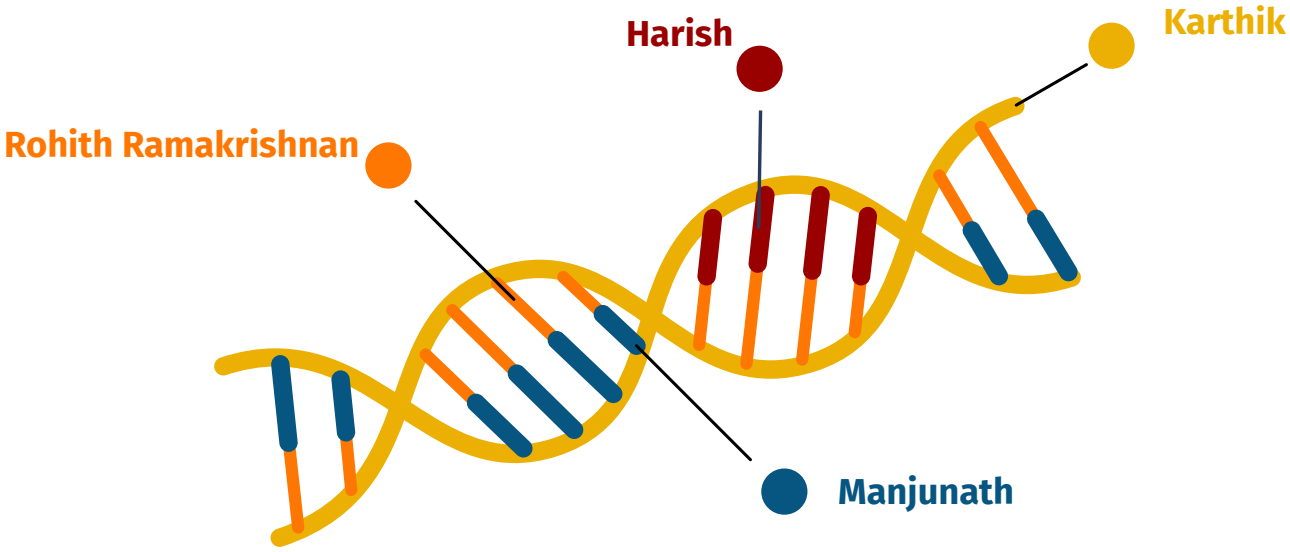
Genome

Reconstruction via Burrow Wheelers Transform

Team Members



Genetic Code of this Presentation

Compression

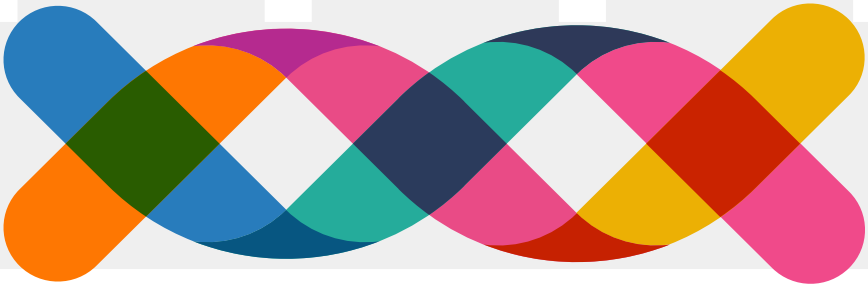
Applying BWT for Lossless Compression

Pattern Matching

Finding an occurance of a pattern in a transformed data

Genome Assembly

Analysis on whether BWT can be used for Reconstruction



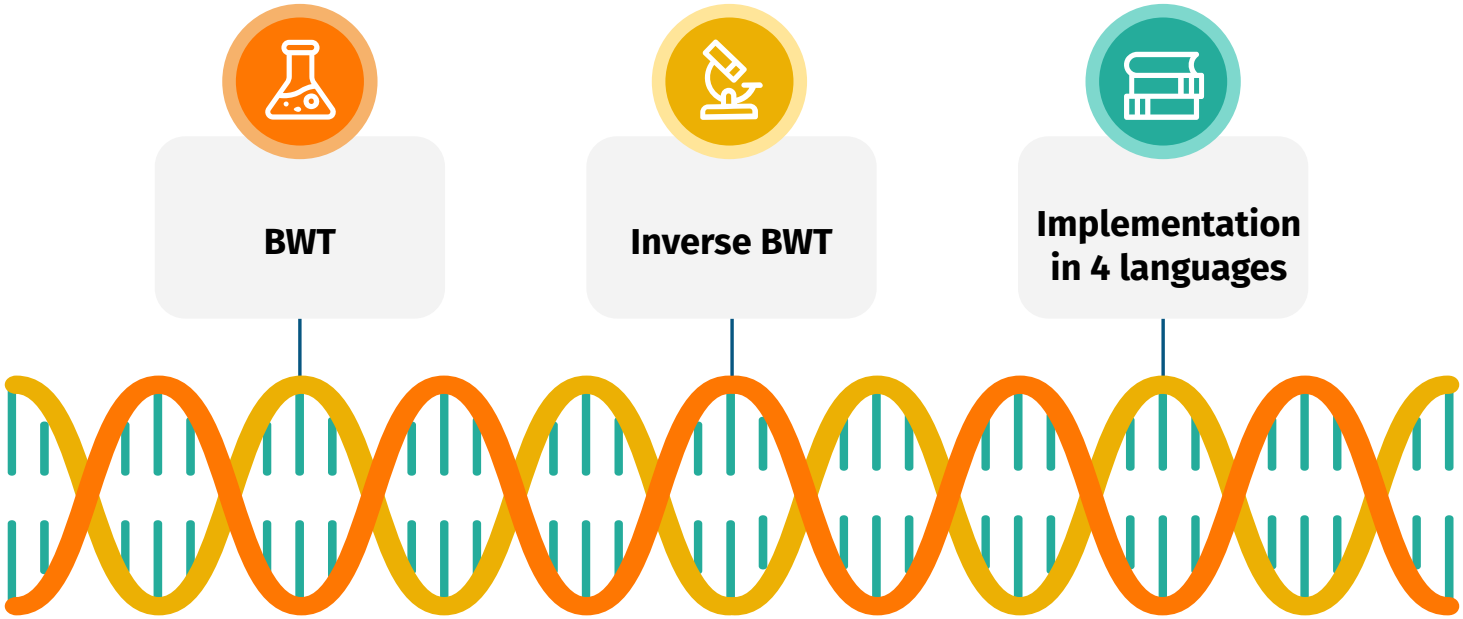
BWT & iBWT

A brief introduction on the algorithms

Result

Conclusion and further discussion

Burrows Wheeler Transform

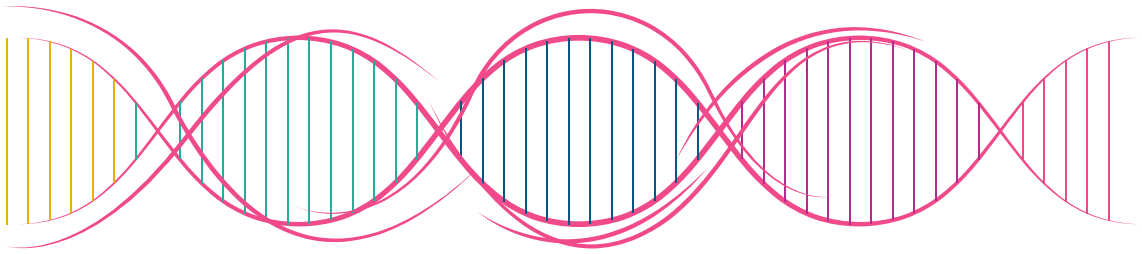


Burrows Wheeler Transform

- The Burrows–Wheeler transform (BWT) rearranges a character string into runs of similar characters.

- The transformation is *reversible*, without needing to store any additional data except the position of the first original character.

- The BWT is an effective method of improving the efficiency of text compression algorithms, costing only some extra computation.



BWT

STEP-1

Append a dollar to start/end of the string being transformed.

STEP-2

Create a table of cyclic rotations

STEP-3

Sort the BWT table in lexicographical order

STEP-4

BWT of the string → last column of the BWT table

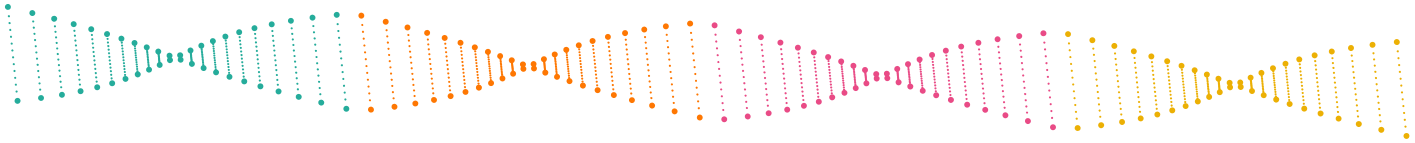
BWT

Transformation				
Input	All Rotations	Sorting All Rows into Lex Order	Taking Last Column	Output Last Column
^BANANA|	^BANANA| |^BANANA A|^BANAN NA|^BANA ANA|^BAN NANA|^BA ANANA|^B BANANA|^	**A**NANA|^B **AN**A|^BAN **A**|^BANAN **B**ANANA|^ **N**ANA|^BA **N**A|^BANA ^BANANA| |^BANANA	ANANA|^**B** ANA|^BA**N** A|^BANA**N** BANANA|^ NANA|^B**A** NA|^BAN**A** ^BANANA| |^BANAN**A**	BNN^AA|A

TRANSFORMATION

Input String-^BANANA|

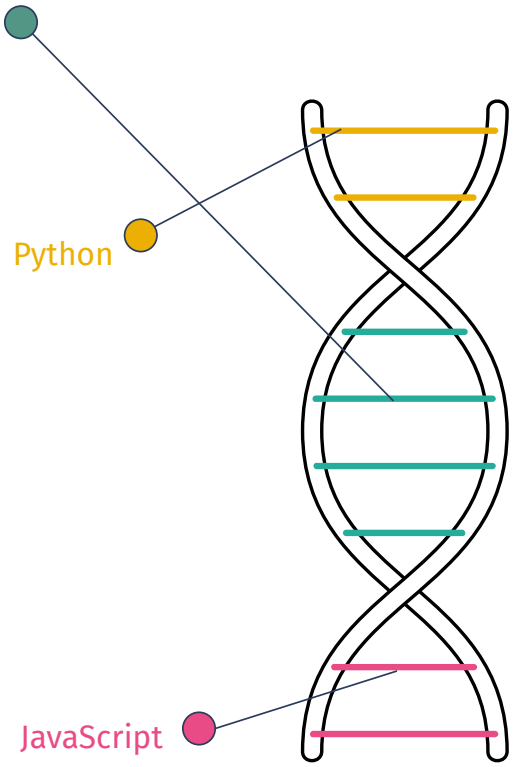
All rotations	Sorting all rows into lex order	Taking last column	Output last column
^BANANA|	ANANA|^B	ANANA|^B	
|^BANANA	ANA|^BAN	ANA|^BAN	
A|^BANAN	A|^BANAN	A|^BANAN	
NA|^BANA	BANANA|^	BANANA|^	
ANA|^BAN	NANA|^BA	NANA|^BA	
NANA|^BA	NA|^BANA	NA|^BANA	
ANANA|^B	^BANANA|	^BANANA|	
BANANA|^	|^BANANA	|^BANANA	BNN^AA|A



BWT - Implementation

```python
def BWT(sequence):
    sequence += '$'
    table = [sequence[index:] + sequence[:index] for index, _ in enumerate(sequence)]
    table.sort()
    bwt = [rotation[-1] for rotation in table]
    bwt = ''.join(bwt)
    return bwt


def i_BWT(sequence):
    table = [col for col in sequence]
    for i in range(len(sequence) - 1):
        table.sort()
        table = [sequence[i] + table[i] for i in range(len(sequence))]
    return table[[row[-1] for row in table].index('$')][:-1]
```

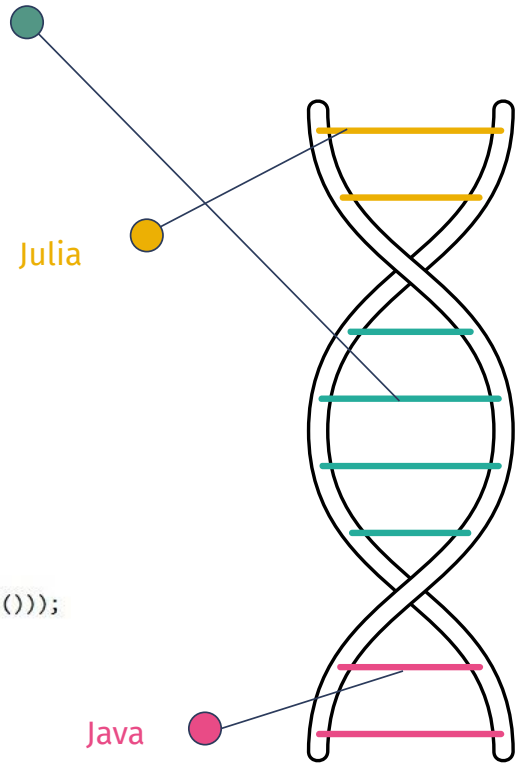
```javascript
function bwt(S){
    S = "$".concat(S)
    const a=[];
    var S_m = S;

    for(var i=0;i<S.length;i++){
        S_m = S_m.substring(S_m.length-1) + S_m.substring(0,S_m.length-1);
        a.push(S_m);
    }

    a.sort();

    var ss="";
    for(var x=0;x<a.length;x++){
        ss=ss.concat(a[x].substring(a[x].length-1));
    }

    return ss
}
```



BWT - Implementation

```
#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
```

```
public static String bwt(List<String> A){
    List<String> bwt = sorted(A);
for(int i = 0;i<A1.size();i++) {
    S = S + (A1.get(i).substring(A1.get(i).length()-1, A1.get(i).length()));

}
System.out.println("The bwt sequence is");
System.out.println(S);
```

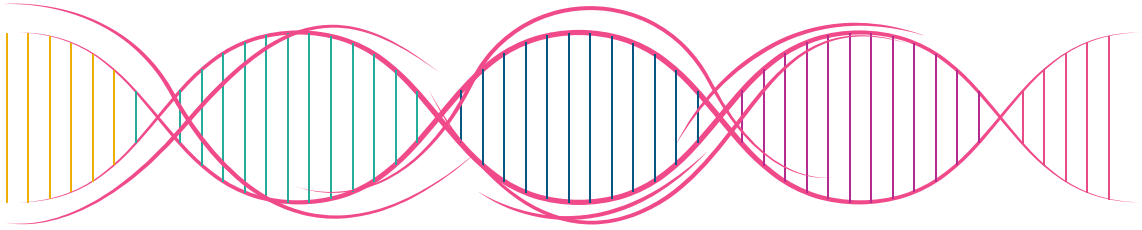


Inverse Burrows Wheeler Transform

The process of inverting the BW Transformed data to the original sequence without loss.

Methods:
1) Naive
2) LF approach (Last - First)

While the naive method is easier to implement, the LF method is more efficient in terms of speed and efficiency.



Naive Method

Step-1

1st column of the BWT table → BWT string .

Prepend the sorted BWT string to the table.

Step-2

Sort the table containing both the columns

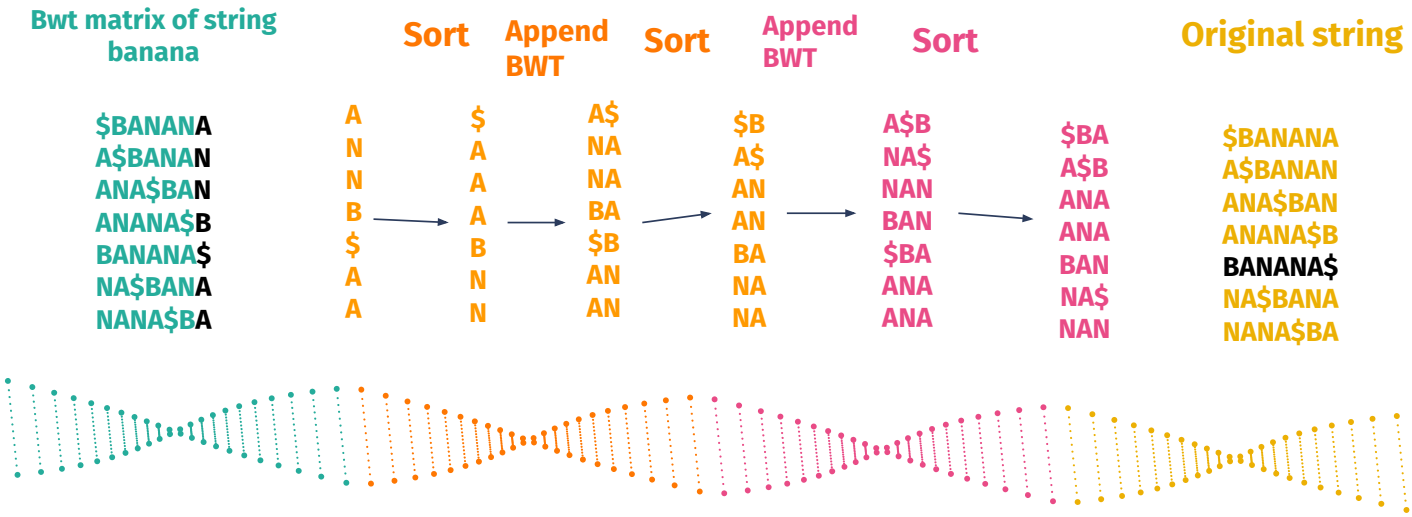
Prepend the BWT string to the table and sort again.

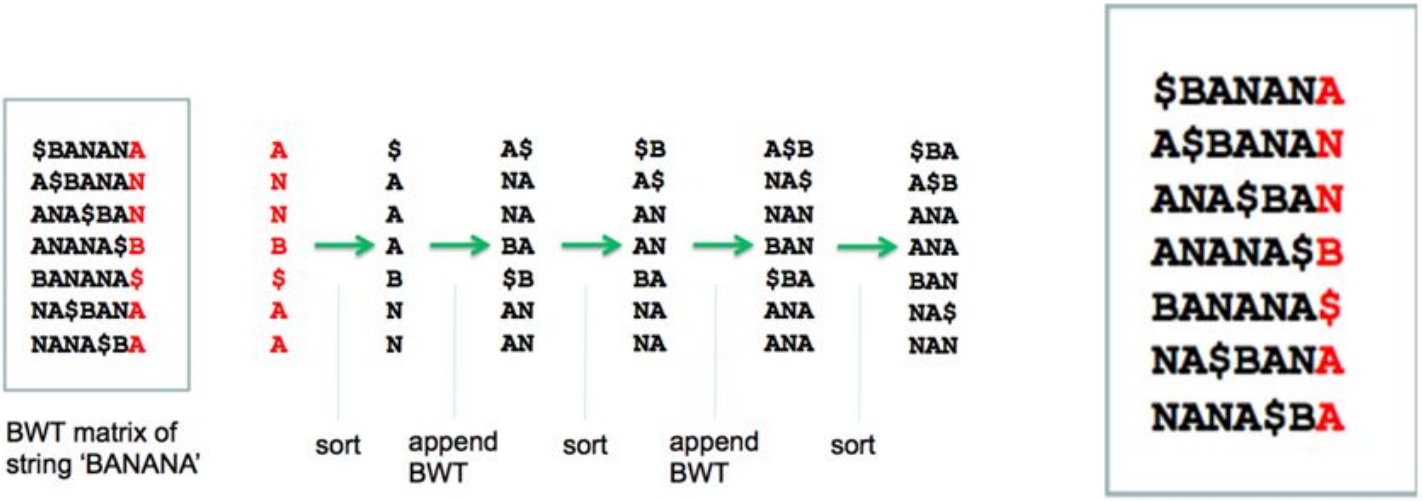
Step-3

Repeat the process for n times where n = length of the string

Row which starts/ends with the ‘$’ (marker) → original string

EXAMPLE

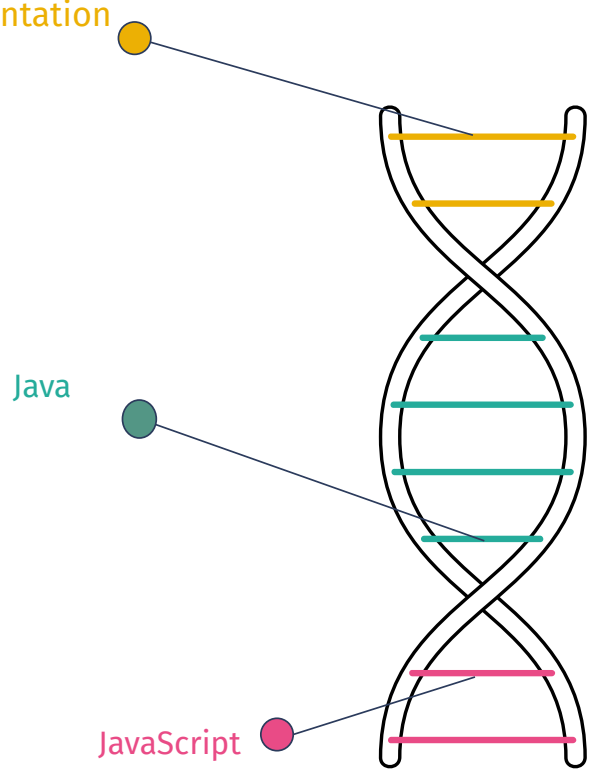

Bwt matrix of string banana
$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA
Sort
A
N
N
B
$
A
A
Append BWT
$
A
A
A
B
N
N
Sort
A$
NA
NA
BA
$B
AN
AN
Append BWT
$B
A$
AN
AN
BA
NA
NA
Sort
A$B
NA$
NAN
BAN
$BA
ANA
ANA
$BA
A$B
ANA
ANA
BAN
NA$
NAN
Original string
$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA


$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA
A N N B $ A A
$ A A A B N N
A$ NA NA BA $B AN AN
$B A$ AN AN BA NA NA
A$B NA$ NAN BAN $BA ANA ANA
$BA A$B ANA ANA BAN NA$ NAN
sort
append BWT
sort
append BWT
sort
BWT matrix of string 'BANANA'
$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA

Inverse BWT - Implementation

```java
public static void ibwt(String a){
    String ibwt = decode(c);
    a = S;
     List<String> C = new ArrayList<String>();
    for(int i = 0;i<a.length();i++) {
        C.add(a.substring(i, i+1));
        temp.add(a.substring(i, i+1));

    }
    Collections.sort(temp);
    for(int i = 0;i<C.size()-1;i++) {
      for(int j = 0;j<C.size();j++) {
        temp.set(j,C.get(j)+temp.get(j));
        }
    Collections.sort(temp);
    }
    System.out.println("The inverted(sorted) list from the bwt sequence is");
    System.out.println(temp);

}
```

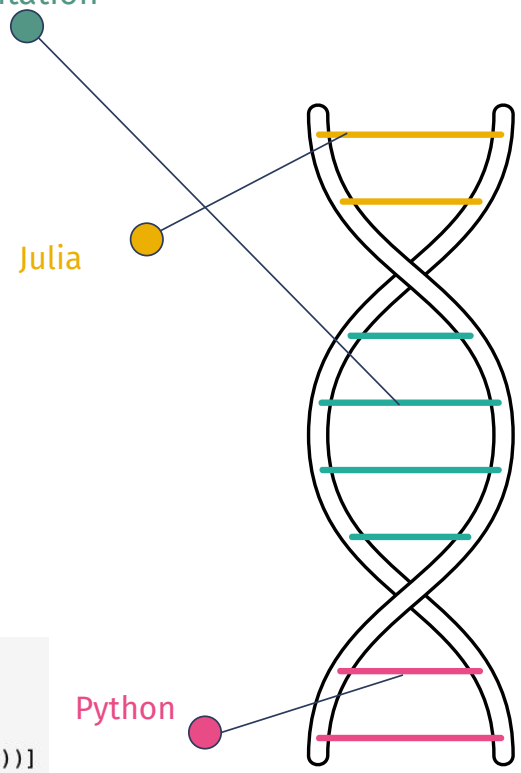
```javascript
function iBWT(ss){
    var a = ss.split("");
    ss = ss.split("");
    ss.sort();
    for(var i=0;i<a.length-1;i++){
        for(var x=0;x<a.length;x++){
            ss[x] = a[x] + ss[x];
        }
        ss.sort()
    }
    return ss[0].substring(1,ss[0].length)
}
```



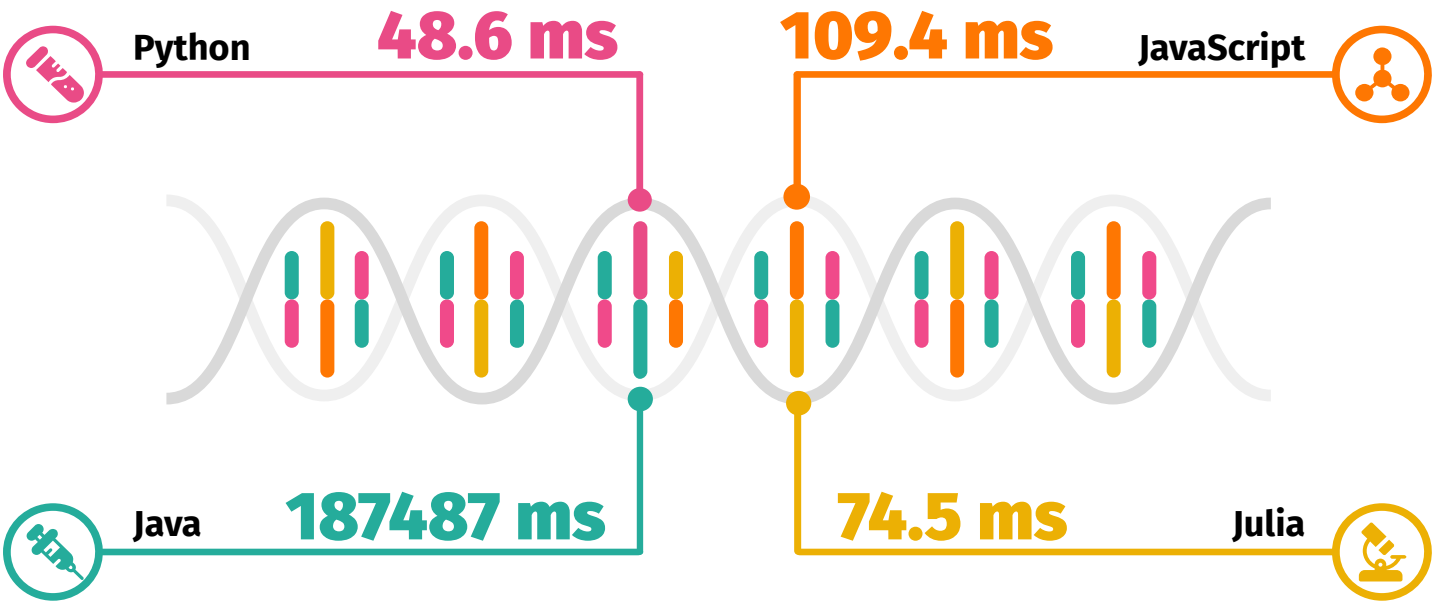
Inverse BWT - Implementation

```
# 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
```

```
def i_BWT(sequence):
    table = [col for col in sequence]
    for i in range(len(sequence) - 1):
        table.sort()
        table = [sequence[i] + table[i] for i in range(len(sequence))]
    return table[[row[-1] for row in table].index('$')][:-1]
```

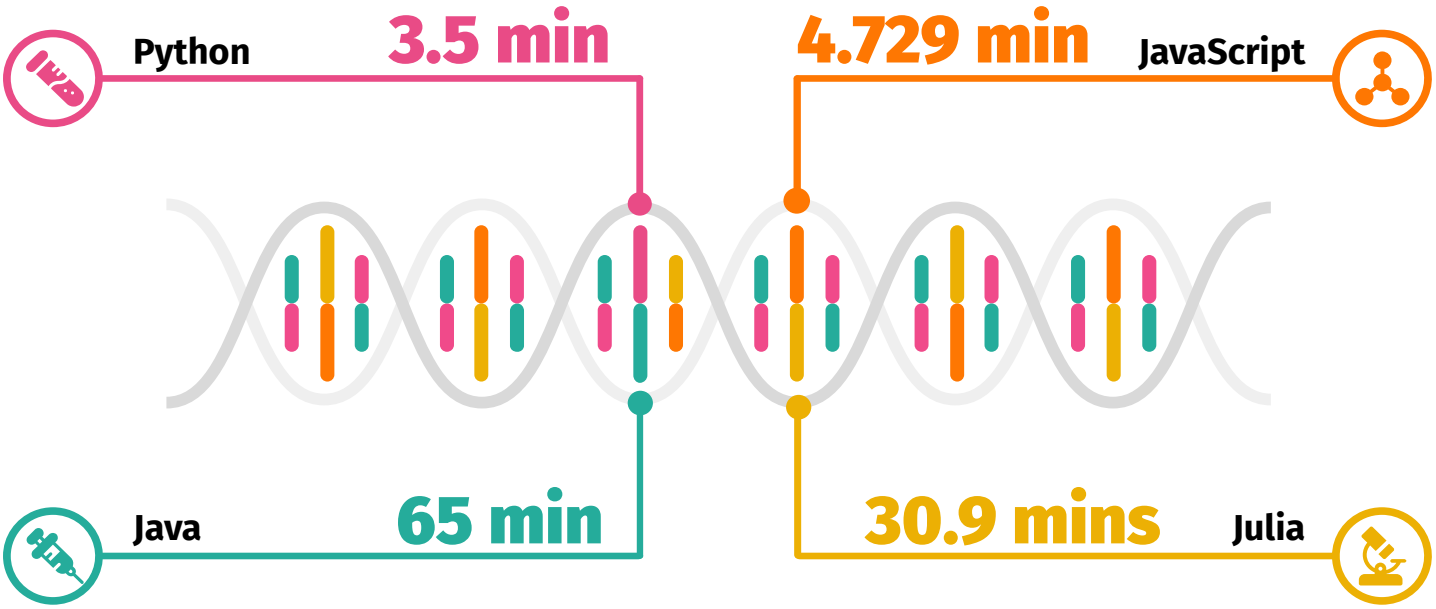


Runtime Analysis for BWT



For this analysis, U00096.3 Escherichia coli str. K-12 substr.MG1655 genome sequence was loaded. 10,000 bases was loaded as a string in both the languages. Performed on 2.4GHz Quad-Core 10th Generation Intel Core i5 , 8GB of 2133MHz LPDDR3 memory , Intel Iris Plus Graphics 655 - 1.5GB

Runtime Analysis for inverse BWT



For this analysis, U00096.3 Escherichia coli str. K-12 substr.MG1655 genome sequence was loaded. 10,000 bases was loaded as a string in both the languages. Performed on 2.4GHz Quad-Core 10th Generation Intel Core i5 , 8GB of 2133MHz LPDDR3 memory , Intel Iris Plus Graphics 655 - 1.5GB

Applications of Burrows Wheeler Transform

Compression

Using Run-Length Encoding file sizes are compared

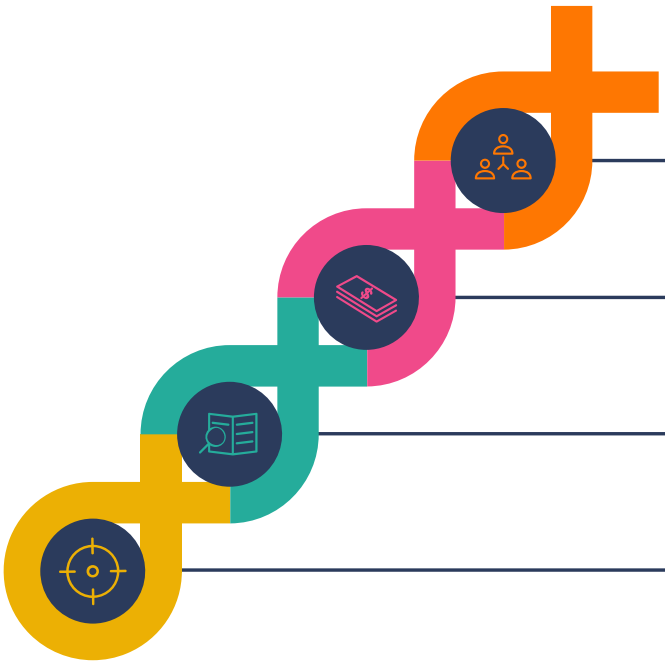
Pattern Matching

From the BWT-ed data, patterns can be found for analysis

Investigation into Genome Assembly

Our attempts to use BWT

Compression of Genome Sequences



01 Observing the Original FIle Size

02 Running the BWT algorithm

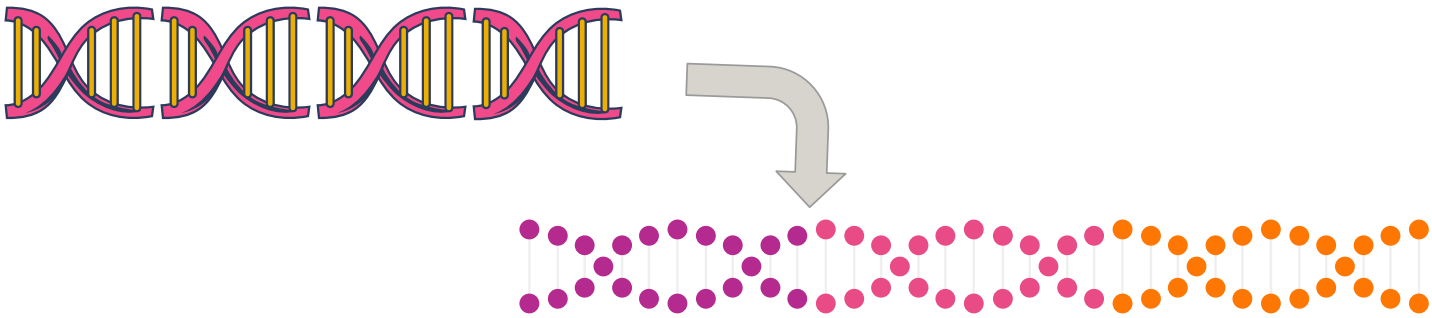
03 Run Length Encoding

04 Huffman Compression

Run-Length & Huffman via Burrow Wheelers

- The Efficiency of the algorithm is due to its capability to append similar strings together which aids in compression via Run-Length Encoding and Huffman Compression.

- Here's an example of Run-Length Encoding, "wwwwaaadexxxxxx" will be converted to "w4a3dex6" and Huffman algorithm assigns binary values depending on the frequency of recurrence of a string in a sequence.



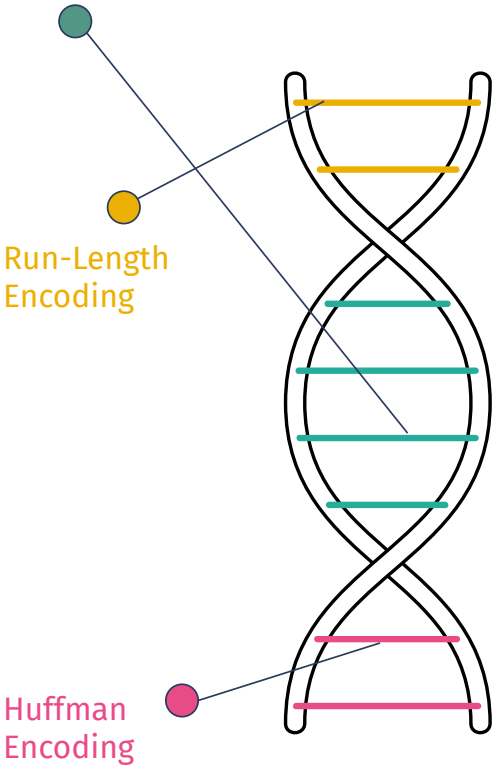
BWT Compression - Implementation

```
import itertools

def compress(string):
    return ''.join(
            letter + str(len(list(group)))
            for letter, group in itertools.groupby(string))
```

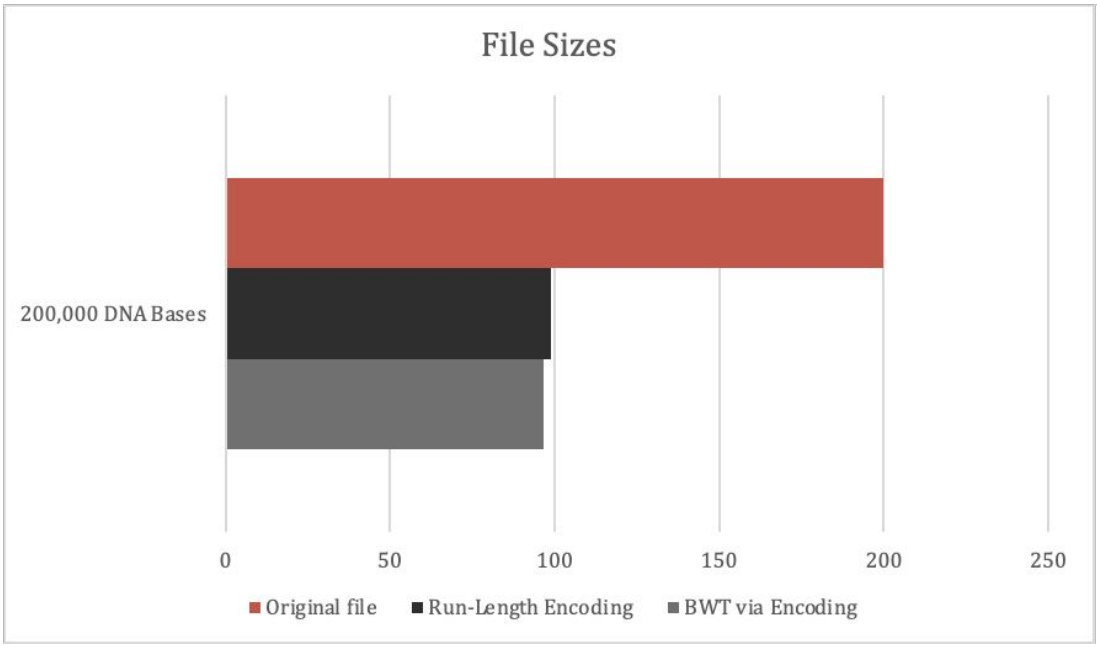
```
bwt_c=compress(T)
seq_c=compress(s)
```

```
s_s = len(s)*8
s_bwt  = 0
s_h=0
for i in bwt_c:
    s_bwt+=len(huffmanCode[i])
for i in seq_c:
    s_h+=len(huffmanCode1[i])
print("Original File Size:",s_s*0.000125,'kB')
print("Huffman  File Size after Length-Encoding:",s_h*0.000125,'kB')
print("Huffman via  Length-Encoding after BWT File:",s_bwt*0.000125,'kB')
```

```
Original File Size: 200.0 kB
Huffman  File Size after Length-Encoding: 98.81225 kB
Huffman via  Length-Encoding after BWT File: 96.807625 kB
```

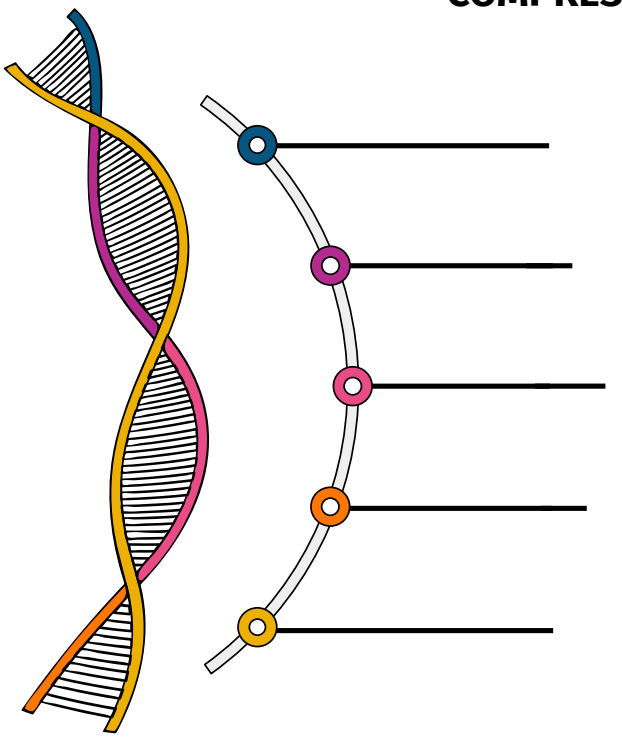


Result of using Huffman Encoding via BWT:



Char	Huffman code
'1'	11
'G'	101
'C'	100
'T'	011
'A'	010
'2'	001
'3'	0001
'4'	00001
'5'	000001
'6'	0000001
'7'	00000001
'8'	000000001
'9'	0000000001
'0'	00000000001
'$'	00000000000

COMPRESSION WITH BWT



The Burrows–Wheeler transform (BWT, also called block-sorting compression) rearranges a character string into runs of similar characters

The BWT algorithm has the ability to reconstruct the original data into a compressed data.

The major benefit of this implementation is that once the characters have been clustered together, they can have an effective ordering.

This makes our string a way more compressible for major algorithms like Huffman Coding , Move-to-front Transform and Run-length encoding.

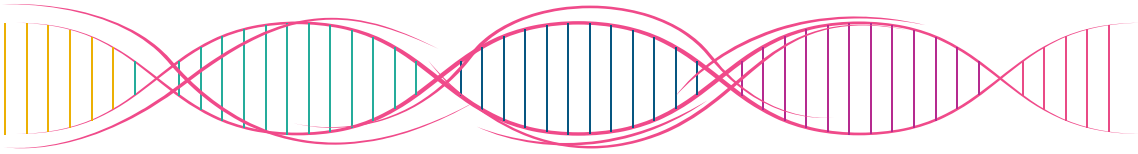
The Burrows–Wheeler transform is an algorithm used to prepare data for use with data compression techniques such as bzip2.

Move-To-Front Transform

- The move-to-front (MTF) transform is an encoding of data (typically a stream of bytes) designed to improve the performance of entropy encoding techniques of compression.

- When efficiently implemented, it is fast enough that its benefits usually justify including it as an extra step in data compression algorithm

- An important use of the MTF transform is in Burrows–Wheeler transform based compression. Compression benefits greatly from following up the BWT with an MTF transform before the final entropy-encoding step.



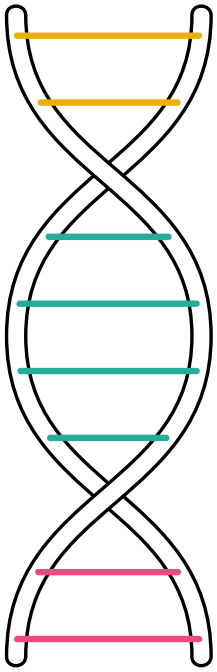
Move To Front - Implementation

MOVE TO FRONT TRANSFORM (MTF)

In [1]:
```python
cmn_dict = list(range(256))# Here we use the 256 possible values of a byte:
def encode(string):
    string = string.encode('utf-8') # Change to bytes for 256.
    dict = cmn_dict.copy()

    # Transformation
    comp_txt = list()    #represents list of compressed data in bytes
    rank = 0

    # Read in each character
    for i in string:
        rank = dict.index(i)    # Find the rank of the character in the dictionary
        comp_txt.append(rank)  # Update the encoded text

        # Update the dictionary
        dict.pop(rank)#To Update the dictionary
        dict.insert(0, i)

    return comp_txt
```

In [2]:
```python
X = encode("ipssm$pissii")
print(X)
```

```
[105, 112, 115, 0, 111, 40, 3, 4, 4, 0, 1, 0]
```



In [3]:

```python
def decode(comp_data):
    comp_txt = comp_data
    dict = cmn_dict.copy()
    decoded_str = []

    # Read in each rank in the encoded text
    for j in comp_txt:
        # Read the character of that rank from the dictionary
        decoded_str.append(dict[j])

        # Update the dictionary
        e = dict.pop(j)
        dict.insert(0, e)

    return bytes(decoded_str).decode('utf-8')  # Return original string
```

In [4]:

```python
print(decode(X))
```

```
ipssm$pissii
```

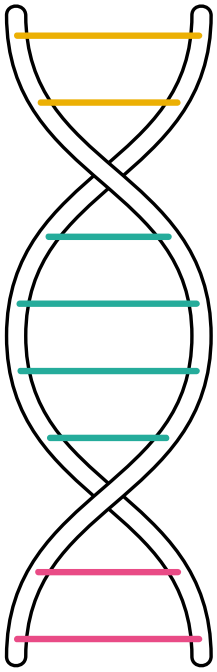
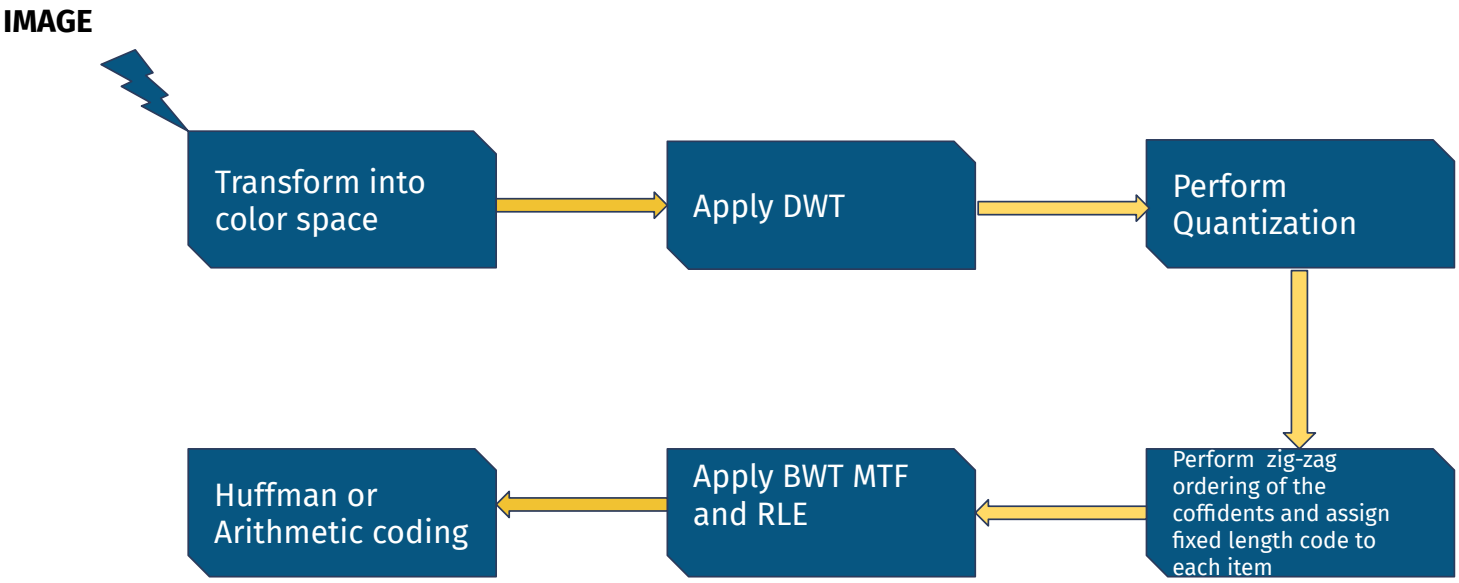


IMAGE COMPRESSION WITH BWT



- The ideal Burrows-Wheeler compression is quite slow due to sorting process and this is a major drawback of the process. Here, a new approach has been discussed in which BWT is applied before entropy encoding.

- In this method, the image is first transformed into color-space, then the image components are divided into tiles, one dimensional DWT applied and quantization is performed. After quantization, instead of using Huffman or arithmetic coding, each of the quantized coefficients are assigned a number and then BWT transformation is applied.

- With the use of BWT, data is transformed into formatted block of data which becomes easier to be compressed. MTE (Move to Front encoding) and RLE (Run length encoding) is performed on the data. The last step is compression of the image by the use of Huffman coding.

Pattern Matching

BWT

Finding the position of a particular pattern in the original sequence using only the BWT string

STEP-1

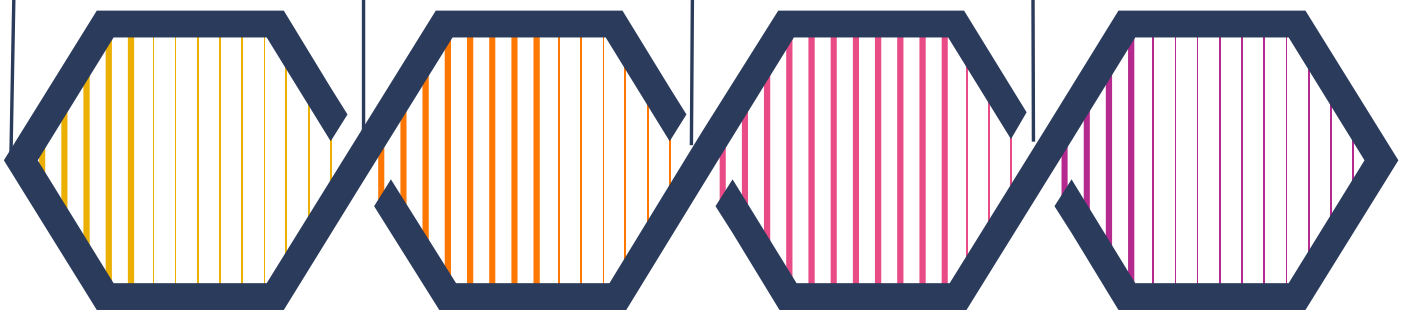
Using the BWT string, the BWT table can be constructed using cyclic rotations

STEP-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

STEP-3

This process is done iteratively for the length of the given pattern



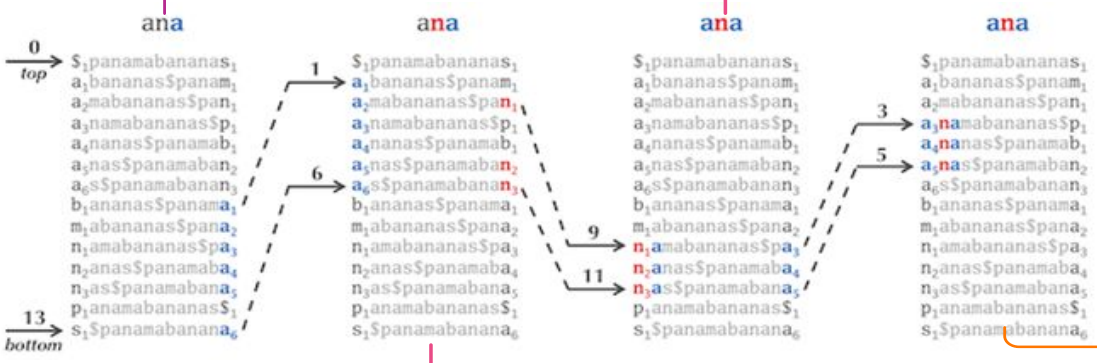
Pattern Matching

Example

Consider an example of "$panamabananas"

Here we have started with ana from the last a

Identifying the index of the 2nd last letter in the query in the 1st column and checking if it ends with the 1st letter in the query



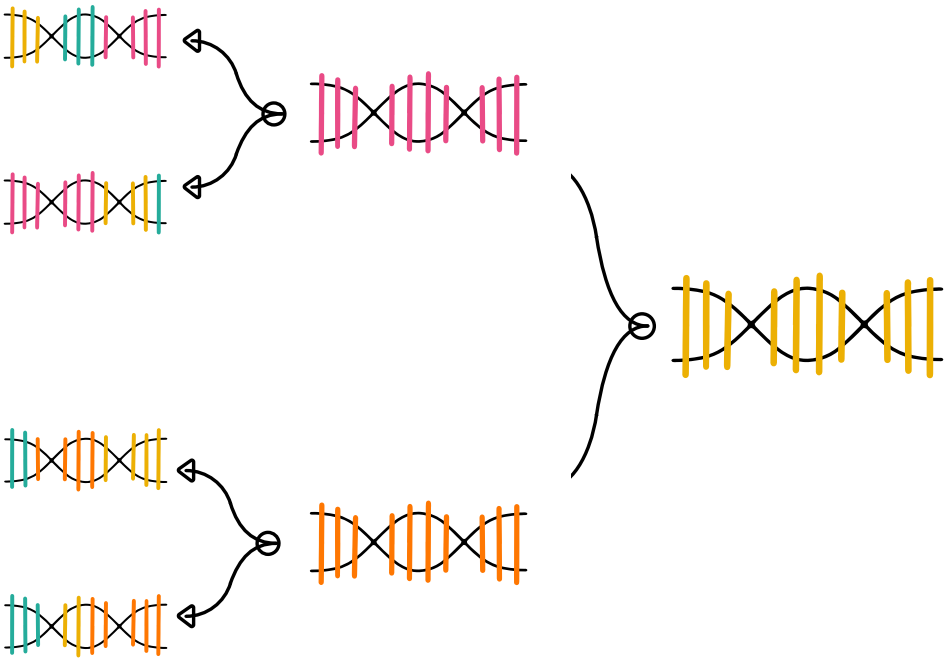
Identifying rows beginning with last element of pattern

Using the indexes obtained in the previous step, the rows containing the pattern can be found

Finding a match in BWT compressed Data

```
function exact_match(query,suffix_array,bwt_array){
    var C={};
    var ss = bwt_array.slice().sort(function(a, b){return a-b});
    for (var i = 0;i<ss.length;i++){
        if(!C[ss[i]]){
            C[ss[i]]=i;
        }
    }
    const query_array = fromstring(query);
    var start = 0;
    var end = bwt_array.length;
    c=query_array.reverse();
    for (var i=0;i<c.length;i++){
        rank_start = sumsplice(bwt_array,start,c[i]);
        rank_end = sumsplice(bwt_array,end,c[i]);
        start = C[c[i]] + rank_start;
        end = C[c[i]] + rank_end;
    }
    return suffix_array.slice(start,end).sort(function(a, b){return a-b});
}
```

Investigation into Genome Assembly

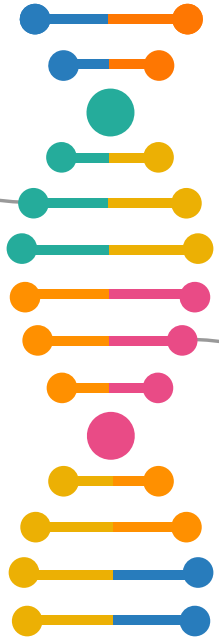


The given problem is to generate a sequence given reads of fixed length k.

Given a list of kmer reads, we will scrutinize on whether Burrow Wheelers Transform can aid in Sequence Construction.

Our Attempts for using BWT

Reconstruction using suffix prefix method



Cosine Distance Similarity

Analysing the Suffix-Prefix based Algorithm

```
km_list = sorted(extractKmer(km,5))
```

```
ind=[]
km_bwt=[]
for i in km_list:
    k,ii = bwt(i)
    km_bwt.append(k)
    ind.append(ii)
```

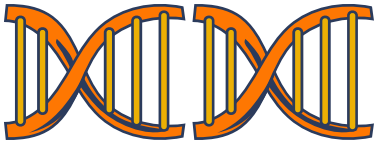
```
print(string_reconstruction_problem(km_list))
```

```
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAA
```

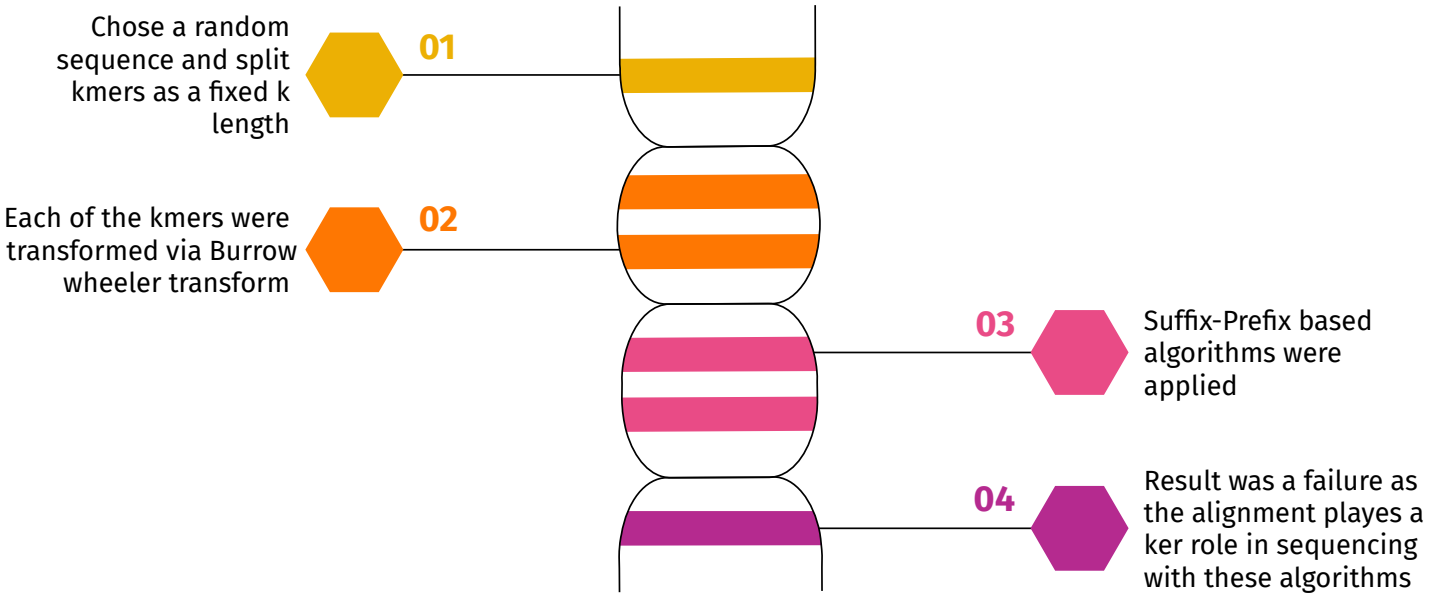
```
print(string_reconstruction_problem(km_bwt))
```

```
GAGGC
```

Primary experiment was to use the BWTed reads with an Eulerian path finding algorithm to see how the suffix prefix methods performs. Since BWT involves rigorous text rearrangement the original information is temporarily lost hence this method fails.



Procedure of the experiment performed



Experimenting via distance based computation

Cosine Distance Similarity

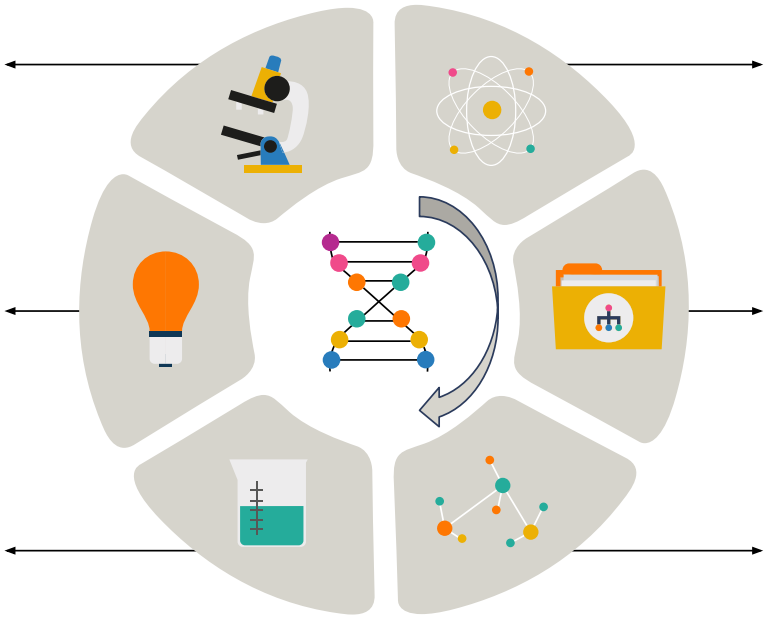
Computing the cosine similarity for each k-mer

Word2Vec

A neural network based algorithm for word embedding

Corpus Preparation

A list of the BWTed kmers was prepared for training



Loaded a fasta file

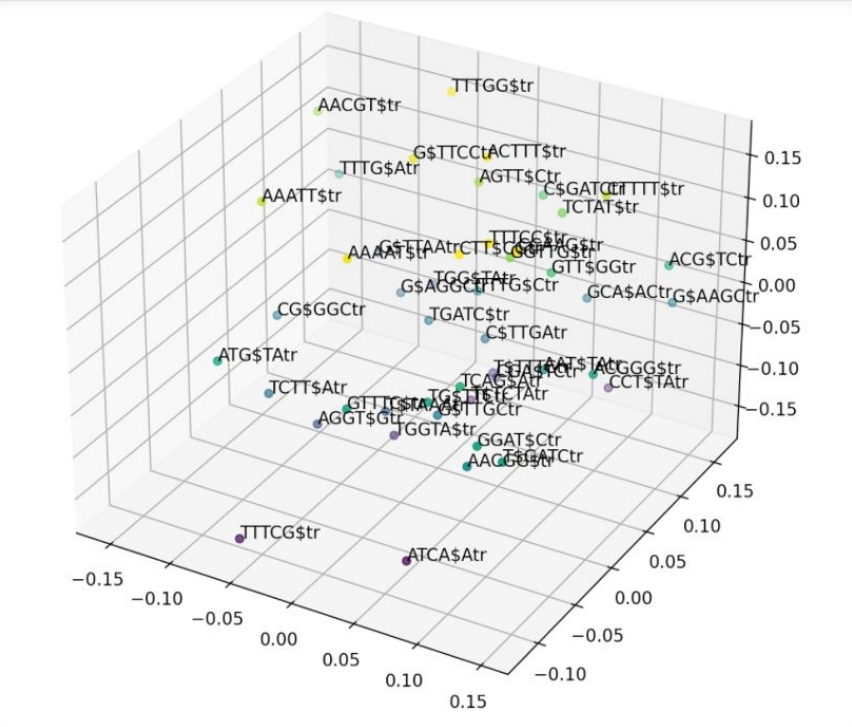
Kmer Extraction

For ease in computation a small sequence was extracted from the fasta file and kmers were extracted

BWT

BWT was performed on each kmer

Visual Representation of Vectorising kmers



Result of Cosine Similarity

```
bwtmodel.most_similar(positive=['G$AAGC'], topn = 5)
```

```
<ipython-input-12-7c7e17815894>:1: DeprecationWarning: Call to deprecated `most_similar`
(Method will be removed in 4.0.0, use self.wv.most_similar() instead).
  bwtmodel.most_similar(positive=['G$AAGC'], topn = 5)
```

```
[('ACG$TC', 0.9674394130706787),
 ('GCA$AC', 0.9432275891304016),
 ('C$TTGA', 0.8230779767036438),
 ('GTT$GG', 0.8011133074760437),
 ('CCT$TA', 0.8001498579978943)]
```

```
i_BWT('G$AAGC')
```

```
'AACGG'
```

```
i_BWT('CGA$TC')
```

```
'CTGAC'
```

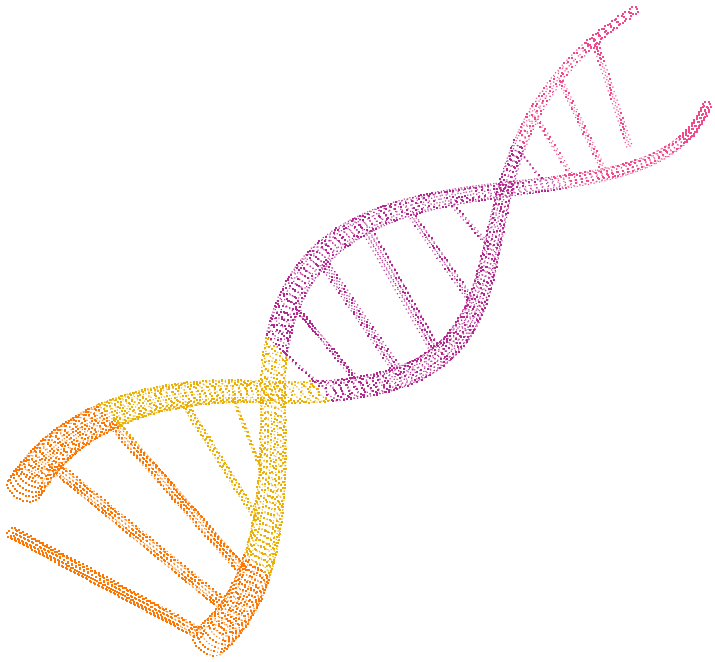
```
i_BWT('ACG$TC')
```

```
'CTGCA'
```

A random BWTed kmer was chosen and top five closest kmers were computed and inverted.

On inversing these kmers, “AACGG” was closest to “CTGCA” and on testing with various methods like Reverse Complement, a relation between “close” kmers couldn't be bridged.

Conclusion



- The results from various experiments conducted on the kmers from applying BWT and trying to to sequence them drastically fails as BWT's application is limited to compression and compression based pattern matching.
- As BWT involves rearrangement of string, the loss in the genomic arrangement affects its capability to help in genome reconstruction.
- There is a possibility of reconstruction a sequence via BWT with the help of a reference sequence, which will be an extension of Pattern MAtching.