearances of any character in the string that come lexigraphically earlier
    C = {}
    // For every character "c" and its index "i" in the alphabetically sorted
        BW transformed string (stored as an array)
    for (var i = 0; i < bwt_array.length; i++){
        sortedbwt = [...bwt_array]
        sortedbwt.sort((a, b) => a - b)
        c = sortedbwt[i]
        /* If this is the first appearance of c in the sorted string, its index
        will correspond to the sum total number of characters which come
        lexigraphically before c in the original or BW transformed string. */
        if (Object.keys(C).includes(c.toString()) == false){
            C[c] = i
        }
    }
    query_array = encoder.encode(query) //convert the query (k-mer) to an array of numbers
    k = query.length
    n = bwt_array.length
    start = 0 // Define the start and end of our suffix array range
    end = n   // which will be updated until the range corresponds to the query (k-mer) positions.
    // reverses the query array
    query_array_flip = query_array.reverse()
    for (var i = 0; i < query_array_flip.length; i++){
        c = query_array_flip[i]
        rank_start = bwt_array.slice(0, start).filter(x => x == c).length;
        rank_end = bwt_array.slice(0, end).filter(x => x == c).length;
        start = C[c] + rank_start
        end = C[c] + rank_end
    }
    // Return the values in the suffix array within the final range, sorted
    // from the smallest value (first match) to largest (last match).
    return suffix_array.slice(start, end).sort()
}
```



```
out = BWT('ACGTCGTCGT')
suffix_array = out[0]
bwt_array = out[1]
exact_match('CG', suffix_array, bwt_array)
```

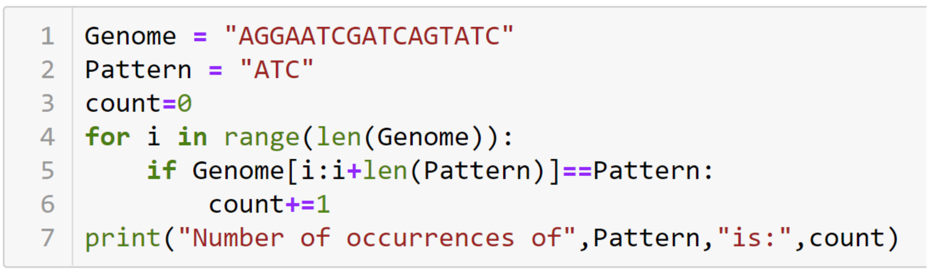
```
[ 1, 4, 7 ]
```

Pattern Counting:

Pattern counting is the process of counting the number of occurrences a certain pattern has in a genomic sequence. In the case that the pattern does not occur then it will be noted as 0 occurrences and will print out pattern doesn't exist.

One might ask why we must approach this problem with the Burrow Wheelers Transform method rather than the brute force sliding approach. The brute force sliding approach is easier to implement but it has certain demerits. It takes both high amount of memory and time for the results to be formed. The approach that we have adopted relatively takes lesser memory and is way faster than the brute force method.

Brute Force Sliding Method:



```python
Genome = "AGGAATCGATCAGTATC"
Pattern = "ATC"
count=0
for i in range(len(Genome)):
    if Genome[i:i+len(Pattern)]==Pattern:
        count+=1
print("Number of occurrences of",Pattern,"is:",count)
```

```
Number of occurrences of ATC is: 3
```

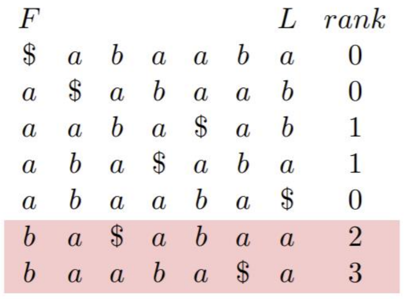
BWT Approach:

Taking the pattern 'aba' in the string abaaba:

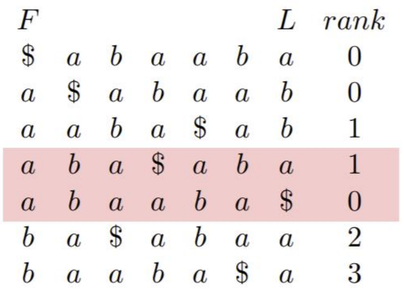
The algorithm proceeds by first finding the rows beginning with the shortest suffix of P: a in this case. The first column is part of our index, so this is trivial. These are the rows in the 0-based range [1, 4].

F						L	$rank$
$	a	b	a	a	b	a	0
a	$	a	b	a	a	b	0
a	a	b	a	$	a	b	1
a	b	a	$	a	b	a	1
a	b	a	a	b	a	$	0
b	a	$	a	b	a	a	2
b	a	a	b	a	$	a	3

Now we find rows beginning with the final suffix, aba. Again, we look at the shaded characters in the last column. We see that the occurrences of ba are preceded by a2 and a3, giving us the range of rows prefixed by P:



F						L	$rank$
\$	a	b	a	a	b	a	0
a	\$	a	b	a	a	b	0
a	a	b	a	\$	a	b	1
a	b	a	\$	a	b	a	1
a	b	a	a	b	a	\$	0
b	a	\$	a	b	a	a	2
b	a	a	b	a	\$	a	3



F						L	$rank$
\$	a	b	a	a	b	a	0
a	\$	a	b	a	a	b	0
a	a	b	a	\$	a	b	1
a	b	a	\$	a	b	a	1
a	b	a	a	b	a	\$	0
b	a	\$	a	b	a	a	2
b	a	a	b	a	\$	a	3

This is called backwards matching. In short, we apply the LF Mapping repeatedly to find the range of rows prefixed by successively longer proper suffixes of P until the range becomes empty, in which case P does not occur in T, or until we run out of suffixes. If we run out of suffixes, the size of the range equals the number of times P occurs in T. Hence the number of occurrences of aba in the string is 2. This can be derived from the fact that two rows have been highlighted at the end.

Python Code & Output:

```python
def countMatches(bw, p):
    """ Given BWT(T) and a pattern string p, return the number of times
    p occurs in T. """
    ranks, tots = rankBwt(bw)
    first = firstCol(tots)
    l, r = first[p[-1]]
    i = len(p)-2
    while i >= 0 and r > l:
        c = p[i]
        # scan from left, looking for occurrences of c
        j = l
        while j < r:
            if bw[j] == c:
                l = first[c][0] + ranks[j]
                break
            j += 1
        if j == r:
            l = r
            break # no occurrences -> no match
        r -= 1
        while bw[r] != c:
            r -= 1
        r = first[c][0] + ranks[r] + 1
        i -= 1
    return r - l
```

```
occurences of ACA are: 1
occurences of CAA are: 1
occurences of AAT are: 1
occurences of ATG are: 2
occurences of TGA are: 1
occurences of GAG are: 1
occurences of AGG are: 1
occurences of GGT are: 1
occurences of GTC are: 1
occurences of TCA are: 1
occurences of CAC are: 1
occurences of ACT are: 1
occurences of CTA are: 1
occurences of TAT are: 1
```

BWT & IBWT after KMER Decomposition:

A DNA string can be decomposed into kmers and Burrow-Wheelers Transform can be performed on each of the kmers.

Code - JavaScript:

```
function splitkmers(given, k){
// pre-initialising an array for the kmers
var kmers = [];
// for loop that iterates through the DNA string and extracts the kmers
for (var i = 0; i <= (given.length - k); i++){
        // adding the kmers into the kmers array by slicing
       kmers.push(given.slice(i, i+k));
}
    return kmers
}
```

```
function sortkmers(given){
    // storing the kmers into a new var sortedkmers
    var sortedkmers = kmers;
    // for loop which loops through kmers loop
    for (var i = 0; i < kmers.length; i++) {
    // nested for loop which starts at the next element
        for (var j =i+1; j < kmers.length; j++) {
        // checks if kmers[i] is greater than kmers[i+1]
            if (kmers[i].localeCompare(kmers[j]) > 0) {
            // if yes, it stores kmers[i] into a var temp
            var temp = kmers[i];
            // stores kmers[i+1] into the place of kmers[i]
            sortedkmers[i] = kmers[j];
            // stores kmers[i] into place of kmers[i+1]
            sortedkmers[j]= temp;
            }
        }
    }
    return sortedkmers
}
```

```
function bwt_withsplit(given_string, k){
    // splitting a string into kmers and sorting
    kmers = splitkmers(given_string, k)
    kmers = sortkmers(kmers)

    kmer_d = []
    // appending each kmer with an appended $
    for (var i = 0; i < kmers.length; i++){
        kmer_d.push(bwt_table(kmers[i]+'$'))
    }

    bwt_kmers = []
    // performing bwt on each of the kmers
    for (var j = 0; j < kmer_d.length; j++){
        bwt_kmers.push(bwt_transform(kmer_d[j]))
    }
return bwt_kmers
}
```

```
console.time('BWT with split')
bwtout_split = bwt_withsplit(string, k)
console.log(bwtout_split)
console.timeEnd('BWT with split')
```

```
[ 'G$GA', 'G$GA', 'GGA$', 'GGA$', 'AGG$', 'AGG$' ]
BWT with split: 4.128ms
```

```
function invBWT_split(bwt_list){
    inv = []
    // performing inverse BWT on the obtained BWT strings for each kmer
    for (var i = 0; i<bwt_list.length ; i++){
        inv.push(invBWT(bwt_list[i]))
    }
    return inv
}
```

```
console.time('IBWT with split')

IBWTout_split = invBWT_split(bwtout_split)
console.log(IBWTout_split)

console.timeEnd('IBWT with split')
```

```
[ 'AGG', 'AGG', 'GAG', 'GAG', 'GGA', 'GGA' ]
IBWT with split: 0.701ms
```

Code – Python:

```python
# function which splits a string into kmers
def kmer_gen(st,k):
    kmer=[]
    for i in range(len(st)-k+1):
        kmer.append(st[i:i+k])
    return kmer
```

```python
# function to sort kmers lexicographically
def sort_kmer(km):
    kmer=km.copy()
    for i in range(len(kmer)):
        for j in range(1,len(kmer)-i):
            if kmer[j-1] > kmer[j]:
                (kmer[j-1], kmer[j]) = (kmer[j], kmer[j-1])
    return kmer
```

```python
def BWT_split(string, k):
    # generates kmers from string
    kmers = kmer_gen(string, k)
    # sorts kmers
    kmers = sort_kmer(kmers)
    #kmers.sort()
    BWT_split_out = []
    # performs BWT on each of the kmer
    for i in kmers:
        BWT_split_out.append(BWT(i))
    return BWT_split_out
```

```python
BWT_out_split = BWT_split(dna_string, 3)
print(BWT_out_split)
```

```
['G$GA', 'G$GA', 'GGA$', 'GGA$', 'AGG$', 'AGG$']
```

```python
def inverseBWT_split(BWT_kmers):
    IBWT_kmers = []
    # performing inverse BWT on each of BWT kmers
    for i in BWT_kmers:
        IBWT_kmers.append(inverseBWT(i))
    return IBWT_kmers
```

```python
IBWT_out_split = inverseBWT_split(BWT_out_split)
print(IBWT_out_split)
```

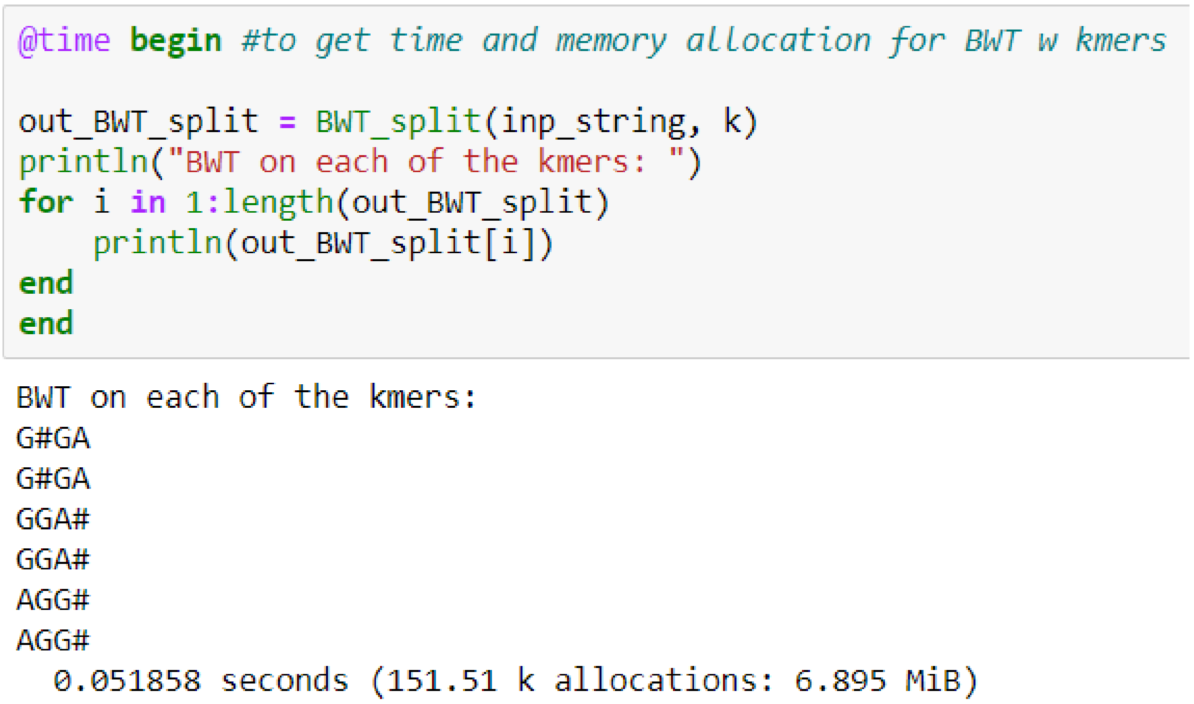
```
['AGG', 'AGG', 'GAG', 'GAG', 'GGA', 'GGA']
```

Code – Julia:

```
#to split the DNA String into given no. of k-mers
function splitkmers(string, k)
    out = []
    # for loop that iterates through the DNA string and extracts the kmers
    for i in 1:(length(string)-k+1)
        out = push!(out, string[i:i+k-1])
    end
    #to sort the k-mers in lexicographical order
    out = sort!(out)
    return out
end
```

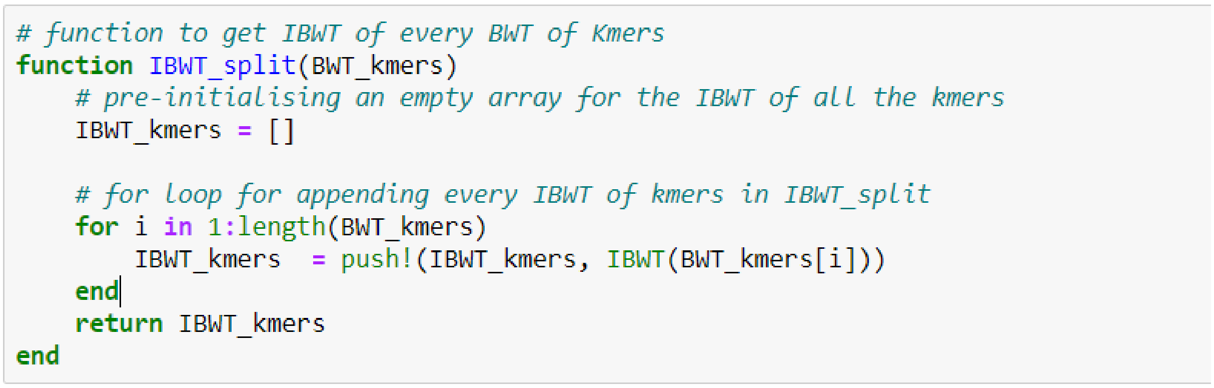
```
#function for sorting by Bubble sort algorithm
function Bub_sort(list)
    for i in 1:length(list)
    for j in 1:length(list)
    #to compare the two adjacent elements and swap them
    if list[i] < list[j]
        temp = list[i]
        list[i] = list[j]
        list[j] = temp
    end
    end
    end
    return list
end
```

```
#function to get the BWT for each kmer in the list
function BWT_split(string, k)
    kmers = splitkmers(string, k)
    #pre-initialising an empty array for the BWT of all the kmers
    BWTsplit_out = []
    #for loop for appending every BWT of kmers in BWTsplit_out
    for i in 1:length(kmers)
        BWTsplit_out = push!(BWTsplit_out, BWT(kmers[i]))
    end
    return BWTsplit_out
end
```

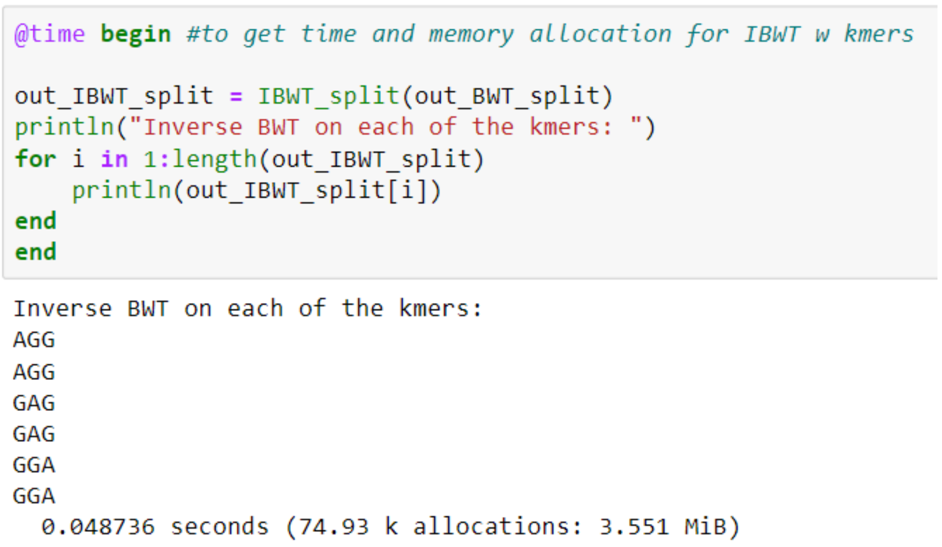


```julia
@time begin #to get time and memory allocation for BWT w kmers

out_BWT_split = BWT_split(inp_string, k)
println("BWT on each of the kmers: ")
for i in 1:length(out_BWT_split)
    println(out_BWT_split[i])
end
end
```

```
BWT on each of the kmers: 
G#GA
G#GA
GGA#
GGA#
AGG#
AGG#
  0.051858 seconds (151.51 k allocations: 6.895 MiB)
```



```julia
# function to get IBWT of every BWT of Kmers
function IBWT_split(BWT_kmers)
    # pre-initialising an empty array for the IBWT of all the kmers
    IBWT_kmers = []

    # for loop for appending every IBWT of kmers in IBWT_split
    for i in 1:length(BWT_kmers)
        IBWT_kmers  = push!(IBWT_kmers, IBWT(BWT_kmers[i]))
    end
    return IBWT_kmers
end
```



```julia
@time begin #to get time and memory allocation for IBWT w kmers

out_IBWT_split = IBWT_split(out_BWT_split)
println("Inverse BWT on each of the kmers: ")
for i in 1:length(out_IBWT_split)
    println(out_IBWT_split[i])
end
end
```

```
Inverse BWT on each of the kmers: 
AGG
AGG
GAG
GAG
GGA
GGA
  0.048736 seconds (74.93 k allocations: 3.551 MiB)
```

Code – Scala:

```scala
// Splitting a string into kmers
def kmer_decomposition(DNA_string: String, k: Int): MutableList[String] = {
  var kmers = MutableList[String]()
  val num_kmers = (DNA_string.length() - k)
  for (i <- 0 to num_kmers) {
    kmers += DNA_string.slice(i, i + k)
  }
  return kmers
}

// Function to sort kmers
def sorting(kmers: MutableList[String]): MutableList[String] = {
  var lex = kmers
  // for loop which loops through the kmers
  for (i <- 1 to lex.length - 1) {
    for (j <- (i - 1) to 0 by -1) {
      // checks if lex(j) is greater than lex(j+1)
      if (lex(j) > lex(j + 1)) {
        var temp = lex(j + 1)
        lex = lex.updated(j + 1, lex(j))
        lex = lex.updated(j, temp)
      }
    }
  }
  return lex
}

def BWT_split(DNA_string: String, k: Int): MutableList[String] = {
  // generates kmers from input string
  var kmers = kmer_decomposition(DNA_string, k)
  // sorts kmers
  var sorted_kmers = sorting(kmers)
  // performs BWT on each of the kmer
  var BWT_split_out = MutableList[String]()
  for (i <- 0 to sorted_kmers.length - 1) {
    BWT_split_out += BWT(sorted_kmers(i))
  }
  return BWT_split_out
}

var t1 = System.nanoTime
var BWT_split_out = BWT_split(str, k)

val duration1 = (System.nanoTime - t1) / 1e9d    //> duration1  : Double = 0.050381
print(BWT_split_out)                             //> MutableList(G$GA, G$GA, GGA$, GGA$, AGG$, AGG$)


def IBWT_split(BWT_out_split: MutableList[String]): MutableList[String] = {
  var IBWT_kmers = MutableList[String]()
  for (i <- 0 to BWT_out_split.length - 1) {
    IBWT_kmers += IBWT(BWT_out_split(i))
  }
  return IBWT_kmers
}

var t2 = System.nanoTime
var IBWT_out_split = IBWT_split(BWT_split_out)
                                                 //| t(AGG, AGG, GAG, GAG, GGA, GGA)
var duration2 = (System.nanoTime - t2) / 1e9d    //> duration2  : Double = 0.0023857
}
```

Run times of each language:

Computer Specifications of systems used to run the following codes:

JavaScript and Scala: Processor: Intel(R) Core(™) i7-10510U
CPU @ 1.80GHz 2.30GHz RAM: 16.0GB

Python: Processor: Intel(R) Core(TM) i7-8750H
CPU @ 2.20GHz 2.21 GHz RAM: 16.0GB

Julia: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz
RAM: 16.0GB

String Length: 500

Python:

K	Time taken for BWT (s)	Time taken for IBWT (s)
5	0.07854127883911133	0.008981466293334961
10	0.1056818962097168	0.01595330238342285
15	0.12172460556030273	0.023958921432495117
20	0.14061832427978516	0.040889739990234375
25	0.10578417778015137	0.05287909507751465

JavaScript:

K	Time taken for BWT (s)	Time taken for IBWT (s)
5	0.016947	0.04192
10	0.018083	0.11265
15	0.020553	0.309325
20	0.019817	0.718586
25	0.021327	1.266435

Scala:

K	Time taken for BWT (s)	Time taken for IBWT (s)
5	1.4173541	0.0336926
10	1.4411696	0.0584635
15	1.4433011	0.08408
20	1.4302153	0.0866745
25	1.6724986	0.1091746

Julia:

K	Time taken for BWT (s)	Time taken for IBWT (s)
5	0.138038	0.112778
10	0.125184	0.109100
15	0.131068	0.139528
20	0.140327	0.196783
25	0.175064	0.631486

String Length: 2000

Python:

K	Time taken for BWT (s)	Time taken for IBWT (s)
5	0.3420865535736084	0.02496957778930664
10	0.2981719970703125	0.05086636543273926
15	0.37403035163879395	0.10572409629821777
20	0.38300490379333496	0.18550515174865723
25	0.37602734565734863	0.279221773147583

JavaScript:

K	Time taken for BWT (s)	Time taken for IBWT (s)
5	0.140287	0.121380
10	0.152774	0.565675
15	0.186439	1.189235
20	0.192212	2.592653
25	0.190146	4.772273

Scala:

K	Time taken for BWT (s)	Time taken for IBWT (s)
5	98.2277459	0.1358753
10	74.7768807	0.0930074
15	75.9185703	0.1274475
20	74.7621887	0.1845562
25	94.1470351	0.7468491

Julia:

K	Time taken for BWT (s)	Time taken for IBWT (s)
5	0.193696	0.177949
10	0.246129	0.251371
15	0.237819	0.396306
20	0.278681	0.661648
25	0.331872	1.011497

String Length: 20000

Python:

K	Time taken for BWT (s)	Time taken for IBWT (s)
5	2.8863234519958496	0.17553305625915527
10	3.196842908859253	0.5006504058837891
15	3.436218023300171	0.893338680267334
20	3.259413480758667	1.512526035308838
25	3.776116132736206	2.1897740364074707

JavaScript:

K	Time taken for BWT (s)	Time taken for IBWT (s)
5	12.2921067	0.867638
10	14.293564	4.187898
15	18.293467	13.526163
20	18.450545	27.880089
25	18.146937	52.809712

Julia:

K	Time taken for BWT (s)	Time taken for IBWT (s)
5	1.101833	1.145914
10	1.230348	1.852064
15	1.567450	3.473270
20	2.120496	5.755633
25	2.442508	9.129068

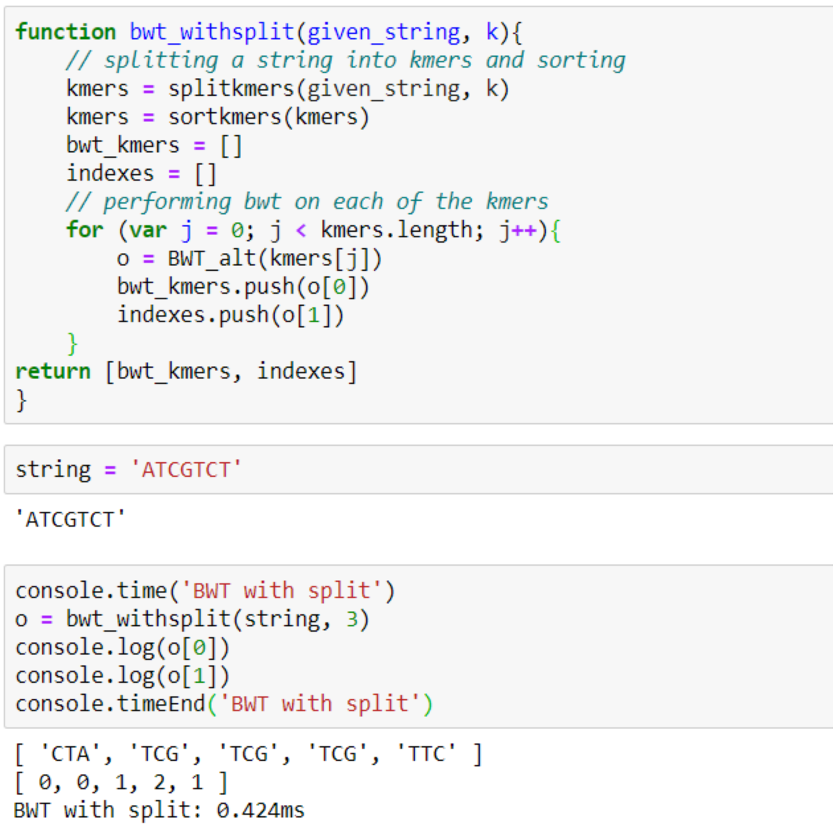
The analysis done based upon these run times has been included in the conclusion of this report.

Genome reconstruction:

In the field of Bioinformatics, the computational power and memory required to analyse genomes of massive lengths is expensive and inaccessible. To combat this, smaller fragments of the sequence are utilised. This is termed as reads or kmers.

In this project, the possibility of using Burrows-Wheeler Transform for genome reconstruction has been explored. A given string has first been split into kmers of a fixed length, and the output has been sorted. Burrows Wheeler Transform and Inverse Burrows-Wheeler Transform has been performed.

However, since the Burrows-Wheeler Transform is purely a data transformation technique which is used as a predecessor for data compression, none of the aspects of the Burrows-Wheeler Transform algorithm allows for genome reconstruction without the inclusion of other methods.



```
function bwt_withsplit(given_string, k){
    // splitting a string into kmers and sorting
    kmers = splitkmers(given_string, k)
    kmers = sortkmers(kmers)
    bwt_kmers = []
    indexes = []
    // performing bwt on each of the kmers
    for (var j = 0; j < kmers.length; j++){
        o = BWT_alt(kmers[j])
        bwt_kmers.push(o[0])
        indexes.push(o[1])
    }
return [bwt_kmers, indexes]
}
```

```
string = 'ATCGTCT'
```

```
'ATCGTCT'
```

```
console.time('BWT with split')
o = bwt_withsplit(string, 3)
console.log(o[0])
console.log(o[1])
console.timeEnd('BWT with split')
```

```
[ 'CTA', 'TCG', 'TCG', 'TCG', 'TTC' ]
[ 0, 0, 1, 2, 1 ]
BWT with split: 0.424ms
```

The output of the BWT shows that the kmers have lost their initial property which is the basis of the suffix-prefix based methodologies such as Eulerian and Hamiltonian.

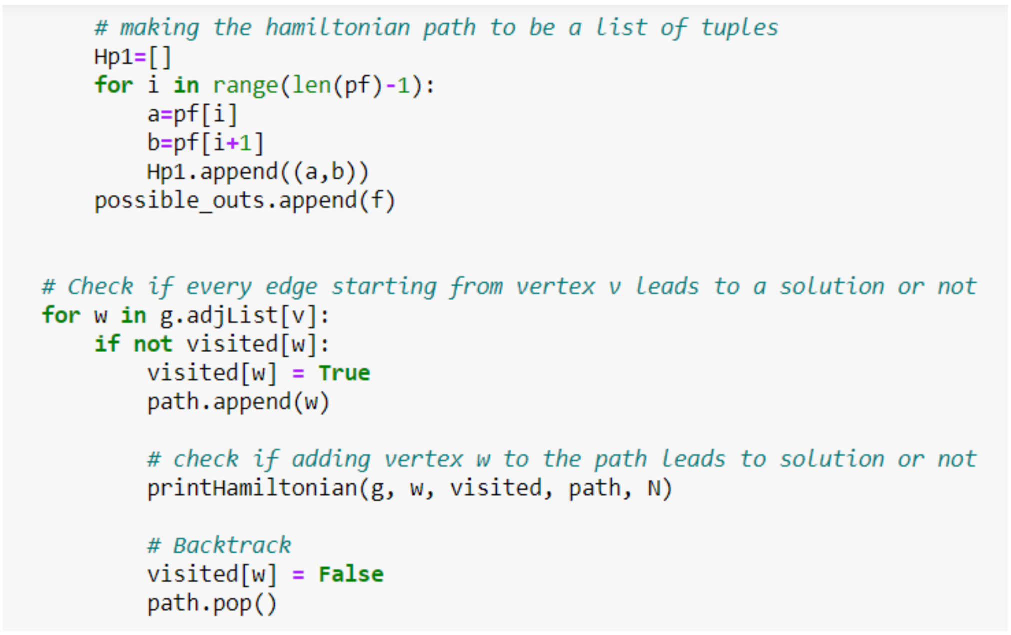
This is due to the fact that the BWT process transforms the data.

For example, the first kmer ‘ATC’ is not even present in the output of the BWT.

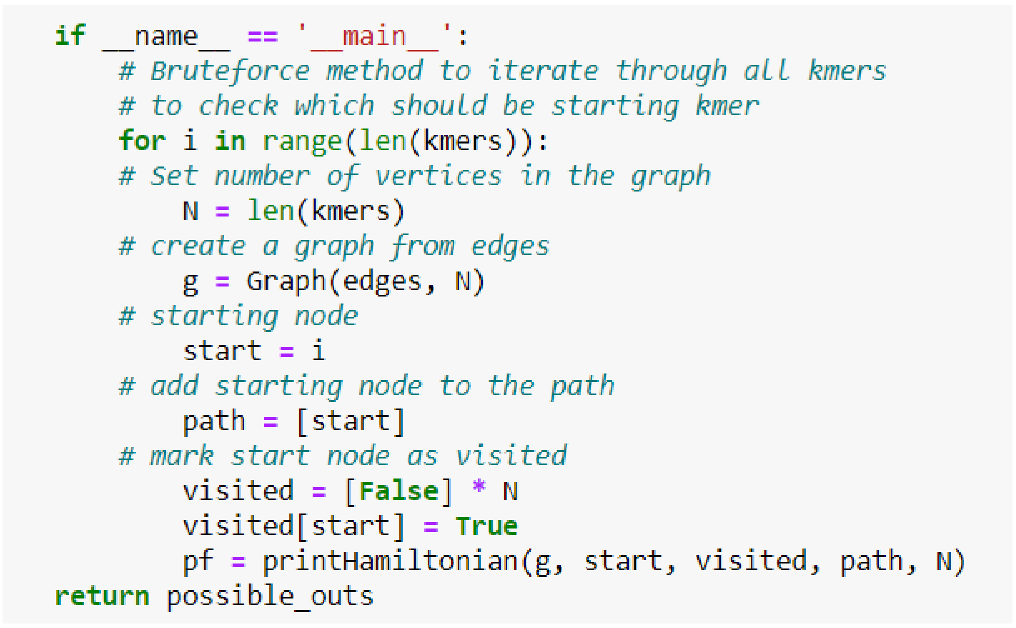
Genome reconstruction using Hamiltonian:

```python
def Hamiltonian(IBWT_out_split):
    possible_outs = []
    # initialising empty dictionary
    kmers = IBWT_out_split
    d1 = {}
    coun = 0
    # assigning numbers to kmers
    for i in kmers:
        d1[i] = coun
        coun+=1
    # reversing dictionary order
    inv_d1 = {v: k for k, v in d1.items()}

    # empty list for edges
    edges=[]
    # number used for suffix/prefix
    fix = 2
    # for elements in dictionary
    for i in d1:
        for j in d1:
            # checking if suffix = prefix
            if (i[-fix:] == j[0:fix]):
                # if yes, append the vertex numbers to edges
                edges.append((d1[i], d1[j]))
```

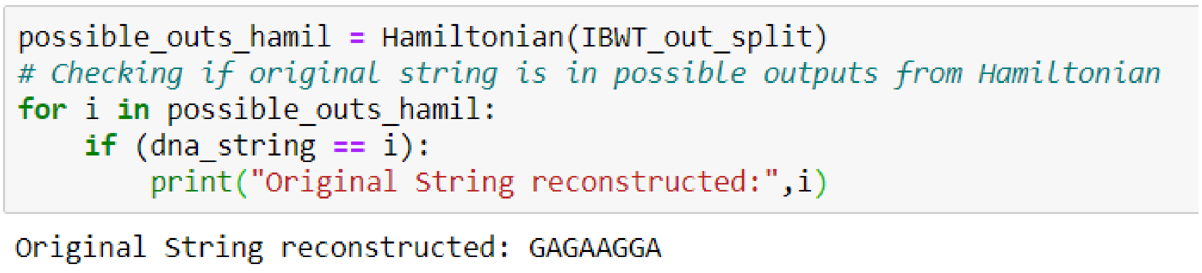
```python
# class to represent a graph object:
    class Graph:

        # Constructor
        def __init__(self, edges, N):

        # A List of Lists to represent an adjacency list
            self.adjList = [[] for _ in range(N)]
        # add edges to the undirected graph
            for (src, dest) in edges:
                self.adjList[src].append(dest)

    def printHamiltonian(g, v, visited, path, N):
        # empty array for hamiltonian path
        pf = []
        # if all the vertices are visited, then hamiltonian path exists
        if len(path) == N:
            # print hamiltonian path
            for i in path:
                # append the kmers
                pf.append(inv_d1[i])

            #taking first kmer in Hamiltonian path
            pf1 = pf[0]
            # empty string
            f = ""
            # appending prefix of first kmer
            f += pf1[0:fix]
            # appending last base of remaining kmers in path
            for j in pf:
                f += j[-1]
```



```
        # making the hamiltonian path to be a list of tuples
        Hp1=[]
        for i in range(len(pf)-1):
            a=pf[i]
            b=pf[i+1]
            Hp1.append((a,b))
        possible_outs.append(f)


    # Check if every edge starting from vertex v leads to a solution or not
    for w in g.adjList[v]:
        if not visited[w]:
            visited[w] = True
            path.append(w)

            # check if adding vertex w to the path leads to solution or not
            printHamiltonian(g, w, visited, path, N)

            # Backtrack
            visited[w] = False
            path.pop()
```



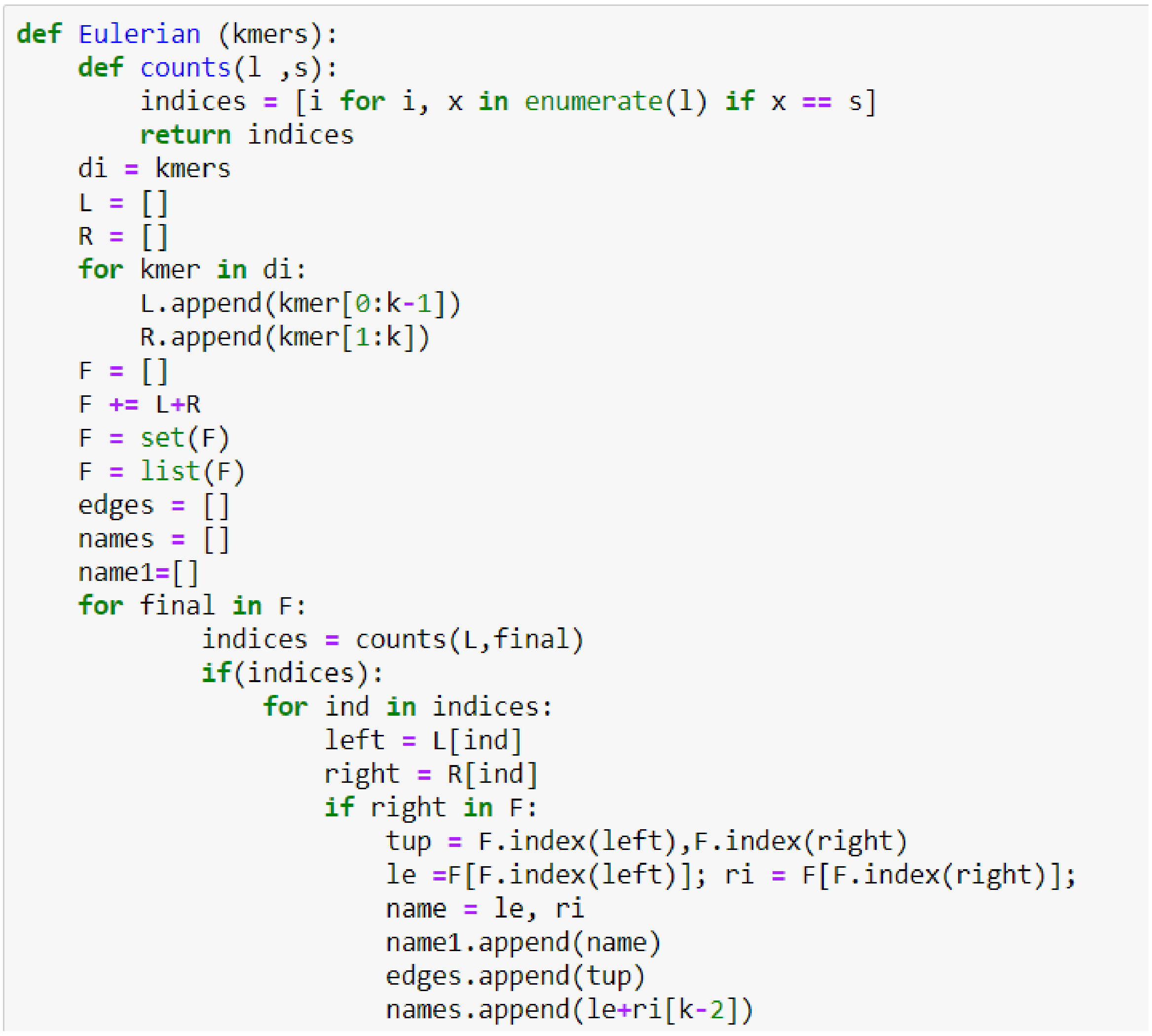
```
if __name__ == '__main__':
    # Bruteforce method to iterate through all kmers
    # to check which should be starting kmer
    for i in range(len(kmers)):
    # Set number of vertices in the graph
        N = len(kmers)
    # create a graph from edges
        g = Graph(edges, N)
    # starting node
        start = i
    # add starting node to the path
        path = [start]
    # mark start node as visited
        visited = [False] * N
        visited[start] = True
        pf = printHamiltonian(g, start, visited, path, N)
return possible_outs
```



```
possible_outs_hamil = Hamiltonian(IBWT_out_split)
# Checking if original string is in possible outputs from Hamiltonian
for i in possible_outs_hamil:
    if (dna_string == i):
        print("Original String reconstructed:",i)
```

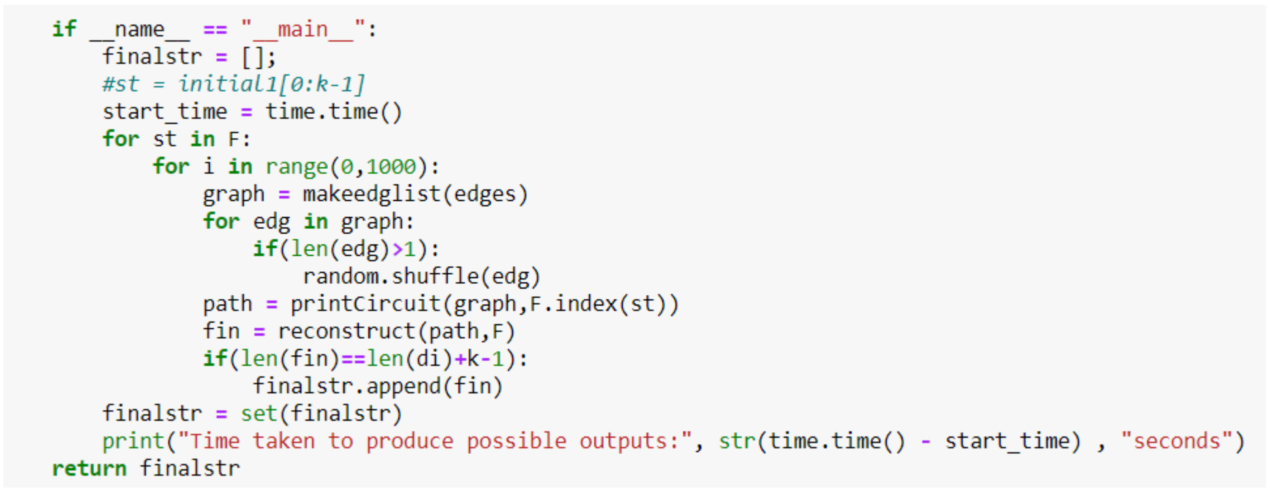
```
Original String reconstructed: GAGAAGGA
```

Genome Reconstruction using Eulerian:

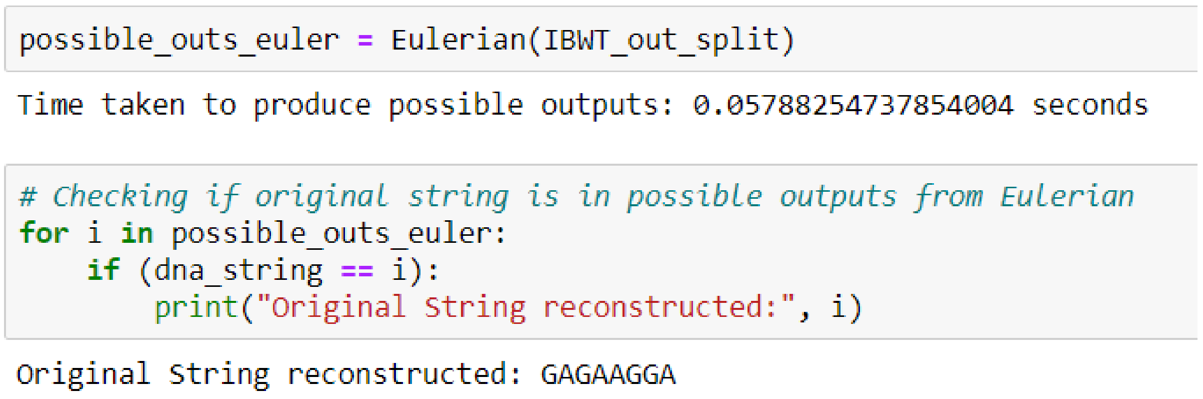


```python
def Eulerian (kmers):
    def counts(l ,s):
        indices = [i for i, x in enumerate(l) if x == s]
        return indices
    di = kmers
    L = []
    R = []
    for kmer in di:
        L.append(kmer[0:k-1])
        R.append(kmer[1:k])
    F = []
    F += L+R
    F = set(F)
    F = list(F)
    edges = []
    names = []
    name1=[]
    for final in F:
            indices = counts(L,final)
            if(indices):
                for ind in indices:
                    left = L[ind]
                    right = R[ind]
                    if right in F:
                        tup = F.index(left),F.index(right)
                        le =F[F.index(left)]; ri = F[F.index(right)];
                        name = le, ri
                        name1.append(name)
                        edges.append(tup)
                        names.append(le+ri[k-2])
    def reconstruct(x,a):
        fin = F[x[0]]
        for i in range(1,len(x)):
            fin+=F[x[i]][k-2]
        return fin
    def makeedglist(edg):
        graph = []
        for fin in F:
            li = []
            for ed in edg:
                if(ed[0]==F.index(fin)):
                    li.append(ed[1])
            graph.append(li)
        return graph
    def printCircuit(adj,st):
        edge_count = dict()
        for i in range(len(adj)):
            # find the count of edges to keep track
            # of unused edges
            edge_count[i] = len(adj[i])
        if len(adj) == 0:
            return # empty graph
        # Maintain a stack to keep vertices
        curr_path = []
        # vector to store final circuit
        circuit = []
        # start from any vertex
        curr_path.append(st)
        curr_v = st # Current vertex
        path=[]
```

```
    while len(curr_path):
        # If there's remaining edge
        if edge_count[curr_v]:
            # Push the vertex
            curr_path.append(curr_v)
            # Find the next vertex using an edge
            next_v = adj[curr_v][-1]
            # and remove that edge
            edge_count[curr_v] -= 1
            adj[curr_v].pop()
            # Move to next vertex
            curr_v = next_v
        # back-track to find remaining circuit
        else:
            circuit.append(curr_v)
            # Back-tracking
            curr_v = curr_path[-1]
            curr_path.pop()
    # we've got the circuit, now print it in reverse
    for i in range(len(circuit) - 1, -1, -1):
        path.append(circuit[i])
    return path
```



```
    if __name__ == "__main__":
        finalstr = [];
        #st = initial1[0:k-1]
        start_time = time.time()
        for st in F:
            for i in range(0,1000):
                graph = makeedglist(edges)
                for edg in graph:
                    if(len(edg)>1):
                        random.shuffle(edg)
                path = printCircuit(graph,F.index(st))
                fin = reconstruct(path,F)
                if(len(fin)==len(di)+k-1):
                    finalstr.append(fin)
        finalstr = set(finalstr)
        print("Time taken to produce possible outputs:", str(time.time() - start_time) , "seconds")
    return finalstr
```



```
possible_outs_euler = Eulerian(IBWT_out_split)
```

```
Time taken to produce possible outputs: 0.05788254737854004 seconds
```

```
# Checking if original string is in possible outputs from Eulerian
for i in possible_outs_euler:
    if (dna_string == i):
        print("Original String reconstructed:", i)
```

```
Original String reconstructed: GAGAAGGA
```

Conclusion:

Based upon the tables which show the time taken in each language, it is evident that Python is the most efficient, with the program taking the least amount of time. On the contrary, Scala is the least efficient, with the language taking the highest amount of time to produce the results. Furthermore, the Scala compiler crashed when provided with a string of 20000, a problem that was not faced with the other languages.

It is also conclusive that genome reconstruction cannot be done solely based upon Burrows-Wheeler Transform as this data transformation technique causes the suffix-prefix property of kmers to be diminished.

References:

- http://www.cs.jhu.edu/~langmea/resources/bwt_fm.pdf
- https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-16-S18-S5
- http://csbio.unc.edu/mcmillan/Comp555S16/Lecture22.pdf
- https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/bwt.pdf
- https://web.stanford.edu/class/cs262/archives/notes/lecture4.pdf

Work Split:

Team Member	Work Done
Abijith Pradeep (02)	• BWT and IBWT in Python • Pattern Counting in Python • Encoding and Decoding in Python
G Rohith (26)	• BWT and IBWT in Scala • Reconstructing string using Hamiltonian
Lakshaya Karthikeyan (39)	• BWT and IBWT in JavaScript • Patten Matching in JavaScript • Encoding and Decoding in JavaScript • Conducting genome reconstruction experiment using suffix-prefix analysis after BWT
Sanjay B (56)	• BWT and IBWT in Julia • Reconstructing string using Eulerian