

Intelligence in Biological Systems

End Semester 3 Project

Burrow Wheeler Transform and its Applications in Genome Assembly



Arjun Anil (CB.EN.U4AIE19012)
Arrun Sivasubramanian (CB.EN.U4AIE19013)
Sabarishwaran Ganesan (CB.EN.U4AIE19053)
Vasudevan K (CB.EN.U4AIE19067)

Index:

S no.	Contents	Page No.
1	List of Figures	2
2	Abstract & Introduction	3
3	Solving Methodology	3 - 4
4	Experimental Results: Python Code	5 - 7
5	First Last Property	7 – 10
6	Sort Table Technique	10 – 11
7	Java Code	11 – 15
8	Julia Code	15 – 19
9	Java Script Code	19 – 22
10	Using Burrow Wheeler transform for genome assembly	23 – 26
11	Other Applications of Burrow Wheeler Transform	26 – 27
12	Conclusion and Future Work	27
13	Acknowledgements	28
14	References	28

List of Figures:

S no.	Figure	Page No.
1	Burrow Wheeler Transformation	4
2	I/O for BWT	5
3	Compression and Expansion Methods	6
4	I/O for First Last Property	7 – 8
5	Counter and filt Methods	9
6	Output of Corrected First Last Code	10
7	Sort Table Technique and Code	10 – 11
8	Java Output Snippet	15
9	Julia Output Snippet	19
10	JavaScript Output Snippet	22
11	kMeriser method	23
12	Burrow Wheeler of k-mers with Hamming Distances	24
13	Output with appended symbols	25

Abstract:

Burrow wheeler Transform is a very efficient algorithm that can be used to compress long sequences which can be images, strings or even large DNA sequences. It has a worst case time complexity of $O(n)$ in terms of computation as well as memory. The compressed string usually has a length less than initial data, but may exceed in case of less repeats. In this work, we have used it for compressing as it's the primary purpose and we have also discussed its applications in genome assembly for reconstruction from reads and a possibility of it being used in Eulerian path problem.

Introduction:

BWT is the transformation which structures the data in such a way that helps in efficient compression of data. It simply rearranges similar characters together. Hence, it is used in many compression algorithms. In an effort to reduce the memory requirement for sequence alignment, several alignment programs were developed (Bowtie, BWA, and SOAP2) that use the Burrows–Wheeler transform.

Problem statement:

The primary objective of this assignment is: Given a DNA sequence or a word, we have to find the Burrow wheeler Transform for it and then use it for genome assembly and other such related problems.

Solving Methodology:

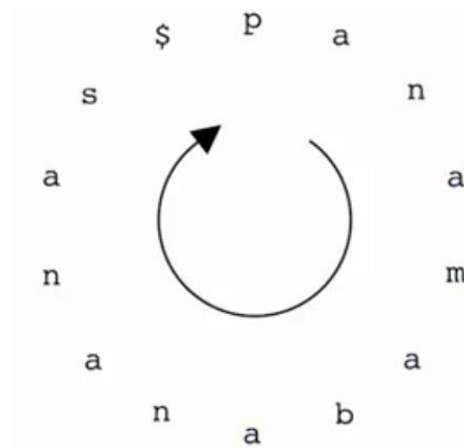
- To solve this, we need to append a special character at the end of the sequence.
- Following this, we need to perform a cyclic shift of characters in clockwise manner.
- Once this is done, the last character of the list of sequences will give us the BWT of the initial sequence.
- Once we obtain the BWT sequence, all the like characters counts are taken care by the counter. Once we get the count of each variable,

we can compress the sequence using the corresponding digit and a single alphabet in place of all characters.

- Once we compress it, it's ready for transmission using minimum characters. The end user can decode the sequence using two methods as discussed later.
- After performing BWT for individual stripped sequences, trend analysis for replacing the canonical prefix suffix method for genome reconstruction can be applied.

Example: Let's take "panamabananas"

```
panamabananas$
$panamabananas
s$panamabanan
as$panamabanan
nas$panamabana
anas$panamaban
nanas$panamaba
ananas$panamab
bananas$panama
abananas$panam
mabananas$pana
amabananas$pan
namabananas$pa
anamabananas$p
```



```
$panamabananas
abananas$panam
amabananas$pan
anamabananas$p
ananas$panamab
anas$panamaban
as$panamabanan
bananas$panama
mabananas$pana
namabananas$pa
nanas$panamaba
nas$panamabana
panamabananas$
s$panamabanan
```

Burrows-Wheeler Transform (BWT):
Last column = smnpbnnaaaaa\$a

Experimental results:

Since the nature of experiment in all the four languages (Python, Java, Julia and JS) remains the same, we have elaborately discussed working of our algorithm only in one language for reference (Python). A few variable names may vary, but the overall working of the code is the same.

I] Python Code:

Step 1: Taking input form user

```
import numpy as np

word = input("Enter String: ") + "$"

Enter String: panamabananas

l = len(word)
lis = []
for i in range(l):
    lis.append(word)
    word = word[-1] + word[:-1]
print(lis)

['panamabananas$', '$panamabananas', 's$panamabanan$', 'as$panamabanan', 'nas$panamabana', 'anas$panamaban', 'nanas$panamaba', 'ananas$panamab', 'bananas$panama', 'abananas$panam', 'mabananas$pana', 'amabananas$pan', 'namabananas$pa', 'anamabananas$p']

def bub_sort(seq):
    for i in range(0, len(seq) - 1):
        for j in range(0, len(seq) - 1):
            if (seq[j] > seq[j+1]):
                # Check whether 2 strings appear in Lexicographic Order
                bwt_seq = seq[j]
                # Swapping of adjacent kmers if they don't appear in Lexicographic order
                seq[j] = seq[j + 1]
                seq[j + 1] = bwt_seq
    return seq

def burrow_wheeler(seq):
    bwt_str = ""
    for i in range(len(seq)):
        bwt_str += seq[i][-1]
    return bwt_str
```

Output is as follows:

```
print(f"List in lexicographic order is : \n{bub_sort(lis)}")

List in lexicographic order is :
['$panamabananas', 'abananas$panam', 'amabananas$pan', 'anamabananas$p', 'ananas$panamab', 'anas$panamaban', 'as$panamabanan', 'bananas$panama', 'mabananas$pana', 'namabananas$pa', 'nanas$panamaba', 'nas$panamabana', 'panamabananas$', 's$panamabanan']

bwtseq = burrow_wheeler(lis)
print(f"Burrow Wheeler transform of {word} yields: {bwtseq}")

Burrow Wheeler transform of panamabananas$ yields: smnpbnnaaaaa$a
```

Step 2: Compression of burrow wheeler sequence:

Here we use a count variable. Whenever the adjacent characters are same we will increment count else we will merge count and the character and

we will reset count to 1. And if we reach the last character we will merge the count and the last character.

Code and Output:

```
def compress(seq):
    comp = ""
    count = 1
    for i in range(1, len(seq)):
        if seq[i] == seq[i-1]:
            count += 1
        else:
            comp += str(count) + seq[i-1]
            count = 1
    if i == len(seq)-1:
        comp += str(count) + seq[i]

    return comp

compressed = compress(bwtseq)
print(f"Compressed BWT of {bwtseq} yields: {compressed}")
print(f"Length of Compressed Sequence: {len(compressed)}")

Compressed BWT of smnpbnaaaaa$a yields: 1s1m1n1p1b2n5a1$1a
Length of Compressed Sequence: 18
```

Step 3: Expansion of Burrow wheeler transform:

For expansion we have count variable as well as start_index and end_index. While the value is a digit we will increment the position till it becomes a character. So that we get the count.

Which is the value from start_index to end_index. Then we will put for loop up to the count and print the character count times.

Code and Output:

```
def expand(comp_str):
    exp = ""
    count = 0
    start_ind = 0
    end_ind = 0

    for i in range(len(comp_str)):
        start_ind = i
        while comp_str[i].isdigit():
            i += 1
        if comp_str[i].isdigit() == False:
            end_ind = i
            count = int(comp_str[start_ind:end_ind])
            for j in range(count):
                exp += comp_str[i]

    return exp

expanded_str = expand(compressed)
print(f"Expanded Version of {compressed} gives: {expanded_str}")
print(f"Length of expanded sequence is : {len(expanded_str)}")

Expanded Version of 1s1m1n1p1b2n5a1$1a gives: smnpbnaaaaa$a
Length of expanded sequence is : 14
```

Step 4: Inverse Burrow wheeler transform:

To retrieve the actual sequence from BWT, we need to perform IBWT or inverse burrow wheeler transform. It can be done in several ways. The most efficient (having least time complexity) is using First last property. While the most accurate is using sorted table.

linsearch() function is used to linearly search for an element in an array ensuring that that index is not repeated in a previous iteration. We do this because of possible repeats present in a sequence in inverse Burrow wheeler transform.

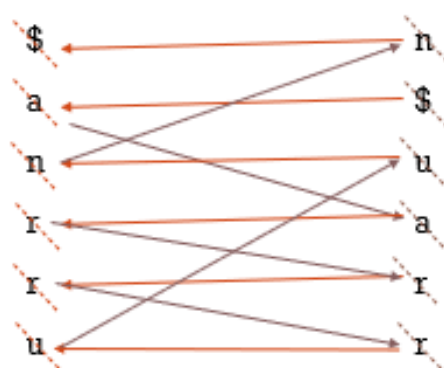
Inv_burrow_wheeler() function first checks in which index the \$ sign comes in the burrow wheeler sequence and after that merges the corresponding character in the sorted sequence to the reconstructed sequence. Then checks that character comes in which index in burrow wheeler sequence and the iteration goes up to the length of sequence. And finally returns the reconstructed sequence.

4.1. First last property:

“ This property basically states that value at index of sorted sequence is the present character and index at which sorted character occurs in BWT sequence is the index for next iteration”.

Example:

```
bwtseq = burrow_wheeler(lis)
print(f"Burrow Wheeler transform of {word} yields: {bwtseq}")
Burrow Wheeler transform of arrun$ yields: n$uarr
```



Final Text :
arrun\$

Code to implement and output is as follows:

```
def linsearch(seq,char,ind_arr):
    ind = -1 #in case not found
    for i in range(len(seq)):
        if(seq[i]==char and i not in ind_arr):
            ind = i
            break
    return ind

def inv_burrow_wheeler(bwt_seq):
    print(f"Burrow wheeler : {bwt_seq}")
    ind = linsearch(bwt_seq,"$",[0])
    print(f"Index = {ind}")
    chars = sorted(bwt_seq)
    print(f"Chars : {chars}\n")
    n = len(bwt_seq[:])
    recons_seq = ""
    traversed_indexes = np.zeros((n,1))
    for i in range(n-1):
        recons_seq += chars[ind]
        #print(f"Reconstructed sequence: {recons_seq}")
        ind = linsearch(bwt_seq,chars[ind],traversed_indexes)
        #print(f"Index = {ind}")
        traversed_indexes[i] = ind
    return recons_seq

print(f"Inverse Burrow Wheeler transform of {bwtseq} yields: {inv_burrow_wheeler(bwtseq)}$")

Burrow wheeler : n$uarr
Index = 1
Chars : ['$','a','n','r','r','u']

Inverse Burrow Wheeler transform of n$uarr yields: arrun$
```

Why does this algorithm fail in a few cases? An execution to prove this taking “arrunarjun” as an example:

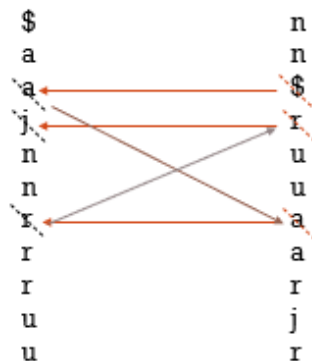
```
bwtseq = burrow_wheeler(lis)
print(f"Burrow Wheeler transform of {word} yields: {bwtseq}")

Burrow Wheeler transform of arrunarjun$ yields: nn$ruuaarjr
```

```
print(f"Inverse Burrow Wheeler transform of {bwtseq} yields: {inv_burrow_wheeler(bwtseq)}$")

Burrow wheeler : nn$ruuaarjr
Index = 2
Chars : ['$','a','a','j','n','n','r','r','r','u','u']

Inverse Burrow Wheeler transform of nn$ruuaarjr yields: arjunarrun$
```



Thus we land at arjunarrun\$ instead of arrunarjun\$

Thus we realise that lexicographic arrangement of *j* before *r* leads to a blatant error. Though such an error will not happen in DNA sequences due to occurrence of only 4 characters, we must ensure that it is rectified for all practical cases during execution. So we'll have to consider indices of each character too.

The cyclic property of BWT ensures that the index of a character in the BWT sequence is also followed in the lexicographic ordering of the same sequence.

To achieve it, we'll make a few changes to existing with these two methods:

```
def counter(arr):
    ind = []
    finStr = []
    for i in range(len(arr)):
        ch = arr[i+1]
        val = ch.count(arr[i])
        finStr.append(arr[i] + str(val))

    return finStr

def filt(lis):
    filt_str = ""
    for i in range(len(lis)):
        for j in range(len(lis[i])):
            if(lis[i][j].isalpha()):
                filt_str += lis[i][j]
            else:
                continue

    return filt_str
```

The counter() function counts all characters in the string and appends it to a list with the corresponding count as if all repeated characters are unique.

The filter function is used to filter out the characters from repeated counter list and make it into a single string without indices.

When we apply the former method to the burrow wheeler sequence and the latter to the sorted BWT string, the sorting can become manifold faster and accurate.

Snip of that and its corresponding output is as follows:

```
def inv_burrow_wheeler(bwt_seq):
    print(f"Burrow wheeler : {bwt_seq}")
    unq = counter(bwt_seq)
    print(f"Unique characters:{unq}")
    ind = linsearch(unq,"$1")
    chars = sorted(unq)
    print(f"Chars : {chars}\n")
    n = len(unq[:])
    recons_seq = ""
    for i in range(n-1):
        recons_seq += chars[ind]
        #print(f"Reconstructed sequence: {recons_seq}")
        ind = linsearch(unq,chars[ind])
        #print(f"Index = {ind}")

    return filt(recons_seq)

print(f"Inverse Burrow Wheeler transform of {bwtseq} yields: {inv_burrow_wheeler(bwtseq)}$")

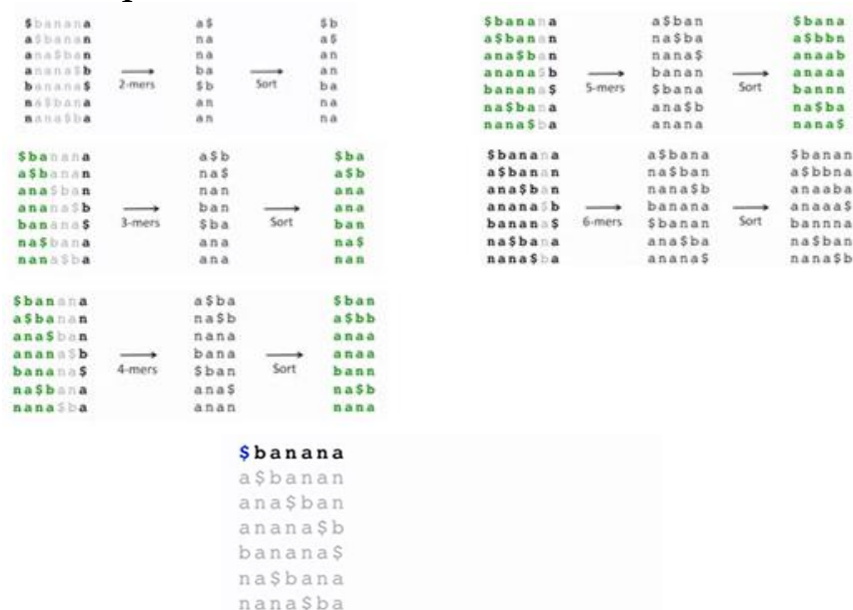
Burrow wheeler : smnpbnnaaaaa$a
Unique characters:['s1', 'm1', 'n1', 'p1', 'b1', 'n2', 'n3', 'a1', 'a2', 'a3', 'a4', 'a5', '$1', 'a6']
Chars : ['$1', 'a1', 'a2', 'a3', 'a4', 'a5', 'a6', 'b1', 'm1', 'n1', 'n2', 'n3', 'p1', 's1']

Inverse Burrow Wheeler transform of smnpbnnaaaaa$a yields: panamabanas$
```

The following is the canonical sort table technique

4.2: Sort Table Technique:

We need to resort to repeated lexicographic sorting and appending the sorted sequence as our final matrix. Once we do this for the total length of the sequence, the first element of that list will give us the reconstructed sequence.



We now know the entire matrix!

Code to implement it and output is as follows:

```
print(f"Inverse Burrow Wheeler transform of {bwtseq} yields: {inv_burrow_wheeler(bwtseq)}$")

Burrow wheeler : smnpbnnaaaaa$a
Index = 12
Chars : ['$ ', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'b', 'm', 'n', 'n', 'n', 'p', 's']

Inverse Burrow Wheeler transform of smnpbnnaaaaa$a yields: pabamananas$

def ibwt2(seq):
    matrix = [''] * len(seq)
    for _ in seq:
        matrix = sorted(i + j for i, j in zip(seq, matrix))
    return matrix[0]

print(f"Inverse Burrow Wheeler transform of {bwtseq} yields: {ibwt2(bwtseq)}")

Inverse Burrow Wheeler transform of smnpbnnaaaaa$a yields: $panamabananas
```

The zip function returns a zip object that concatenates/zips all i^{th} iteration parameters together and pairs them together. And the for each iteration we just need to sort lexicographically after appending the additional characters determined.

II] Code in Java:

```
import java.util.*;
public class BurrowWheeler
{
    public String[] bubbleSortList(String str[])
    {
        String t[] = str;
        String temp;
        for (int j = 0; j < str.length; j++)
        {
            for (int i = j + 1; i < str.length; i++)
            {
                // comparing adjacent strings
                if (t[i].compareTo(t[j]) < 0)
                {
                    temp = t[j];
                    t[j] = t[i];
                    t[i] = temp;
                }
            }
        }
        return t;
    }
}
```

```

public String compress(String str)
{
    String comp = "";
    int count = 1;
    for(int i=1;i<str.length();i++)
    {
        if(str.charAt(i)==str.charAt(i-1))
        {
            count += 1;
        }
        else
        {
            comp += Integer.toString(count)+(str.charAt(i-1));
            count = 1;
        }

        if(i==str.length()-1)
        {
            comp += Integer.toString(count)+(str.charAt(i));
        }
    }
    System.out.println("Length of compressed Sequence: "+comp.length());
    return comp;
}

public String expand(String com)
{
    String exp = "";
    int count=0;

    /*
    for(int i=0;i<com.length();i++)
    {
        if(Character.isDigit(com.charAt(i)))
        {
            count = (int)com.charAt(i)-48; //ASCII
            for(int j=0;j<count;j++)
            {
                exp += com.charAt(i+1);
            }
        }
        else
        {
            count = 0;
        }
    }
    */

    /*
    int pos = 0;

```

```

for(int i=0;i<com.length();i++)
{
    while(Character.isDigit(com.charAt(i)))
    {
        count += ((int)com.charAt(i) - 48)* Math.pow(10,pos); //ASCII
        pos += 1;

        i++;
        if(Character.isDigit(com.charAt(i))== false)
        {
            for(int j=0;j<count;j++)
            {
                exp += com.charAt(i);
            }
        }
        count = 0;
        pos = 0;
    }
    */

    int start_ind = 0;
    int end_ind = 0;
    for(int i=0;i<com.length();i++)
    {
        start_ind = i;
        while(Character.isDigit(com.charAt(i)))
        {
            i++;
            if(Character.isDigit(com.charAt(i))== false)
            {
                end_ind = i;
                count = Integer.parseInt(com.substring(start_ind,end_ind));
                for(int j=0;j<count;j++)
                {
                    exp += com.charAt(i);
                }
                start_ind = end_ind;
            }
        }
        count = 0;
    }
    return exp;
}

public String BWT(String str)
{
    String word = str;
    int l = word.length();
    String lis[] = new String[l];
    for(int i=0;i<l;i++)

```

```

    {
        lis[i] = word;
        word = word.charAt(l-1) + word.substring(0,l-1);
    }
    System.out.println("\nList of words:");
    for(int i=0;i<l;i++)
    {
        System.out.print(lis[i]+"\\t");
    }
    String bw = "";
    String sortedSeq[] = bubbleSortList(lis);
    System.out.println("\n\\nList in sorted order:");
    for(int i=0;i<l;i++)
    {
        System.out.print(sortedSeq[i]+"\\t");
    }
    for(int i=0;i<l;i++)
    {
        bw += sortedSeq[i].charAt(l-1);
    }
    return bw;
}

public String ibwt(String bwt)
{
    int len = bwt.length();
    List<String> table = new ArrayList<>();
    for (int i = 0; i < len; ++i) {
        table.add("");
    }
    for (int j = 0; j < len; ++j)
    {
        for (int i = 0; i < len; ++i)
        {
            table.set(i, bwt.charAt(i) + table.get(i));
        }
        table.sort(String::compareTo);
    }
    return table.get(0);
}

public static void main(String args[])
{
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter String: ");
    String seq = sc.next()+"$";
    System.out.println("String input: "+seq);
    BurrowWheeler bw = new BurrowWheeler();
    String bwtSeq = bw.BWT(seq);
    System.out.println("\\nLength of sequence :"+bwtSeq.length());
}

```

```

        System.out.println("\nBurrow Wheeler Transform of "+seq+ " yields:"
+ bwtSeq);
        String comp = bw.compress(bwtSeq);
        System.out.println("Compressed version of BWT is:" +comp);
        String expand = bw.expand(comp);
        System.out.println("Expanded version of BWT is:" +bw.expand(comp));
        System.out.println("\nInverse Burrow Wheeler Transform of "+bwtSeq+ "
yields:" +bw.ibwt(expand));
        System.out.println("\nLength of IBWT sequence :"+bw.ibwt(expand).length());
    }
}

```

Output in Java:

```
Blue: Terminal Window - Welcome
Options
Enter String: panamabananas
String input: panamabananas$

List of words:
panamabananas$ $panamabananas s$panamabananana as$panamabananan nas$panamababana anas$panamababan nanas$panamababa ananas$panamabab ananas$panamabab bananas$panamabab abananas$panamabab mabananas$panamabab amabananas$panamabab

List in sorted order:
$panamabananas abananas$panamabab amabananas$panamabab anamabananas$panamabab ananas$panamabab anas$panamababan as$panamababanan bananas$panamabab mabananas$panamabab namabananas$panamabab nanas$panamababa nas$panamababana
Length of sequence :14

Burrow Wheeler Transform of panamabananas$ yields:smnpbnnaaaaa$
Length of compressed Sequence: 18
Compressed version of BWT is:1s1m1n1p1b2n5a1$1a
Expanded version of BWT is:smnpbnnaaaaa$

Inverse Burrow Wheeler Transform of smnpbnnaaaaa$ yields:$panamabananas

Length of IBWT sequence :14
```

III] Code in Julia:

```

print("Enter the word : ")
word = readline()           # Requesting the Sequence
println("Sequence : $word")

function BWT(word)          # Function for finding BWT of a Sequence
    word = string(word,$')  # Adding Dollar Sign to track last word
    l = length(word)
    T = []
    for i in 1:l
        push!(T,word)
        word = string(word[end],word[1:end - 1]) # Cyclic Rotations of the Sequence
    end
    return T
end

function Bub_sort(list)     # Function for Sorting
    for i in 1:length(list)
        for j in 1:length(list)
            if list[i] < list[j]
                temp = list[i]

```

```

        list[i] = list[j]
        list[j] = temp
    end
end
end
return list
end

function short_form(BWT_sequence)
    BWT_list = []

    for i in BWT_sequence
        push!(BWT_list,i)
    end

    current = BWT_list[1]
    count = 0
    counts = []
    chr = []

    for i in 1:length(BWT_list)
        if current == BWT_list[i]
            if i == length(BWT_list)
                count += 1
                push!(counts,count)
                push!(chr,current)
            else
                count += 1
            end
        else
            if i == length(BWT_list)
                push!(counts,count)
                push!(chr,current)
                count = 1
                current = BWT_list[i]
                push!(counts,count)
                push!(chr,current)
            else
                push!(counts,count)
                push!(chr,current)
                count = 1
            end
        end
        current = BWT_list[i]
    end

    final = ""

    for i in 1:length(counts)
        final = string(final,string(counts[i]) * string(chr[i]))
    end
end

```



```

    return final
end

function Expansion(short)
    expr = ""
    for i in 1:length(short)
        start = i
        while isnumeric(short[i])
            i += 1
            if isnumeric(short[i]) == false
                e = i
                counts = parse(Int,short[start:e-1])
                for j in 1:counts
                    expr = expr * short[i]
                end
            end
        end
    end
    return expr
end

List = BWT(word)
Lexicographical_order = Bub_sort(List)

BWT_sequence = ""
for i in 1:length(Lexicographical_order)
    global BWT_sequence *= Lexicographical_order[i][end] # Concatenation of last
    char of the Sequences
end

println("Burrow Wheeler Transform of $word is : $BWT_sequence ")

BWT_short = short_form(BWT_sequence)

println("Short form of BWT sequence : $BWT_short ")

BWT_expand = Expansion(BWT_short)

println("Expansion of short form of BWT sequence : $BWT_expand")

#-----
# Inverse Burrow Wheeler Transform using Sort table Method
#-----

function IBWT(BWT_sequence) # Function to Find IBWT of a BWT Sequence
    BWT = []

    for i in 1:length(BWT_sequence)
        push!(BWT,BWT_sequence[i])
    end
end

```

```

l = length(BWT)

IBWT = Array{String}(undef,l) # Creating an Empty Array

temp = BWT

for i in 1:l
    temp_1 = copy(temp)
    temp_1 = Bub_sort(temp_1)
    for j in 1:l
        IBWT[j] = BWT[j] * temp_1[j] # Sorting and Concatenating
    end
    temp = IBWT
end

for i in 1:l
    IBWT[i] = IBWT[i][2:end]
end

return IBWT[1][2:end] # returning the First String of the list
end

IBWT_sequence = IBWT(BWT_expand)

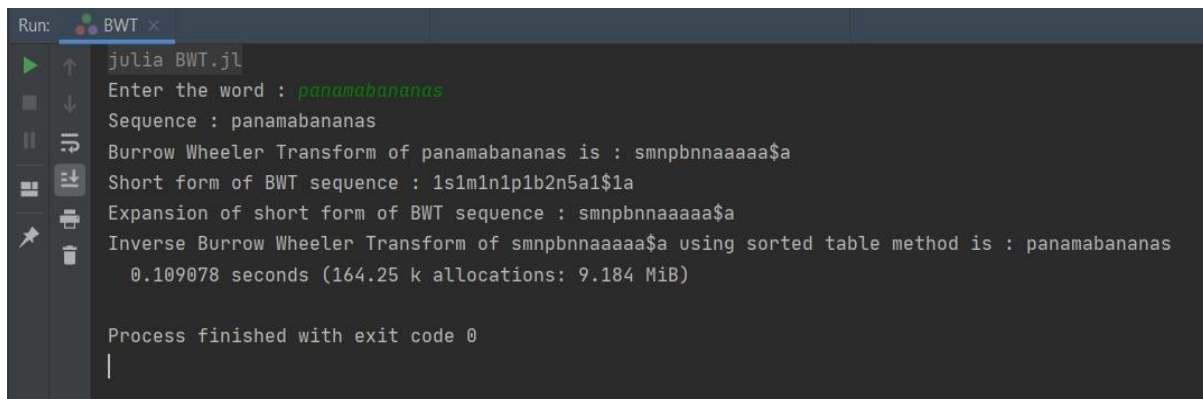
println("Inverse Burrow Wheeler Transform of $BWT_sequence is :
$IBWT_sequence ")

function ibwt2(seq)
    matrix=[]
    for i in 1:length(seq)
        push!(matrix,"")
    end
    temp=[]
    for i in 1:length(seq)
        push!(temp,"")
    end
    for i in 1:length(seq)
        for j in 1:length(seq)
            temp[j]=seq[j] * matrix[j]
        end
        matrix=sort(temp)
    end
    return matrix[1]
end

IBWT_sequence2 = ibwt2(BWT_expand)
IBWT_sequence2=IBWT_sequence2[2:end]
println("Inverse Burrow Wheeler Transform (using sort table method) of
$BWT_expand is : $IBWT_sequence2\$ ")

```

Output in Julia code:



```
Run: BWT x
julia BWT.jl
Enter the word : panamabanananas
Sequence : panamabanananas
Burrow Wheeler Transform of panamabanananas is : smnpbnnaaaaa$a
Short form of BWT sequence : 1s1m1n1p1b2n5a1$1a
Expansion of short form of BWT sequence : smnpbnnaaaaa$a
Inverse Burrow Wheeler Transform of smnpbnnaaaaa$a using sorted table method is : panamabanananas
0.109078 seconds (164.25 k allocations: 9.184 MiB)

Process finished with exit code 0
|
```

IV] Code in Javascript:

```
//Burrow Wheeler
```

```
// first part
```

```
var word = window.prompt("Enter the string: ")+"$";
```

```
l = word.length;
```

```
var lis = [];
```

```
var i;
```

```
for(i=0;i<l;i++){
```

```
    lis.push(word);
```

```
    word = word[l-1] + word.slice(0,l-1);
```

```
    }
```

```
    console.log(lis);
```

```
//bubble sort
```

```
function bubble_sort(seq){
```

```
    var bwt_seq;
```

```
    for(var i=0;i<seq.length-1;i++){
```

```
        for(var j=0;j<seq.length-1;j++){
```

```
            if(seq[j] > seq[j+1]){
```

```
                bwt_seq = seq[j]
```

```
                seq[j] = seq[j+1]
```

```
                seq[j+1] = bwt_seq
```

```
            }
```

```
        }
```

```
    }
```

```
    return seq
```

```
}
```

```
console.log("List in lexicographic order is : "+bubble_sort(lis))
```

```

// burrow wheeler function

function burrow_wheeler(seq){
    var bwt_str = ""
    for(i=0;i<seq.length;i++){
        bwt_str+=(seq[i][seq.length-1])
    }
    return bwt_str
}

bwtseq = burrow_wheeler(lis)
console.log("Burrow Wheeler transform of "+word.slice(0,l-1)+" yields: "+bwtseq)

// compression

function compress(seq){
    comp = ""
    count = 1
    for(var i=1;i<seq.length;i++){
        if(seq[i]==seq[i-1])
            count += 1
        else{
            comp += String(count) + seq[i-1]
            count = 1
        }
        if(i==seq.length-1)
            comp += String(count) + seq[i]
    }

    return comp
}

compressed = compress(bwtseq)
console.log("Compressed BWT of "+bwtseq+" yields: "+compressed)
console.log("Length of Compressed Sequence: "+(compressed.length))

//expansion

function isdigit(myString) {
    return /\d/.test(myString);
}

function expand(comp_str){
    exp = ""
    count = 0
    start_ind = 0
    end_ind = 0

    for(var i=0;i<comp_str.length;i++){
        start_ind = i
        while(isdigit(comp_str[i])==true){

```

```

        i += 1
        if(isdigit(comp_str[i])===false){
            end_ind = i
            count = comp_str.slice(start_ind,end_ind)
            //print(f"Count {count}")
            for(var j=0;j<count;j++){
                exp += comp_str[i]
            }
        }
    }
}
return exp
}

```

```

expanded_str = expand(compressed)
console.log("Expanded Version of "+compressed+" gives: "+expanded_str)
console.log("Length of expanded sequence is : "+expanded_str.length)

```

// Inverse Burrow Wheeler

```

function linsearch(seq,char,ind_arr){
    ind = -1 //incase not found
    for(i=0;i<seq.length;i++){
        if(seq[i] == char && ind_arr.includes(i)===false){
            ind = i
            break;
        }
    }
    return ind
}

```

```

function zeros(n) {
    var list = [];

    for (var i = 0; i < n; i++) {
        list.push(0)
    }

    return list;
}

var sortAlphabets = function(text) {
    return text.split("").sort();
};

function inv_burrow_wheeler(bwt_seq){
    console.log("Burrow wheeler : "+bwt_seq)
    ind = linsearch(bwt_seq,"$",[])
    console.log("Index = "+ind)
    chars = sortAlphabets(bwt_seq)
    console.log("Chars : "+chars)
    n = bwt_seq.length
    recons_seq = ""

```

```

traversed_indexes = zeros(n)
for(i=0;i<=n;i++){
    recons_seq += chars[ind]
    traversed_indexes[i] = ind
    ind = linsearch(bwt_seq,chars[ind],traversed_indexes)
}
return recons_seq
}
console.log("Inverse Burrow Wheeler transform of "+bwtseq+" yields:"+
inv_burrow_wheeler(bwtseq)+"$")

//IBWT sort table method

function ibwt2(seq){
    len=seq.length

    matrix=[]
    for(i=0;i<len;++i){
        matrix.push("")
    }
    temp=[]
    for(i=0;i<len;++i){
        temp.push("")
    }
    for(j=0;j<len;++j){
        for(i=0;i<len;++i){
            temp[i]=seq[i]+matrix[i]
        }
        matrix=temp.sort()
    }
    return matrix[0]
}
console.log("IBWT(sort table method) of "+bwtseq+" yields:"+
ibwt2(bwtseq).slice(1,bwtseq.length)+"$")

```

Output in JavaScript:

```

List in lexicographic order is : $panamabananas,abananas$panam,amaban
anas$pan,anamabananas$p,ananas$panamab,anas$panamaban,as$panamabanan,
bananas$panama,mabananas$pana,namabananas$pa,nanas$panamaba,nas$panam
abana,panamabananas$,s$panamabananana
Burrow Wheeler transform of panamabananas yields: smnpbnnaaaaa$a
Compressed BWT of smnpbnnaaaaa$a yields: 1s1m1n1p1b2n5a1$1a
Length of Compressed Sequence: 18
Expanded Version of 1s1m1n1p1b2n5a1$1a gives: smnpbnnaaaaa$a
Length of expanded sequence is : 14
Burrow wheeler : smnpbnnaaaaa$a
Index = 12
Chars : $,a,a,a,a,a,a,b,m,n,n,n,p,s
Inverse Burrow Wheeler transform of smnpbnnaaaaa$a yields:pabamananan
as$
IBWT(sort table method) of smnpbnnaaaaa$a yields:panamabananas$

```

Using Burrow wheeler transform for genome assembly:

Whatever we have seen so far is simply applying Burrow Wheeler for compressing the longer genomic sequences. But the real aim of the project was to conclude that this transformation can be used for genome assembly and replace the prefix suffix approach or Eulerian path methods.

Many experiments were carried out on a genome sequence and trend analysis was done based on occurrences of BWT sequences in clusters. But none yielded a suitable result because the transformed sequence has no information about the ordering of the original sequence.

The experiments performed include stripping the DNA sequence to several k-mers and performing BWT on each of them. After that computing hamming distance was done for BWT with the adjacent BWT as well as with the original k-mer.

The main problem arises when we are told to find the starting k-mer from the list of k-mers after performing BWT. There is no trend that will help us identify it as each sequence has a lexicographic order. If we take for granted that we know starting kmer as it may be the first read that we got from sequencer, there are several strings that may have the same BWT. We will have to encounter a huge error in the process.

Another surprising thing is that many different k-mers occurring together in the string will have the same BWT. With that we may be able to over the sequences after performing IBWT of these sequences. But in this too, the order in which we have to consider overlap and the value of k that we need to consider will vary. Despite knowing all these we will be confused with the order in which the clusters have to be taken after performing IBWT.

Let's take an example:

```
def k_meriser(seq,k):  
    stripped = [(seq[i:i + k]) for i in range(0, len(seq)- k + 1)]  
    return stripped
```

k_meriser(): This function helps us to strip the DNA sequence into k mers and store all reads in a list

```
seq = "gcgctggctagcctaatagactcgcggcctcgtccaattggactagcagacctaagctccggtggtaaagtaataattcgattatttcgcaacacacctgc"
n_mer_len = 5
three_mers = k_meriser(seq,n_mer_len)
print(f"Unsorted List of K-mers is: \n{three_mers}")
print(f"Length of sequence: {len(seq)}")
print(f"number of k-mers generated: {len(three_mers)}")

Unsorted List of K-mers is:
['gcgct', 'cgctg', 'gctgg', 'ctggt', 'tggtc', 'ggcca', 'gtcag', 'tcagc', 'cagct', 'agcta', 'gctaa', 'ctaata', 'taata', 'aatag', 'ataga', 'tagac', 'agact', 'gactc', 'actcg', 'ctcgc', 'tcgcg', 'cgcgg', 'gcggc', 'cggcc', 'ggcct', 'gcctc', 'cctcg', 'ctcgc', 'tcgct', 'cgctc', 'gtccc', 'ctcca', 'tccaa', 'ccaat', 'caatt', 'aattg', 'attgg', 'ttgga', 'tggac', 'ggact', 'gacta', 'actag', 'ctagc', 'tagca', 'agcag', 'gcaga', 'cagac', 'agacc', 'gacct', 'acct', 'cctag', 'ctagc', 'tagct', 'agctc', 'gtccc', 'ctccg', 'tccgg', 'cgggt', 'cggtg', 'gggtg', 'gtggt', 'tggtg', 'ggtaa', 'gtaaa', 'taaac', 'aaacg', 'aacgt', 'acgtg', 'cgtaa', 'gtaat', 'taata', 'aatat', 'atatt', 'tattc', 'attcg', 'ttcga', 'tcgat', 'cgatt', 'gattt', 'attta', 'tttat', 'ttatt', 'tattc', 'attcg', 'ttcgc', 'tcgca', 'cgcaa', 'gcaac', 'caaca', 'aacac', 'acaca', 'cacac', 'acacc', 'cacct', 'acctg', 'cctgc']
Length of sequence: 100
number of k-mers generated: 96
```

```
print("k-mer \t bwt \t hamming")
for i in range(len(three_mers)):
    word = three_mers[i]
    l = len(word)
    lis = []
    for j in range(l):
        word = word[-1] + word[:-1]
        lis.append(word)

    bwt_kmer = burrow_wheeler(bub_sort(lis))
    hamming = np.zeros(len(three_mers))
    for k in range(l):
        hamming[i] += abs(ord(word[k]) - ord(bwt_kmer[k]))
    print(word, "\t", bwt_kmer, "\t ", hamming[i])
```

The first few outputs for this is given in the next page

k-mer	bwt	hamming
gcgct	ggccc	34.0
cgctg	ggccc	42.0
gctgg	gggct	34.0
ctggt	ttggc	34.0
tggtc	ttggc	26.0
ggcca	ctagg	46.0
gtcag	ctagg	12.0
tcagc	ctgac	46.0
cagct	ctgac	38.0
agcta	tagac	50.0
gctaa	tagac	30.0
ctaata	tatca	76.0
taata	ttaaa	38.0
aatag	gtaaa	50.0
ataga	gtaaa	12.0
tagac	gtaac	38.0
agact	gtaac	38.0
gactc	gtacc	38.0
actcg	gtacc	46.0
ctcgc	gtccc	8.0
tcgcg	tgccg	8.0
cgcgg	gggcc	16.0
gcggc	gggcc	8.0
cggcc	gccgc	16.0
ggcct	gcgtc	42.0
gcctc	gtccc	34.0
cctcg	gtccc	42.0
ctcgc	gtccc	8.0
tcgct	tgctc	42.0
cgctc	tcgcc	42.0
gctcc	tcgcc	26.0

k-mer	bwt	hamming
gcgct\$	tgg\$cc	160.0
cgctg\$	g\$gtcc	142.0
gctgg\$	ggg\$tc	160.0
ctggt\$	t\$tggc	186.0
tggtc\$	cttgg\$	60.0
ggtca\$	act\$gg	146.0
gtcag\$	gcta\$g	168.0
tcagc\$	ccgta\$	38.0
cagct\$	tc\$gac	172.0
agcta\$	at\$gac	152.0
gctaa\$	aatg\$c	138.0
ctaata\$	tta\$ac	160.0
taata\$	attaa\$	76.0
aatag\$	g\$taaa	134.0
ataga\$	agt\$aa	160.0
tagac\$	cgtaa\$	38.0
agact\$	tg\$aac	164.0
gactc\$	cgta\$c	172.0
actcg\$	g\$tacc	138.0
ctcgc\$	cgt\$cc	160.0
tcgcg\$	ggtcc\$	34.0
cgccg\$	g\$ggcc	142.0
gcggc\$	cggg\$c	134.0

With “\$” Appended at end

Inference from above table:

What we notice is that BWT of the original sequence has no co-relation with the actual k-mer. Because of this, identifying the starting k-mer is almost impossible.

As previously mentioned, the k-mers that occur together have the same burrow wheeler transform sequence. Yet we will not be able to identify the order in which these k-mers have to be arranged. To identify that an effort was taken by identifying the hamming distance of the k-mer with the sorted sequence. Initial trends show that they are in descending order, but later the order varies and is also ascending in a few cases.

Moreover, the adjacent k-mers have nothing in common with the previous transformation sequence with varying characters and absolute value of different relative hamming distances. We are also encouraged to start with a k-mer having minimum hamming distance, but even that approach fails.

Other Applications of Burrow Wheeler Transform:

a) Image Compression: The aim of image compression is to compress the image into a format which saves the storage space and provides an efficient format for transmission via telecommunication channels. The Burrows-Wheeler transform is based on block sorting, which rearranges data into an easier format for compressing.

Before utilizing the Burrows-Wheeler transform, the image need to be pre-process by using a discrete cosine transform, a discrete wavelet transform or predictive coding. Then the image is converted from a 2-dimensional to a 1-dimensional pixel sequence with different scanning methods. The forward Burrows-Wheeler transform is applied on block of the image data.

While compressing the image into the smallest storage space, the move-to-front and run-length encoding can be used to improve the compression ratio before entropy encoding. This is used to study both lossless and lossy image compression.

b) Approximate Pattern Matching with BWT: The structure contained in the Burrows-Wheeler transform can also be used to aid in approximate pattern matching—that is, finding substrings of the text that are similar to the pattern, but may not match exactly.

One method, ‘k-approximate’ matching, involves computing the edit distance between two strings as the number of basic operations (insertions, deletions and substitutions) needed to transform one string into the other. k-approximate matching involves finding all the substrings of the text for which the edit distance from the pattern is less than some value of k.

Approximate pattern matching is beyond the scope of this project, but we been involved in some preliminary work in this area.

3) Indexing Iris Images using BWT: In most biometric identification systems, the input biometric data has to be compared against that of every identity in the database in order to determine the identity of the input.

A major problem with this approach is the impact on response time which can increase significantly with the size of the database. In certain applications such as real time monitoring, this delay may not be acceptable. BWT is a method for indexing iris images for rapid identity retrieval. Every entry in the database is assigned an index code based on which a small subset is retrieved and matched in response to a query.

The basis of this approach is the sorted context property of the Burrows Wheeler Transform, a popular transformation used in data compression.

Conclusion:

With the above experiments, we would like to conclude that Burrows Wheeler transform is indeed an efficient algorithm to compress and transmit the sequence and the first last property is also very efficient in decoding the sequence correctly. But the use of the transformation algorithm is ineffective for problems relate to genome assembly though it can only be used for searching exact patterns within a sequence using the same first last property.

Future Work:

As ardent student, we would be grateful enough to review this topic again which would pave the way for us to improvise the same. We would encourage all other who take up this work in the future to try out other methods of comparing sequences such as cosine distance or try some other techniques based on modular arithmetic. Efforts based on absolute values are highly discouraged due to multiple occurrences of same transformed sequence without special character.

We also felt that doing the experiment with the special characters will be detrimental as the integer value of the character will play a crucial role. Though a few additional properties of BWT like: Everything before the dollar is arranged lexicographically, it is insignificant information about the original sequence.

Acknowledgements:

We would profusely like to thank Dr. K.P. Soman, the HOD of Dept, of CSE-AI at Amrita School of Engineering Coimbatore to compile this piece of work. We would deeply acknowledge Dr. Sachin Kumar, Faculty of Electronics at Department of CSE_AI, Amrita School of Engineering Coimbatore for helping us understand the requisite concepts. His invaluable contribution in making us acquainted with several methods to perform genome assembly cannot be ignored.

Reference:

- 1] Analysis of genomic rearrangements by using the Burrows-Wheeler transform of short-read data: Kouichi Kimura*, Asako Koike Tokyo, Japan. 9-11 September 2015.
- 2] Lecture 1. Exact Matching - Michael Schatz
<http://schatzlab.cshl.edu/teaching/2013/QB.IFX1.Exact%20Matching.pdf>
- 3] <https://www.coursera.org/lecture/algorithms-on-strings/using-bwt-for-pattern-matching-n8hIN>
- 4] <http://blog.avadis-ngs.com/2012/04/elegant-exact-string-match-using-bwt-2/>
- 5] https://corpus.canterbury.ac.nz/research/powell2001_hons_report.pdf