Burrows-Wheeler Transform

and its applications towards

genome assembly

Done By -

K. Vinayak (CB.EN.U4AIE19032)

Manduva Gnanadeep (CB.EN.U4AIE19041)

Prashanth VR (CB.EN.U4AIE19047)

Vuyyuru Bindu Sri (CB.EN.U4AIE19069)



B.TECH CSE-AI, CEN DEPARTMENT, AMRITA SCHOOL OF ENGINEERING

**ACKNOWLEDGEMENT**

This project would not have been possible without the support and guidance of Dr. Sachin Kumar, 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 providing ample resources to avail our project needs and a platform for online lectures during this lockdown.

**CONTENTS**

1. LIST OF FIGURES………………………………………………………..4
2. ABSTRACT ………………………………………………………………. 5
3. INTRODUCTION ……………………………………………………........ 6
4. PROBLEM STATEMENT ...........................................................................7
5. BWT USING MOVE TO FRONT TRANSFORM ..................................... 8
6. ROLE OF BWT IN COMPRESSION ........................................................ 10
7. SUFFIX ARRAY......................................................................................... 11
8. IMPLEMENTING BWT USING SUFFIX ARRAYS ............................... 12
9. INVERSE BWT ………………………………………………………….. 14
10. K-MER FINDING ....................................................................................... 17
11. FM-INDEX ................................................................................................ 17
12. OFFSET ARRAY ......................................................................................18
13. APPLICATIONS .......................................................................................22
14. TIME ANALYSIS...................................................................................... 23
15. CHALLENGES IN GENOME ASSEMBLY USING BWT ..................... 24
16. IMPLEMENTATIONS IN DIFFERENT LANGUAGES………………..25

**LIST OF FIGURES:**

|  |  |
| --- | --- |
| **FIGURE** | **PAGE NO.** |
| **1** | **8** |
| **2** | **9** |
| **3** | **10** |
| **4** | **10** |
| **5** | **12** |
| **6** | **12** |
| **7** | **13** |
| **8** | **13** |
| **9.1** | **15** |
| **9.2** | **15** |
| **9.3** | **16** |
| **10** | **16** |
| **11** | **18** |
| **12** | **20** |
| **13** | **22** |

**ABSTRACT**

Burrows Wheeler Transform or BWT in short deals with the data compression and indexing and is used in the areas like storing DNA sequences, images, text, sound compressions, sequence alignment in genomes.

The input string given to the BWT algorithm is transformed in such a way that it can be recovered without much computation to be performed when compared to algorithms in this area.

*keywords -*  Suffix array, LF(LasttoFirst), String Rotation, FM index, exact String match, Permutation, BWT matrix, K-mer.

**INTRODUCTION**

The Burrows-Wheeler Transform (BWT) or block sorting compression idea was started by David Wheeler in his unpublished paper in 1983 and later it was modified by Michael Burrows along with the founder in 1994 at DEC Systems Research Center in Palo Alto, California.

In this report we are going to present BWT implementation using Move to Front Transform, Suffix Array methods in detail. Since BWT is easy to implement it has got many applications in medical fields like storing DNA like large sequences, pattern matching using FM indexing, and in other fields where large data is stored using compression methods.

**PROBLEM STATEMENT**

To find the Burrows-Wheeler transform for a given Genome or DNA sequence using various techniques, find the Inverse Burrows-Wheeler transform for a given BWT sequence and discuss in detail about the applications of BWT in genome assembly as well as in other domains.

**BWT USING MOVE TO FRONT TRANSFORM**

Let S be the original string, we rotate the string S ‘n’ times in clockwise direction i.e we move the last character in the string to the first place, where ‘n’ corresponds to the length of the string S and we store the rotated strings in a matrix called BWT matrix.

Let the string S be ‘BANANA$’ and let’s take ‘$’ as the reference symbol to denote the end of string. So if the ‘$’ appears at the start of the string then we can confirm that the rotation got completed.

Let S = BANANA$ be the string then -

Rotations of S -

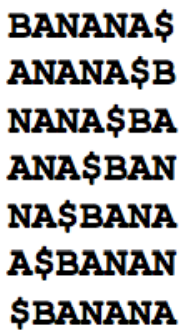


Fig-1

Lexicographically arranged order of the rotated sequences (BWT MATRIX):

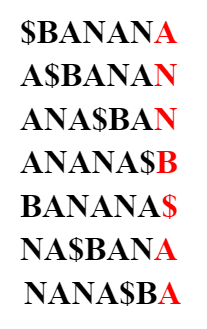
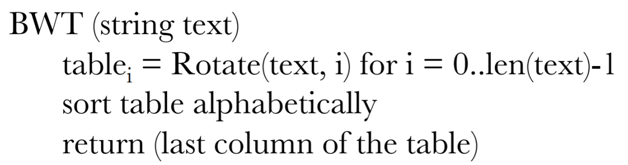


Fig-2

The last column is the rearranged sequence of the original string which has least entropy among all the columns in BWT matrix. This column helps us to retrieve the original string without any additional information(except the index where the sequence occurred in the BWT matrix). So we can conclude BWT as one of the best methods to store or retrieve the information without any additional cost.

**BWT(BANANA$) = ANNB$AA**

**BWT Pseudo Code**



**ROLE OF BWT IN COMPRESSION**

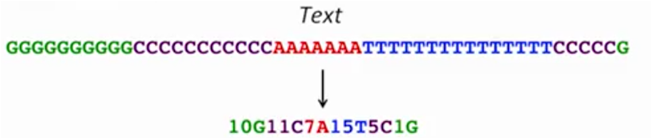
BWT is mentioned to be the method used for storing information because it has the capability to decrease the randomness and to form runs in the string. Let’s look at an example to understand how BWT is used in compression of DNA sequences. 

Fig-3

The below figure shows how BWT is implemented in the compression algorithm.

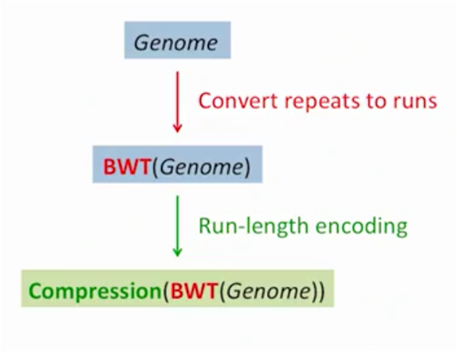


Fig-4

**SUFFIX ARRAY**

Suffix array is a data structure which is a sorted array of all the suffixes obtained from a given text or a string and usually used in storing full string indices and data compression algorithms. This can be used as an index for quickly locating an occurrence of a pattern or a substring given the parent string.

So, finding the occurrence of a pattern is the same as finding all the suffixes that begin with the substring. As already they are lexicographically arranged, searching for suffixes will be efficient within 2 binary searches. First binary search locates the starting position and the second locates the ending position of the pattern.

This algorithm is very efficient in computing Burrows-Wheeler transform (BWT). This involves sorting cyclic permutations of a string and can be computed in linear time by first constructing a suffix array and then extracting the BWT sequence from it. The method used above to build a suffix array which has O(n^2Logn) time complexity, due to O(n) time for string comparisons in O(nLogn) sorting algorithm.

**IMPLEMENTING BWT USING SUFFIX ARRAYS**

1. First, we should store all the cyclic permutations of a given string and store them in an array. Here in this scenario, let us consider a standard example string “BANANA$”.

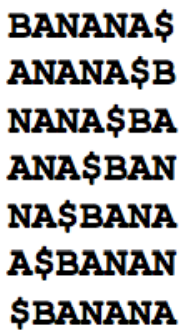


Fig-5

1. Next sort them in lexicographical order and figure out all the suffixes for all the possible permutations.

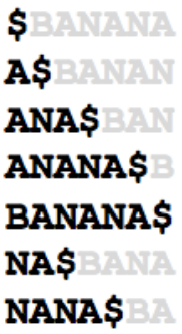


Fig-6

1. Generally, Suffix array S is just an array containing indices, where S[i] contains the index of the ith smallest suffix occurring in the parent string “BANANA$”. For example, S[1] contains the index of the 1st smallest suffix ie, “$” occurring in the string “BANANA$”. This index corresponds to the last index of the original string which is 7.

Finally the Suffix array (at the bottom of the Fig.) will look something like this from the original string (at the top of the Fig.).

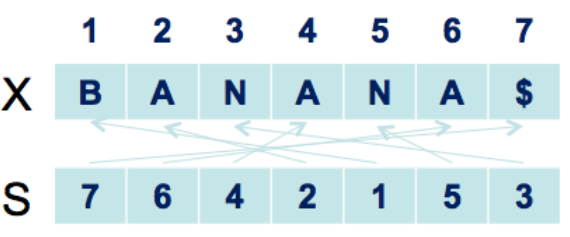


Fig-7

Constructing a BWT sequence from the given Suffix array is as simple as we could imagine. First, we will go to the character of the original sequence indicated by the index S[i] and then move one spot to the left of the indicated index.

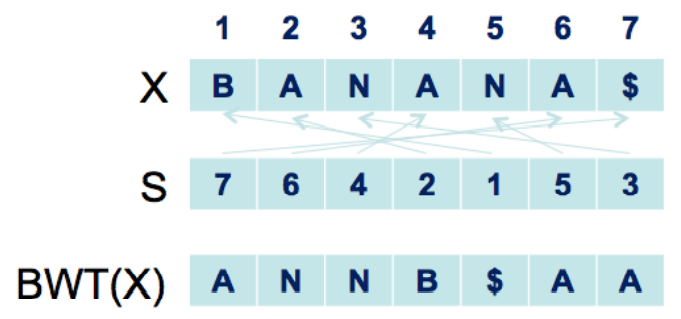


Fig-8

**INVERSE BURROWS WHEELER TRANSFORM**

Inverse BWT is to undo the BWT and get up the original string. It is said that BWT is reversible permutation, which means with only BWT, it will be able to construct the original string. Here we have only two things, that is the BWT array which contains the last column of sorted rotations, “annb$aa” and index at which the original string“banana$” in the sorted rotation list i.e. index 4 in the sorted rotations.

|  |  |  |
| --- | --- | --- |
| **Row Index** | **Original Rotations** | **Sorted Rotations** |
| **0** | *banana$* | *$banana* |
| **1** | *anana$b* | *a$banan* |
| **2** | *nana$ba* | *ana$ban* |
| **3** | *ana$ban* | *anana$b* |
| **4** | *na$bana* | *banana$* |
| **5** | *a$banan* | *na$bana* |
| **6** | *$banana* | *nana$ba* |

**In Simple Approach**

· We append BWT to an empty array as the last column of the array.

· Then we sort the column.

· We do this for n times, where n is length of String

After first iteration,



Fig – 9.1

After second iteration,

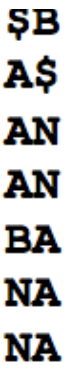


Fig-9.2

After n iteration,

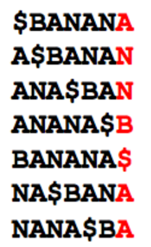


Fig-9.3

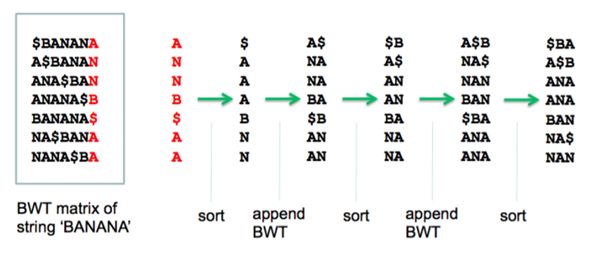
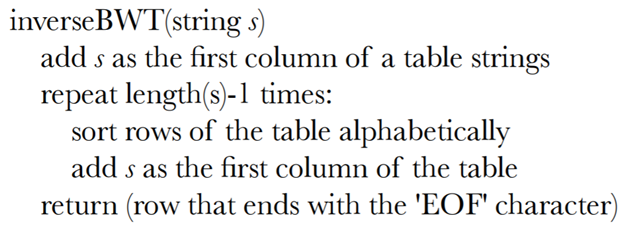


Fig-10

**Inverse BWT Pseudo Code**



**K-MER FINDING**

K-mer refers to a string of length ‘k’ generally in DNA sequences. Here we are trying to optimise time by using BWT to find K-mers. There are many methods to find K-mers in the genome like Brute Force, Suffix Array, Suffix Tree, But this particular BWT is employed for reducing time and complexity.

To know about BWT EXACT MATCH firstly we should be aware of FM-index, Offset array, Predecessor Suffix. Let’s look at these in detail.

**FM-INDEX**

FM index stands for Full-text index in Minute space and was created by Paolo Ferragina and Giovanni Manzini to minimise time complexity in generating substrings. It is basically a compressed full-text [substring index](https://en.wikipedia.org/wiki/Substring_index) known to compress the input text and creates substring queries in no time.

In general FM-index denotes the number of times a symbol occured before the given position.

* N denotes the text length
* F[i][c] stores the number of times symbol ‘c’ occurs before index i

The FM-index of string ‘ $BANANA’ is as follows

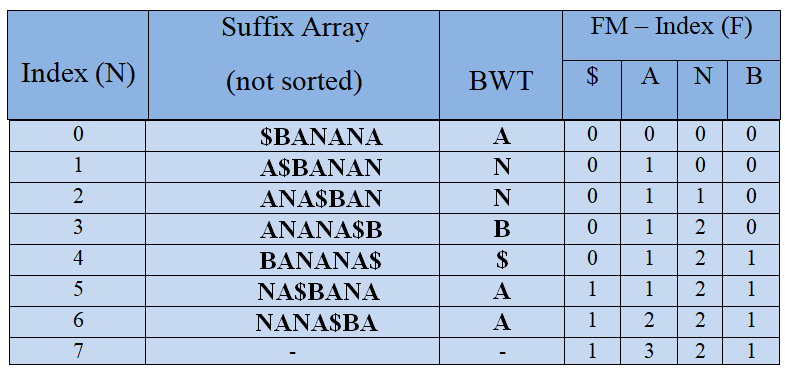


Fig-11

**OFFSET ARRAY**

Offset generally means the number of previous elements to cross to reach the current element. It is the same here to compute the number of previous elements.

Offset array is indicated with the letter O. O[c] stores the index of the first suffix starting with symbol c. It can be computed in two ways using the FM-index matrix that we created before or directly from the BWT string. We’ll discuss these two methods in detail.

* Offset Array using FM-index

It can be derived directly from the final entry in F i.e.

$ A B N

F[lastRow] 1 3 2 1

O[B] = sum(F[lastRow][$:B]) =1+3 = 4

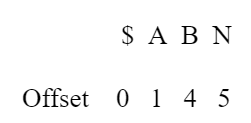
* Offset Array using BWT(string)

BWT(BANANA$) = ANNB$AA

Arrange it in lexicographic order ; $AAABNN

Indices -

$AAABNN

0 1 2 3 4 5 6

Get the first occurence of each symbol -

Predecessor Suffix

Given an index i in the BWT, find the index in the BWT of the suffix preceding the suffix represented by i

* The predecessor suffix of index i

c = BWT[i]

predec = O[c]+F[i][c]

Predecessor of index 1

c = BWT[1] = ‘N’

predec = O[‘N’]+F[1][‘N’] = 5+0 = 5

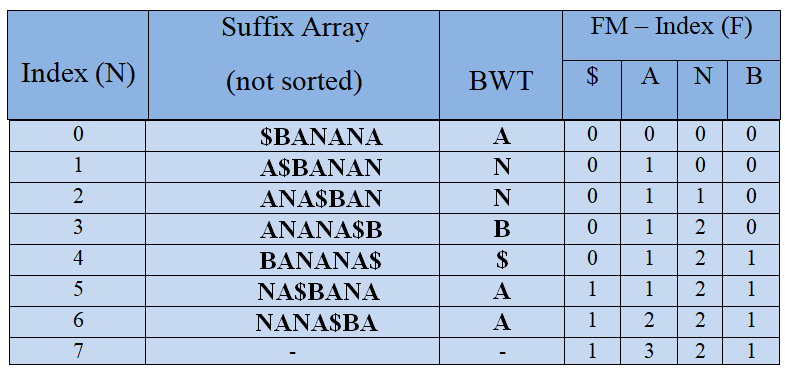
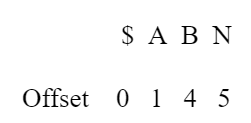


Fig-12



* Time to find predecessor: O(1)

Find K-mer

* All searches occur in reverse order
* Start with full BWT range (0, N)
* Restrict by one symbol at a time

Let’s look at an example of k-mer “AN”

Initializing to full range

low = 0

high = 6

Find occurrences of “N”

Updating low & high each time we iterate

low = O[‘N’]+F[low][‘N’] = 5+0 = 5

high = O[‘N’]+F[high][‘N’]-1= 5+2-1= 6

Find occurrences of “AN”

low = O[‘A’]+F[low][‘A’] = 1+1 = 2

high = O[‘A’]+F[high][‘A’]-1= 1+3-1 = 3

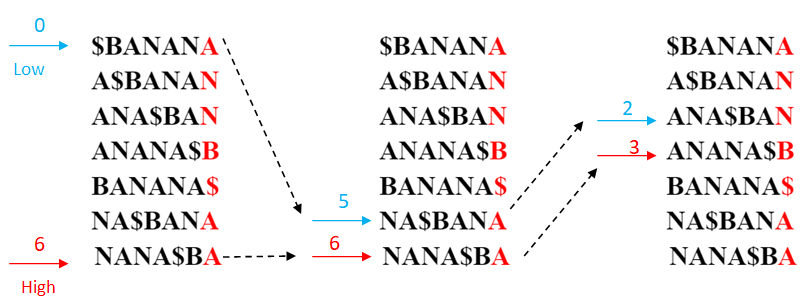
* Does not depend on BWT (data) size 

Fig-13

**APPLICATIONS**

Traditionally the major application of the Burrows-Wheeler Transform has been for data compression and efficient storage. Recent research on the Burrows-Wheeler Transform has shown the versatility of the BWT, and hence efforts are shifting from its traditional application in data compression to other areas of study.

Data compression techniques such as bzip2 use Burrows–Wheeler transform to prepare the data. Next-generation sequencing is any of several high-throughput approaches to DNA sequencing using the concept of massively parallel processing it is also called second-generation sequencing or Massive parallel sequencing.

Some special characteristics of the BWT, include clustering property. BWT is also used in shape analysis in computer vision, and in machine translation. One such application of the BWT is in bioinformatics and computational biology, full-text compressed indexes, prediction and entropy estimation, and recent approaches in joint-source channel coding.

**TIME ANALYSIS**

|  |  |  |
| --- | --- | --- |
| Language | Time Taken for BWT | Time Taken for IBWT |
| Julia | 0.070155 s | 0.053870 s |
| Javascript | 0.048 ms | 8.827 ms |
| Python | 0.932507 s | 0.001993 s |
| R | 0.05s | 0.012 s |
| Java | 0.00238 s | 0.3076098 s |

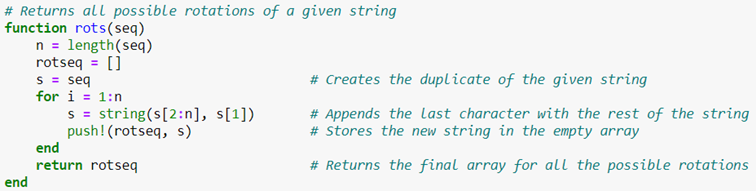
**CHALLENGES IN GENOME ASSEMBLY USING BWT**

Genome assembly is a widely concerned area in bioinformatics where we are given large lengthened reeds and asked to find the original DNA sequence. DNA is about a million to trillion lengthened sequence since it is impossible to fit that entire sequence in computers while dealing with its properties. We split it into multiple reeds and work on them.

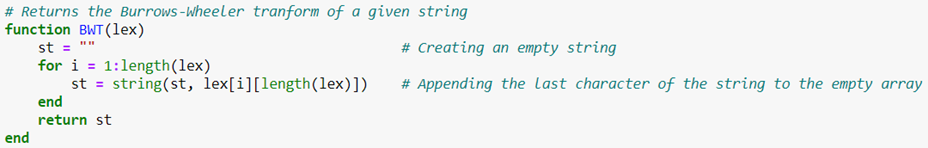
BWT is used when we are familiar with the original sequence and want to find it’s permutation string i.e. the string with more information and least entropy. Using this BWT string we can find the original genome sequence with no extra information except the index where the original string occurred in the BWT matrix. So it is necessary for us to be familiar with the original genome sequence while working with BWT. But in the real word scenario we won’t be knowing the original DNA sequence in prior. This is the main challenge faced in genome assembly.

Generally, we will perform a prefix suffix match to arrive at the original sequence from the given reeds by using string reconstruction methods like Hamiltonian, Eulerian walks and debruijn path. In BWT we can’t perform prefix suffix match. So this is going to be one more challenge in genome assembly approach using BWT.

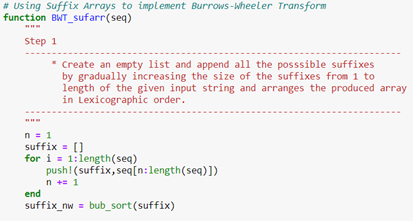
**IMPLEMENTATION IN JULIA**

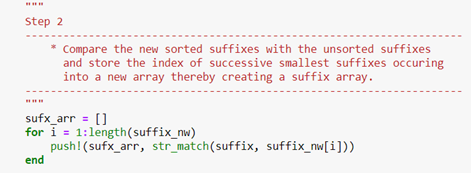
****

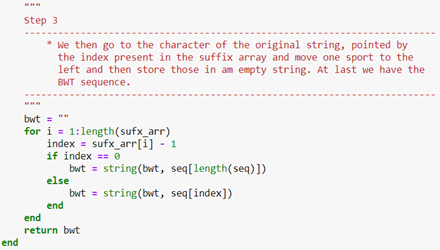
Function rots() returns all the possible rotations in a matrix format given a string as input to the function.

****

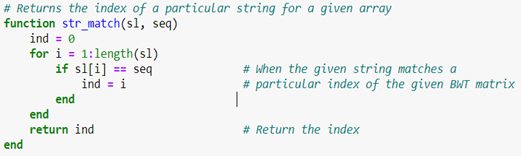
Function BWT() returns the BWT sequence given the input matrix which contains all the possible rotations of a particular string.



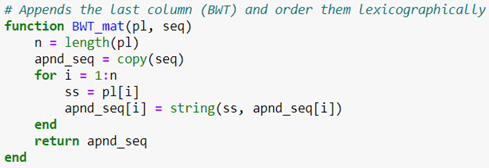


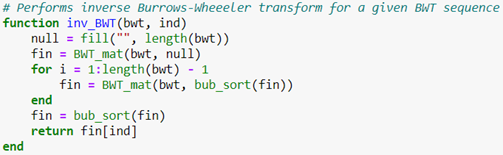
****

Function BWT\_sufarr() returns the BWT sequence given the appropriate string to be compressed using the suffix array approach.

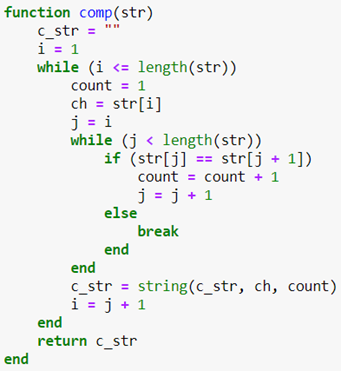
****

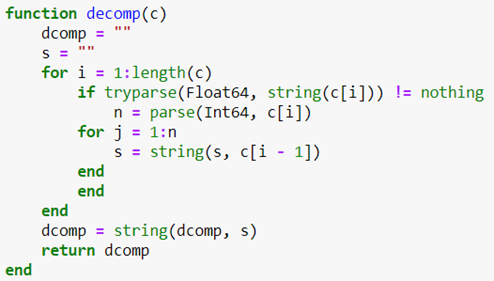
Function str\_match() returns the index of the input string in the given BWT matrix.

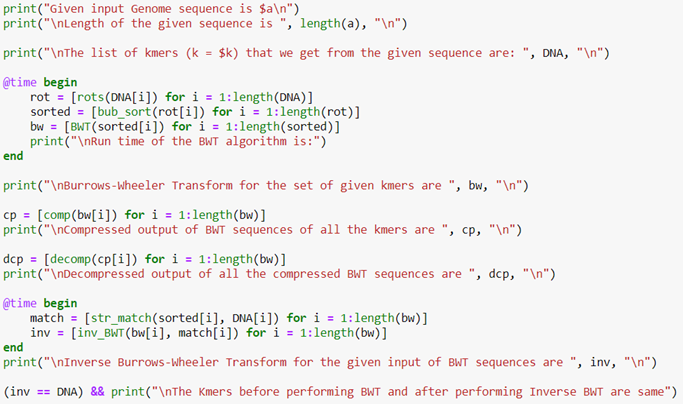
****

****

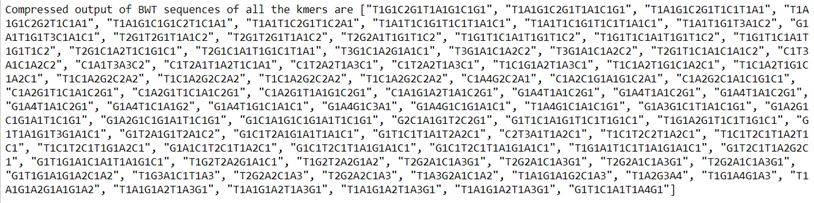
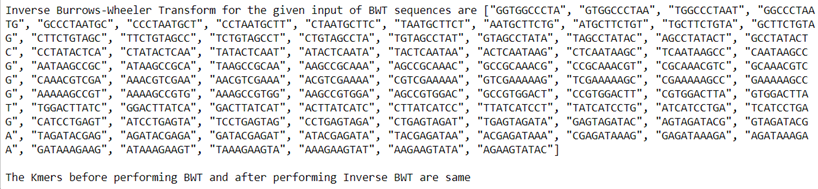
Function inv\_BWT() returns the original sequence by performing inverse BWT given the BWT sequence and index where the original sequence is stored in the BWT sequence.

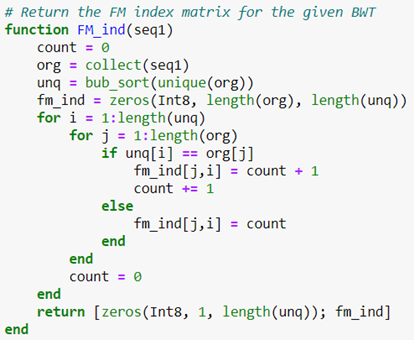
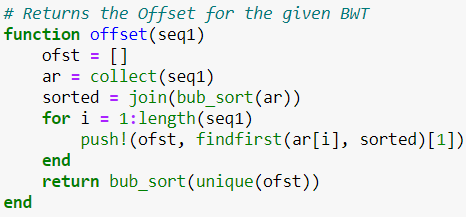


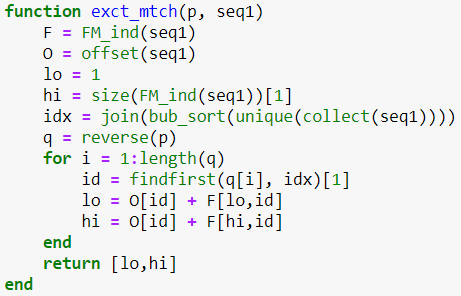
Function comp() compresses a given string appending first the character and succeeded by the count of the preceding character. Function decomp() decompresses a given compressed string by repeatedly pasting the character by the count given after each character in the string.

****





****



Function exct\_match is used to find the position of k-mer in BWT matrix using FM index matrix and Offset array that were computed earlier to update low and high pointers and finally converges to a position where the K-mer can be found.

**IMPLEMENTATION IN JAVASCRIPT**

* ```javascript
* **var** seq = 'AGCAGCCC';
* **function** rotate\_seq(seqnce){
* **var** rot\_seq=[];
* **var** N = seqnce.length;
* **var** s = seqnce;
* **for**(**var** i=0;i<N;i++){
* **var** s=s.substring(1)+s.substring(0,1);
* rot\_seq[i]=s;
* }
* **return** rot\_seq
* }
* ```

* ```javascript
* rt\_sq=rotate\_seq(seq)
* ```

* [ 'GCAGCCCA',
* 'CAGCCCAG',
* 'AGCCCAGC',
* 'GCCCAGCA',
* 'CCCAGCAG',
* 'CCAGCAGC',
* 'CAGCAGCC',
* 'AGCAGCCC' ]
* ```javascript
* **function** bub\_sort(unsrt\_sequence){
* **var** sequence =[...unsrt\_sequence];
* **var** temp;
* **for**(**var** i=0;i<(sequence.length)-1;i++){
* **for**(**var** j=0;j<(sequence.length)-i;j++){
* **if**(sequence[j]>sequence[j+1]){
* temp=sequence[j];
* sequence[j]=sequence[j+1];
* sequence[j+1]=temp;
* }
* }
* }
* **return**(sequence)
* }
* ```
* ```javascript
* srt\_sq=bub\_sort(rt\_sq)
* ```
* [ 'AGCAGCCC',
* 'AGCCCAGC',
* 'CAGCAGCC',
* 'CAGCCCAG',
* 'CCAGCAGC',
* 'CCCAGCAG',
* 'GCAGCCCA',
* 'GCCCAGCA' ]
* ```javascript
* **function** permutation\_string(Lexseq){
* **var** srt\_sq=[...Lexseq];
* **var** st='';
* **for**(**var** i=0;i<Lexseq.length;i++){
* st = st + srt\_sq[i].substring((srt\_sq[i].length-1),srt\_sq[i].length);
* }
* **return**(st)
* }
* ```
* ```javascript
* pmt\_str=permutation\_string(srt\_sq)
* ```
* 'CCCGCGAA'
* ```javascript
* **function** str\_match(seq\_list,seqn){
* **for**(**var** i=0;i<seq\_list.length;i++){
* **if**(seq\_list[i]==seqn){
* **var** strnum = i+1;
* }
* }
* **return**(strnum)
* }
* ```
* ```javascript
* str\_num=str\_match(srt\_sq,seq)
* ```
* 1
* ```javascript
* **function** BWT\_mtrx(pmtstr,apnd\_seq){
* **var** N = pmtstr.length;
* **for**(**var** i=0;i<N;i++){
* ss = pmtstr.substring(i,i+1);
* apnd\_seq[i]=ss+apnd\_seq[i]
* }
* **return**(apnd\_seq)
* }
* ```
* ```javascript
* fin=BWT\_mtrx(pmt\_str,[])
* fin1=BWT\_mtrx(pmt\_str,bub\_sort(fin))
* fin2=BWT\_mtrx(pmt\_str,bub\_sort(fin1))
* fin3=BWT\_mtrx(pmt\_str,bub\_sort(fin2))
* fin4=BWT\_mtrx(pmt\_str,bub\_sort(fin3))
* ```
* [ 'CAGCAundefined',
* 'CAGCCundefined',
* 'CCAGCundefined',
* 'GCAGCundefined',
* 'CCCAGundefined',
* 'GCCCAundefined',
* 'AGCAGundefined',
* 'AGCCCundefined' ]
* ```javascript
* **function** inv\_BWT(pmt\_str,str\_num){
* **var** fin=BWT\_mtrx(pmt\_str,[]);
* **for**(**var** i=0;i<(pmt\_str.length-1);i++){
* **var** fin=BWT\_mtrx(pmt\_str,bub\_sort(fin));
* }
* **for**(**var** j=0;j<pmt\_str.length;j++){
* fin[j] = fin[j].substring(0,pmt\_str.length);
* }
* fin = bub\_sort(fin)
* **return**(fin[str\_num])
* }
* ```
* ```javascript
* fin\_str=inv\_BWT(pmt\_str,str\_num)
* fin\_str
* ```
* 'AGCCCAGC'
* ```javascript
* **var** sq2='BANANA'
* **var** sp\_char ='1'
* **var** sq2 = sq2+sp\_char
* rt\_sq2=rotate\_seq(sq2)
* ```
* [ 'ANANA1B',
* 'NANA1BA',
* 'ANA1BAN',
* 'NA1BANA',
* 'A1BANAN',
* '1BANANA',
* 'BANANA1' ]

* ```javascript
* **function** suffix\_array(sqnce){
* **var** cln\_sq = sqnce;
* **var** N = cln\_sq.length;
* **var** sufx\_arr = [];
* **var** S = [];
* **for**(**var** i =0;i<N;i++){
* sufx\_arr[i] = cln\_sq.substring(i,cln\_sq.length);
* }
* sorted\_sufx\_arr = bub\_sort(sufx\_arr)
* **for**(**var** k=0;k<sufx\_arr.length;k++){
* S[k]=str\_match(sufx\_arr,sorted\_sufx\_arr[k]);
* }
* **return**(S)
* }
* ```
* ```javascript
* sa = suffix\_array(sq2)
* ```
* [ 7, 6, 4, 2, 1, 5, 3 ]
* ```javascript
* **function** BWT\_sfx\_arr(sfx\_arr,seq){
* **var** str ='';
* **for**(**var** i=0;i<sfx\_arr.length;i++){
* **if**(sfx\_arr[i] > 1){
* str = str + seq[sfx\_arr[i]-2];
* }
* **else**{
* str = str + seq[seq.length-1];
* }
* }
* **return**(str)
* }
* ```
* ```javascript
* bwt\_sq = BWT\_sfx\_arr(sa,sq2)
* ```
* 'ANNB1AA'
* ```javascript
* **function** count\_arr(last\_col){
* **var** unq = [];
* **var** cnt = 0;
* **var** unq = [...**new** Set(last\_col)];
* unq = bub\_sort(unq);
* **var** cnt\_arr = Array.from(Array(last\_col.length+1), () => **new** Array(unq.length))
* **for**(**var** j =0;j<unq.length;j++){
* **for**(**var** i =1;i<cnt\_arr.length;i++) {
* **if**(unq[j]==last\_col[i-1]){
* cnt\_arr[i][j]=cnt+1;
* cnt = cnt+1;
* }
* **else**{
* cnt\_arr[i][j]=cnt;
* }
* }
* cnt = 0;
* cnt\_arr[0][j]=0;
* }
* **return**(cnt\_arr)
* }
* ```
* ```javascript
* ca = count\_arr(['A','N','N','1','B','A','A'])
* ```
* [ [ 0, 0, 0, 0 ],
* [ 0, 1, 0, 0 ],
* [ 0, 1, 0, 1 ],
* [ 0, 1, 0, 2 ],
* [ 1, 1, 0, 2 ],
* [ 1, 1, 1, 2 ],
* [ 1, 2, 1, 2 ],
* [ 1, 3, 1, 2 ] ]
* ```javascript
* **function** count (string) {
* **var** count = {};
* string.split('').forEach(s);
* **function** s (s) {
* count[s] ? count[s]++ : count[s] = 1;
* }
* **return** count;
* }
* ```
* ```javascript
* count('1BANANA')
* ```
* { '1': 1, B: 1, A: 3, N: 2 }
* ```javascript
* **function** str2unq(st){
* **var** s = [...st];
* **var** cc = count(st)
* **var** n = Object.keys(cc).length;
* **var** x = [];
* **var** y = [];
* **for**(**var** i =0;i<s.length;i++){
* **if**(cc[s[i]]>1){
* **for**(**var** k=1;k<=cc[s[i]];k++){
* x.push(s[i]+k);
* }
* }
* **else**{
* s[i]=s[i]+'1'
* x.push(s[i]);
* }
* }
* x\_unq = [...**new** Set(x)];
* x = bub\_sort(x\_unq);
* **return**(x);
* }
* ```
* ```javascript
* s=str2unq('ANNB1AA')
* s
* ```
* [ '11', 'A1', 'A2', 'A3', 'B1', 'N1', 'N2' ]
* ```javascript
* **var** p =['1','A','A','A','B','N','N']
* p.indexOf('N')
* ```
* 5



* ```javascript
* **function** offset(Lc){
* **var** O =[];
* **var** st = bub\_sort(Lc);
* **for**(**var** i=0;i<st.length;i++){
* O.push(st.indexOf(st[i]));
* }
* ofst = [...**new** Set(O)];
* **return**(ofst);
* }
* ```
* ```javascript
* offset(['A','N','N','1','B','A','A'])
* ```
* [ 0, 1, 4, 5 ]
* ```javascript
* **function** ExactMatch(p,last\_col){
* **var** F = count\_arr(last\_col);
* **var** O = offset(last\_col);
* **var** lo = 0;
* **var** hi = F.length-1;
* **var** idx = bub\_sort([...**new** Set(last\_col)]);
* **var** q = [...p].reverse().join("");
* **for**(**var** i=0;i<q.length;i++){
* **var** id = idx.indexOf(q[i]);
* lo = O[id] + F[lo][id];
* hi = O[id] + F[hi][id];
* //console.log([lo,hi])
* }
* **return**([lo,hi]);
* }
* ```
* ```javascript
* //'ACG',['A','C','G','1','C','A','A','A','G','C']
* ExactMatch('NA',['A','N','N','1','B','A','A'])
* ```
* [ 1, 4 ]
* [ 5, 7 ]
* [ 5, 7 ]



* ```javascript
* **function** compress\_string(str){
* **var** c\_str = "";
* **var** i = 0;
* **while**(i<str.length){
* **var** count = 1;
* **var** ch = str[i];
* **var** j = i;
* **while**(j<str.length){
* **if**(str[j]==str[j+1]){
* count = count + 1;
* j=j+1;
* }
* **else**
* **break**
* }
* c\_str = c\_str + ch + count;
* i = j+1;
* }
* **return**(c\_str);
* }
* ```
* ```javascript
* **var** str = "caaaccgtappwq";
* compress\_string(str)
* ```
* 'c1a3c2g1t1a1p2w1q1'
* ```javascript
* **function** expand\_string(str){
* **var** dcomp = "";
* str = str + "j";
* **var** s = "";
* **for**(**var** i=0;i<str.length;i++){
* **if**(str[i].localeCompare(str[i+1])==-1){
* n = parseInt(str[i]);
* **for**(**var** j=0;j<n;j++){
* s = s+ str[i-1]
* }
* }
* }
* dcomp = dcomp+s;
* **return**(dcomp)
* }
* ```

* ```javascript
* expand\_string("p5q1w2r3")
* ```
* 'pppppqwwrrr'

**IMPLEMENTATION IN JAVA**

|  |
| --- |
| package exceptions;  import java.io.BufferedReader; package exceptions;  import java.io.BufferedReader;  import java.io.IOException; |

|  |
| --- |
| import java.io.InputStreamReader; import java.util.Arrays; import java.util.List;     public class **SuffixArray**  {  private String[] text;  private int length;  private int[] index;  private String[] suffix;  private String[] new\_text;  private String new\_text1;  private String[] unq;   */\*  \* SuffixArray   \* Substring the input text and store each element to the text array  \* index array is assigned with the size of the input text  \* Suffix array is assigned with the size of the input text  \*/*    public **SuffixArray**(String text)  {    new\_text1 = text; *//Storing the original string*   this.text = new String[text.length()];   for (int i = 0; i < text.length(); i++)  {  this.text[i] = text.substring(i, i+1);  }     this.length = text.length(); *//length is assigned with the size of the input string*  this.index = new int[length];  for (int i = 0; i < length; i++)  {  index[i] = i;  }     suffix = new String[length];  new\_text = this.text;    }    */\*   \* Create Suffix Array   \* Suffix after each element is rotated is stored to the Suffix Array   \* suffix array and index array is sorted using sortStrings function  \* Burrow wheelers Transform of the input string with suffix array is generated using BWT\_suffixarray function  \* Input String compression and expand string to original string using compress\_string and expand\_string function respectively  \* FM\_index Array is generated using fm\_index function  \* Offset indices are generated using offset function  \* Inverse Burrow Wheelers Transform performed to form the orginal string back   \*/*    public void **createSuffixArray**()   {   for(int index = 0; index < length; index++)   {  String text = "";  for (int text\_index = index; text\_index < length; text\_index++)  {  text+=this.text[text\_index];  }   suffix[index] = text;  }    *//Duplicating suffix array to NwSfx array*  String[] NwSfx=new String[suffix.length];  for (int i=0; i<suffix.length; i++)   NwSfx[i] = suffix[i];    *//Duplicating index array to index1 array*  int[] index1=new int[index.length];  for (int i=0; i<index.length; i++)   index1[i] = index[i];    *// Invoking sortStrings function and store the values to sorted\_sfx\_arrs array*  String[] sorted\_sfx\_arrs = sortStrings(NwSfx, index);       *// Invoking BWT\_suffixarray function and store the values to bwt array*    *// long startTime = System.nanoTime();*  String[] bwt = BWT\_suffixarray(new\_text,index);   *// long stopTime = System.nanoTime();*      *// A bufferString is created to store the values in bwt array to a string*  StringBuffer sb = new StringBuffer();  for(int i = 0; i < length; i++)   {  sb.append(bwt[i]);  }    String str = sb.toString();    *//Duplicating bwt array to BWT\_fmindex array*  String[] BWT\_fmindex = new String[bwt.length];  for (int i=0; i<bwt.length; i++)   BWT\_fmindex[i] = bwt[i];    *//Duplicating bwt array to BWT\_offset array*  String[] BWT\_offset = new String[bwt.length];  for (int i=0; i<bwt.length; i++)   BWT\_offset[i] = bwt[i];   *//Duplicating bwt array to BWT\_exactmatch array*  String[] BWT\_exactmatch = new String[bwt.length];  for (int i=0; i<bwt.length; i++)   BWT\_exactmatch[i] = bwt[i];      *// Invoking compress\_string function and store the values to c string*   String c = compress\_string(new\_text1);    *// Invoking expand\_string function and store the values to b string*  String b = expand\_string(c);    *// Invoking offset function and store the values to offset array*  int[] offset = offset(BWT\_offset);    *// Invoking FM\_indx function and store the values to fm\_index array*  int[][] fm\_index= FM\_indx(BWT\_fmindex);    *// Invoking ExactMatch function and store the values to p array*  int[] p = ExactMatch("na",BWT\_exactmatch);      *// PRINTING THE OUTPUTS FROM THE FUNCTIONS*    System.out.println("\nBWT of the given String:");  for(int i = 0;i<length;i++)  {  System.out.print(bwt[i]);  }  *//System.out.println("bwt:");*  *// System.out.print(stopTime - startTime+" nanosecond");*    System.out.println("\n");  System.out.println("-----------------------------------------------------------------------------");      System.out.println("SUFFIX \t INDEX");  for (int iterate = 0; iterate < length; iterate++)  {   System.out.println(suffix[iterate] + "\t" + index1[iterate]);  }  System.out.println("\n");    System.out.println("-----------------------------------------------------------------------------");        System.out.println("Sorted Suffix Array:\t");  for (int iterate = 0; iterate < length; iterate++)  {  System.out.println(sorted\_sfx\_arrs[iterate]);  }  System.out.println("\n");    System.out.println("-----------------------------------------------------------------------------");    System.out.println("\nCompress\_string: \t"+c);  System.out.println("\n\n");      System.out.println("\nExpand\_string: \t"+b);  System.out.println("\n\n");     System.out.println("-----------------------------------------------------------------------------");      System.out.println("Offset: \n");  for(int i=0;i<offset.length ;i++)  {  System.out.print(offset[i]+"\t");  }  System.out.println("\n\n");        System.out.println("FM Index:\n");  for(int i = 0; i < fm\_index.length; i++)   {  for(int j=0;j<unq.length;j++)   {  System.out.print(fm\_index[i][j]);  }  System.out.println();  }  System.out.println("\n\n");        System.out.println("\n ExactMatch: \n");  for(int i = 0; i < p.length; i++)   {  System.out.println(p[i]);  }  System.out.println("\n\n");    System.out.println("-----------------------------------------------------------------------------");    *//Invoking inv\_BWT function to perform INVERSE BURROW WHEELERS TRANSFORM*  *// long startTime1 = System.nanoTime();*  inv\_BWT(str);   *// long stopTime1 = System.nanoTime();*  *// System.out.println("ibwt");*  *// System.out.print(stopTime1 - startTime1+" nanosecond");*        }    */\* Burrows - Wheeler Transform Suffix Array  \* Takes suffix array and the index array as arguments  \* returns the Burrows - Wheeler Transform of given text  \*   \* \*/*   public String[] BWT\_suffixarray(String[] input\_text, int[] index1)  {  *//Function to find the Burrow Wheelers Transform*   *// bwt\_arr string is assigned with the size of input text length*  String[] bwt\_arr= new String[input\_text.length];  for(int i = 0; i<length;i++)  {  int j = index1[i] -1;  if(j<0)  j=j+length;  bwt\_arr[i] = input\_text[j];  }  *//last term from each string in the rotated array is returned*  return bwt\_arr;  }    */\*  \* Sort Strings  \* Takes an Array String and index array  \* lexicographically sort array and their respective sorted index array  \* Returns lexicographically sorted array  \*/*  public static String[] sortStrings(String[] arr, int[] indx)   {     String temp;   int temp1;      *// Sorting strings using bubble sort*   for (int i = 0; i < arr.length - 1; i++)   {   for (int j = 0; j < arr.length-1; j++)   {   if (arr[j].compareTo(arr[j+1]) > 0)   {   temp = arr[j];   arr[j] = arr[j+1];   arr[j+1] = temp;  temp1 = indx[j];  indx[j] = indx[j+1];  indx[j+1] = temp1;  }   }   }    return arr;   }    */\*  \* Compress Strings  \* Takes A string to compress as argument  \* returns compressed string  \*/*  public String **compress\_string**(String str)  {  *//function to compress the string*  *// string c\_str is created to store the compressed string*  String c\_str = "";  int i = 0;  *// iterating through each element in the string*  *// if the current element and the next element,*   *// then there will be increment in count*  while(i<str.length())  {  int count =1;  char ch = str.charAt(i);    int j=i;  while(j<str.length()-1)  {    if (str.charAt(j) == str.charAt(j+1))  {   count++;  j++;  }  else  break;  }  *// the element and repetition of the term*  *// will be appended to the string*   c\_str = c\_str + ch+ count;  i = j+1;  }  return c\_str;  }    */\*  \* Expand Strings  \* Takes a compressed String as argument  \* Returns the original string  \*/*  public String **expand\_string**(String str)  {  */\* function to expand the compressed   string to form original string  \*/*  String dcomp = " ";  str = str+"j";  String s ="";  */\* first it will check for the integer after the term  then it will iterate for that much length to   append the same element to the string   \*/*  for(int i=0;i<str.length();i++)  {  int n=0;  if(Character.isDigit(str.charAt(i)))  {  n = str.charAt(i) - 48;  }  for(int j=0; j<n;j++)  {  s = s+str.charAt(i-1);  }  }  dcomp = dcomp+s;   return dcomp;  }   */\*  \* Search Digit  \* Takes a String as input  \* Returns the position of digit or false message if no digit present  \*/*  public static int **search\_digit**(String s)   {     *// Function to check if is digit*   *// is found or not*   for (int i = 0; i < s.length(); i++) {   if (Character.isDigit(   s.charAt(i))   == true) {   *// return position of digit*   return i + 1;   }   }   *// return 0 if digit not present*   return 0;   }   */\*  \* Bubble Sort  \* Takes A String as Argument  \* Returns Bubble Sorted String  \*/*  public static String[] bub\_sort(String[] arr)    {     String temp;       *// Sorting strings using bubble sort*   for (int i = 0; i < arr.length - 1; i++)   {   for (int j = 0; j < arr.length-1; j++)   {   if (arr[j].compareTo(arr[j+1]) > 0)   {   temp = arr[j];   arr[j] = arr[j+1];   arr[j+1] = temp;  }   }   }  return arr;   }   */\*  \* Offset  \* Takes Bwt string as input  \* Returns offset array  \*/*  public int[] offset(String[] lc)  {  *//function to find the offset of bwt string*  int[] o= new int[lc.length];  o[0] = 0;  String[] st = bub\_sort(lc);  int index1 = 1;  int x; boolean y;  *//first index coming for each term*  for(int i=0;i<length;i++)  {  *//find the first index of the string in the string array*  x = findIndex(st,st[i]);  *//find if x exist in the integer array*  y = ifee(o,x);  *//if false then append to the integer array*  if(!y)  {  o[index1] = x;  index1++;  }    }  *// offset array is created to store the offset index*  *// size of offset array is index1*  int[] offset = new int[index1];  for(int j=0;j<index1;j++)  {  offset[j] = o[j];  }    return offset;  }    */\*  \* Find Index  \* Takes a string array and string as input  \* Returns the first index for which the string appears in the string array  \*/*  public static int **findIndex**(String arr[], String t)   {         *// find length of array*   int len = arr.length;   int i = 0;     *// traverse in the array*   for(i=0;i<len;i++)  {  *// if the i-th element is t*   *// then return the index*   if (arr[i].equals(t))   return i;     }   return -1;   }    */\*  \* ifee-Find Index  \* Takes an int array and integer as argument  \* Returns whether the int present in the integer array or not  \*/*  public static boolean **ifee**(int arr[], int t)   {  if (arr == null) {   return false;   }   int len = arr.length;   int i = 0;   while (i < len) {   if (arr[i] == t) {   return true;   }   else {   i = i + 1;   }   }   return false;   }    */\*  \* Exact Match  \* Takes String and bwt string array as argument  \* Returns indices of ExactMatch  \*/*  public int[] ExactMatch(String p, String[] last\_col)  {  *// substring the input string p into an array q*  String[] q = new String[p.length()];     for (int i = 0; i < p.length(); i++)  {  q[i] = p.substring(i, i+1);  }     int[][] F = FM\_indx(last\_col);  int[] O = offset(last\_col);  int lo =0; *// low in fm index*  int hi = F.length-1; *// high in fm index*    *// creating a unique string removing all duplicates from string*  String[] unique =bub\_sort(last\_col);  int[] o= new int[last\_col.length];  o[0] = 0;  int index1 = 1;  int x; boolean y;  for(int i=0;i<length;i++)  {  x = findIndex(unique,unique[i]);  y = ifee(o,x);  if(!y)  {  o[index1] = x;  index1++;  }    }    unq = new String[index1];  for(int i=0;i<index1;i++)  {  int j = o[i];*//value inside the int array is given to j*  *// value in the index j of unique is stored to the ith index of unq*   unq[i] = unique[j];  }      for(int i=0;i<p.length();i++){  int id = findIndex(unq,q[i]);  lo = O[id] + F[lo][id];  hi = O[id] + F[hi][id];    }  *// appending the low and high value to an array arr*  int[] arr = new int[2];   arr[0] = lo;  arr[1] = hi;  return arr;  }    */\*  \* FM index  \* Takes bwt string as input   \* Return FM Index array  \*/*  public int[][] FM\_indx(String[] lc)  {  *//function to find the FM index*   int count = 0;  String[] last\_col=new String[lc.length];  for (int i=0; i<lc.length; i++)   last\_col[i] = lc[i];  String[] unique =bub\_sort(lc);  *// creating a unique array removing duplicate entries*   int[] o= new int[lc.length];  o[0] = 0;  int index1 = 1;  int x; boolean y;  for(int i=0;i<length;i++)  {  x = findIndex(unique,unique[i]);  y = ifee(o,x);  if(!y)  {  o[index1] = x;  index1++;  }    }  unq = new String[index1];  for(int i=0;i<index1;i++)  {  int j = o[i];  unq[i] = unique[j];  }    *//creating a 2d int array, where the size is (bwt\_string\_length x unique\_string\_length)*  *//by default it is intiallised by 0*  int[][] fm\_index= new int[last\_col.length+1][unq.length];    *//if the element in the unique array and bwt array when equals that cell value increment by 1*  for(int j =0;j<unq.length;j++){  for(int i =1;i<fm\_index.length;i++) {  if(unq[j].equals(last\_col[i-1])){  fm\_index[i][j]=count+1;  count++;  }  else{  fm\_index[i][j]=count;  }  }  count = 0;  fm\_index[0][j]=0;  }  return fm\_index;  }    */\*  \* Sort Strings  \* Takes a string as argument  \* Returns the sorted String  \*/*  public static String **sortString1**(String inputString)   {   *// convert input string to char array*   char tempArray[] = inputString.toCharArray();     *// sort tempArray*   Arrays.sort(tempArray);     *// return new sorted string*   return new String(tempArray);   }     */\*inputs the column letter of BWT string as the corresponding row in array  s - is the bwt string  a - is the string array to form which will lead back to original transform  \*/*  public static void **moveFront**(String s, String[] a)  {  for (int i = 0; i < s.length(); i++)  {  a[i] = s.substring(i, i+1) + a[i];  }  }    */\*  \* Inverse BWT  \* Takes Bwt string as input   \* Print the original string and index where the original   was present in the bwt string  \*/*  public static void **inv\_BWT**(String BWTString)  {  int BWTKey = -1;  for(int i=0;i<BWTString.length();i++)  {  if(Character.isDigit(BWTString.charAt(i)))  {  BWTKey = i;  break;  }  }      *// size will be the amount of characters in the input String*   final int SIZE = BWTString.length();   *// creates a new array of strings same size as input String*  String[] revRotations = new String[SIZE];   *// placeholder to remove null from output string*  *// this will make the sorted array into list which will then print*  List<String> remNull;   *// runs once for each stage*  for(int cycles = 0; cycles < BWTString.length(); cycles++)  {  *//string will be put in the front of each row*  moveFront(BWTString, revRotations);  *//sort the array in lexicographical order*  bub\_sort(revRotations);  *// remove null and make the array as list*   remNull = Arrays.asList(revRotations);  *// print each stage*  System.out.println("Stage " + (cycles+1) + "\n");  *// print array in each stage*   for (int i = 0; i < SIZE; i++)  {  String temporary = remNull.get(i);  int temporaryInt = temporary.length()-4;  temporary = temporary.substring(0, temporaryInt);  System.out.println(temporary);  }  System.out.println();  }   System.out.println("INVERSE BWT COMPLETE! \n \n CHECK RESULTS BELOW");  System.out.println("\n");  *// converting to list and then printing a certain item from list*  List<String> invBST = Arrays.asList(revRotations);  String temp = invBST.get(BWTKey);  System.out.println("-----------------------------------------------------------------------------");  System.out.println("AT INDEX NUMBER: " + BWTKey + "");  System.out.println("THE ORIGINAL STRING: " + temp.substring(0, SIZE));  System.out.println("-----------------------------------------------------------------------------");  }    public static void **main**(String[] arg)throws IOException  {  String text = "";  BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));  System.out.println("Enter the Text String ");  text = reader.readLine();    SuffixArray suffixarray = new SuffixArray(text);  suffixarray.createSuffixArray();      }   } |

Output

|  |
| --- |
| Enter the Text String  banana1  BWT of the given String: annb1aa  ----------------------------------------------------------------------------- SUFFIX INDEX banana1 0 anana1 1 nana1 2 ana1 3 na1 4 a1 5 1 6   ----------------------------------------------------------------------------- Sorted Suffix Array:  1 a1 ana1 anana1 banana1 na1 nana1   -----------------------------------------------------------------------------  Compress\_string: b1a1n1a1n1a111     Expand\_string: banana11    ----------------------------------------------------------------------------- Offset:   0 1 4 5    FM Index:  0000 0100 0101 0102 0112 1112 1212 1312      ExactMatch:   2 4    ----------------------------------------------------------------------------- Stage 1  1 a a a b n n  Stage 2  1b a1 an an ba na na  Stage 3  1ba a1b ana ana ban na1 nan  Stage 4  1ban a1ba ana1 anan bana na1b nana  Stage 5  1bana a1ban ana1b anana banan na1ba nana1  Stage 6  1banan a1bana ana1ba anana1 banana na1ban nana1b  Stage 7  1banana a1banan ana1ban anana1b banana1 na1bana nana1ba  INVERSE BWT COMPLETE!     CHECK RESULTS BELOW   ----------------------------------------------------------------------------- AT INDEX NUMBER: 4 THE ORIGINAL STRING: banana1 ----------------------------------------------------------------------------- |

Implementation in R

BWT - Burrow Wheeler Transform

---

```{r}

#given sequence

seq='AGCAGCCCAC'

#seq='banana'

#rotate sequence clockwise

rotate\_seq<-function(seqnce){

rot\_seq <- vector(mode = "list")

#number of charcters in string

N=nchar(seqnce)

s=seqnce

for (i in 1:N){

#placing 1st character at the end of the string

s=paste(substr(s,2,N),substr(s,1,1),sep='')

rot\_seq[i]=s

}

return(rot\_seq)

}

```

```{r}

ans<-rotate\_seq(seq)

ans

```

[[1]]

[1] "GCAGCCCACA"

[[2]]

[1] "CAGCCCACAG"

[[3]]

[1] "AGCCCACAGC"

[[4]]

[1] "GCCCACAGCA"

[[5]]

[1] "CCCACAGCAG"

[[6]]

[1] "CCACAGCAGC"

[[7]]

[1] "CACAGCAGCC"

[[8]]

[1] "ACAGCAGCCC"

[[9]]

[1] "CAGCAGCCCA"

[[10]]

[1] "AGCAGCCCAC"

```{r}

library(plyr)

library(stringi)

bub\_sort <- function(sequence){

for (i in 1:(length(sequence)-1)){

for (j in 1:(length(sequence)-i)){

if (stri\_cmp(sequence[j], sequence[j+1]) > 0){ # Check whether 2 strings appear in Lexicographic Order

temp <- sequence[j] # Swapping of adjacent sequences if they don't appear

sequence[j] <- sequence[j+1] # in lexicographic order

sequence[j+1] <- temp

}

}

}

sequence<- plyr::compact(sequence)

return(sequence)

}

```

```{r}

lex\_seq=bub\_sort(ans)

lex\_seq

```

[[1]]

[1] "ACAGCAGCCC"

[[2]]

[1] "AGCAGCCCAC"

[[3]]

[1] "AGCCCACAGC"

[[4]]

[1] "CACAGCAGCC"

[[5]]

[1] "CAGCAGCCCA"

[[6]]

[1] "CAGCCCACAG"

[[7]]

[1] "CCACAGCAGC"

[[8]]

[1] "CCCACAGCAG"

[[9]]

[1] "GCAGCCCACA"

[[10]]

[1] "GCCCACAGCA"

```{r}

permutation\_string<-function(LexSeq){

st=''

for (i in 1:length(LexSeq)){

#getting last column from rotated seq list

st=paste(st,substring(LexSeq[i],nchar(LexSeq[i]),nchar(LexSeq[i])),sep='')

}

return(st)

}

```

```{r}

#finding the position of original seq from the rotated seq list

string\_match <- function(seq\_list,seqn){

for(i in 1:length(seq\_list)){

k = stri\_cmp(seq\_list[i],seqn)

if(k==0){

strNum = i

}

}

return(strNum)

}

```

```{r}

str\_num=string\_match(lex\_seq,seq)

str\_num

```

[1] 2

```{r}

pt\_str=permutation\_string(lex\_seq)

pt\_str

```

[1] "CCCCAGCGAA"

```{r}

library(tictoc)

#appending permutation string 'n' times and sorting it in #lexicographic order every time you append the string

# n-> length of given string

BWT\_mtrx<-function(Pmtstr,apnd\_seq){

tic("BWT")

N=nchar(Pmtstr)

#apnd\_seq <- vector(mode = "list")

for(i in 1:N){

ss=substr(Pmtstr,i,i)

apnd\_seq[i]=paste(ss,apnd\_seq[i],sep='')

}

#apnd\_seq<-bub\_sort(apnd\_seq)

toc()

return(apnd\_seq)

}

```

```{r}

fin<-BWT\_mtrx(pt\_str,c())

fin1<-BWT\_mtrx(pt\_str,bub\_sort(fin))

fin2<-BWT\_mtrx(pt\_str,bub\_sort(fin1))

fin3<-BWT\_mtrx(pt\_str,bub\_sort(fin2))

fin4<-BWT\_mtrx(pt\_str,bub\_sort(fin3))

fin2

[1] "CACNA" "CAGNA" "CAGNA" "CCANA" "ACANA" "GCANA" "CCC"

[8] "GCCNA" "AGCNA" "AGCNA"

```

```{r}

#decoding the original seq from BWT matrix

inv\_BWT<-function(bub\_str,str\_num){

tic("IBWT")

fin<-BWT\_mtrx(bub\_str,c())

for (i in 1:(nchar(bub\_str)-1)){

fin<-BWT\_mtrx(bub\_str,bub\_sort(fin))

}

#to remove NA into consideration while sorting

fin<-bub\_sort(substr(fin,1,nchar(bub\_str)))

toc()

return(fin[str\_num])

}

```

```{r}

fin\_str<-inv\_BWT(pt\_str,str\_num)

fin\_str

```

"AGCAGCCCAC"

```{r}

suffix\_array <- function (sqnce){

cln\_sq = sqnce

N = nchar(cln\_sq)

sufx\_arr = vector(mode = "list")

S = vector(mode = "list")

for(i in 1:N){

sufx\_arr[i] = substr(cln\_sq,i,nchar(cln\_sq))

}

sorted\_sufx\_arr = bub\_sort(sufx\_arr)

for(k in 1:length(sufx\_arr)){

tmp = sorted\_sufx\_arr[k]

S[k]=string\_match(sufx\_arr,tmp)

}

return(S)

}

```

```{r}

a = suffix\_array('BANANA1')

a

```

[[1]]

[1] 7

[[2]]

[1] 6

[[3]]

[1] 4

[[4]]

[1] 2

[[5]]

[1] 1

[[6]]

[1] 5

[[7]]

[1] 3

```{r}

library(Matrix,'source')

count\_arr <- function(last\_col){

unq = vector(mode = "list")

cnt = 0

unq = unique(last\_col)

unq = bub\_sort(unq)

cnt\_arr = matrix(nrow=length(last\_col)+1,ncol=length(unq))

#cnt\_arr[is.na(cnt\_arr)] = 0

cnt\_arr <- replace(cnt\_arr, is.na(cnt\_arr), 0)

for(j in 1 : length(unq)){

for(i in 2 : dim(cnt\_arr)[1]) {

if(identical(unq[j],last\_col[i-1])){

cnt\_arr[i,j]=cnt+1

cnt = cnt+1

}

else{

cnt\_arr[i,j]=cnt

}

}

cnt = 0

}

return(cnt\_arr)

}

```

```{r}

lc = c('A','N','N','1','B','A','A')

count\_arr(lc)

```

[,1] [,2] [,3] [,4]

[1,] 0 0 0 0

[2,] 0 1 0 0

[3,] 0 1 0 1

[4,] 0 1 0 2

[5,] 1 1 0 2

[6,] 1 1 1 2

[7,] 1 2 1 2

[8,] 1 3 1 2

```{r}

offset <-function(Lc){

O = vector(mode = "list")

st = bub\_sort(Lc)

for(i in 1:length(st)){

n = match(st[i],st)

O[i] = n

}

ofst = unique(O)

return(ofst)

}

```

```{r}

lc = c('A','N','N','1','B','A','A')

offset(lc)

```

[[1]]

[1] 1

[[2]]

[1] 2

[[3]]

[1] 5

[[4]]

[1] 6

```{r}

ExactMatch <- function(p,last\_col){

Fr = count\_arr(last\_col)

O = offset(last\_col)

lo <- 1

hi = dim(Fr)[1]-1

idx = bub\_sort(unique(last\_col))

q1 <- strsplit(p, NULL)[[1]]

qr <- rev(q1)

q = paste(qr, collapse='')

for(i in 1:length(qr)){

id = match(qr[i],idx)

lo = as.numeric(O[id]) + as.numeric(Fr[lo,id])

hi = as.numeric(O[id]) + as.numeric(Fr[hi,id])

}

return(c(lo,hi))

}

```

```{r}

ExactMatch('NA',c('A','N','N','1','B','A','A'))

```

[1] 6 8

```{r}

comp <- function(str) {

c\_str = ""

i = 1

while (i <= nchar(str)) {

count = 1

ch = substr(str, i, i)

j = i

while (j < nchar(str)) {

if (substr(str, j, j) == substr(str, j + 1, j + 1)){

count = count + 1

j = j + 1

}

else{

break

}

}

c\_str = paste(c\_str, ch, count, sep = "")

i = j + 1

}

return (c\_str)

}

```

```{r}

comp("aaabbccdde")

```

[1] "a3b2c2d2e1"

```{r}

# Decompression

decomp <- function(c){

dcomp = ""

s = ""

for (i in 1: nchar(c)){

if (stri\_cmp(substr(c, i, i),substr(c, i + 1, i + 1)) == -1){

n = as.numeric(substr(c, i , i ))

for (j in 1:n) {

s = paste(s, substr(c, i+1, i+1),sep='')

}

}

}

dcomp = paste(dcomp, s)

return(dcomp)

}

```

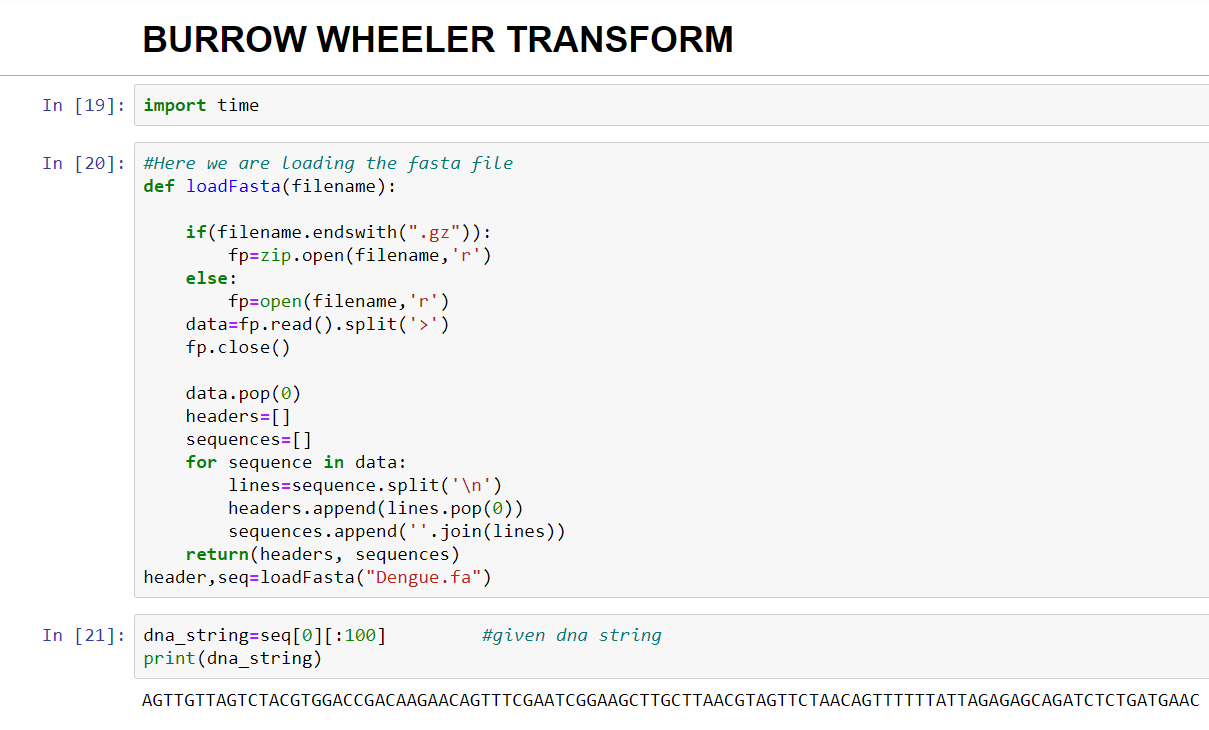
```{r}

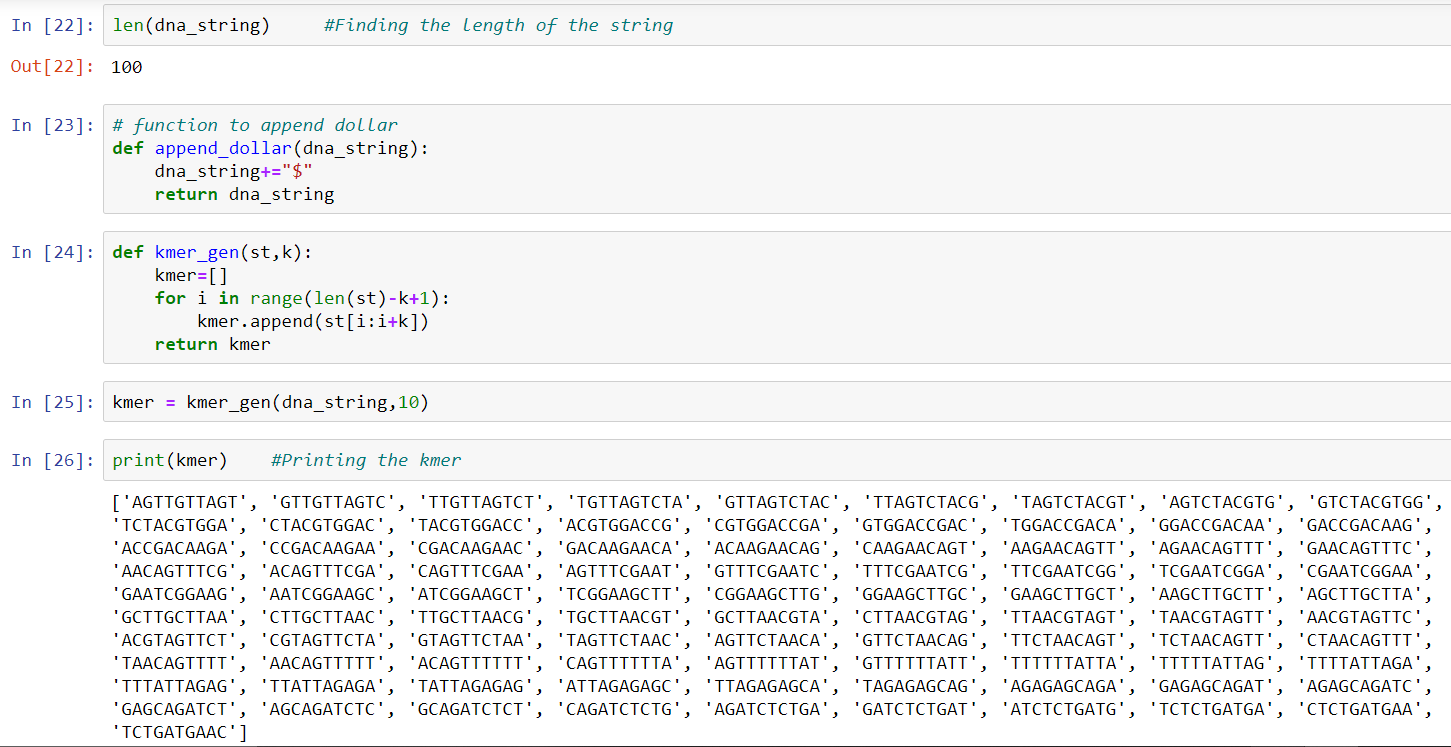
decomp("1a2b3c")

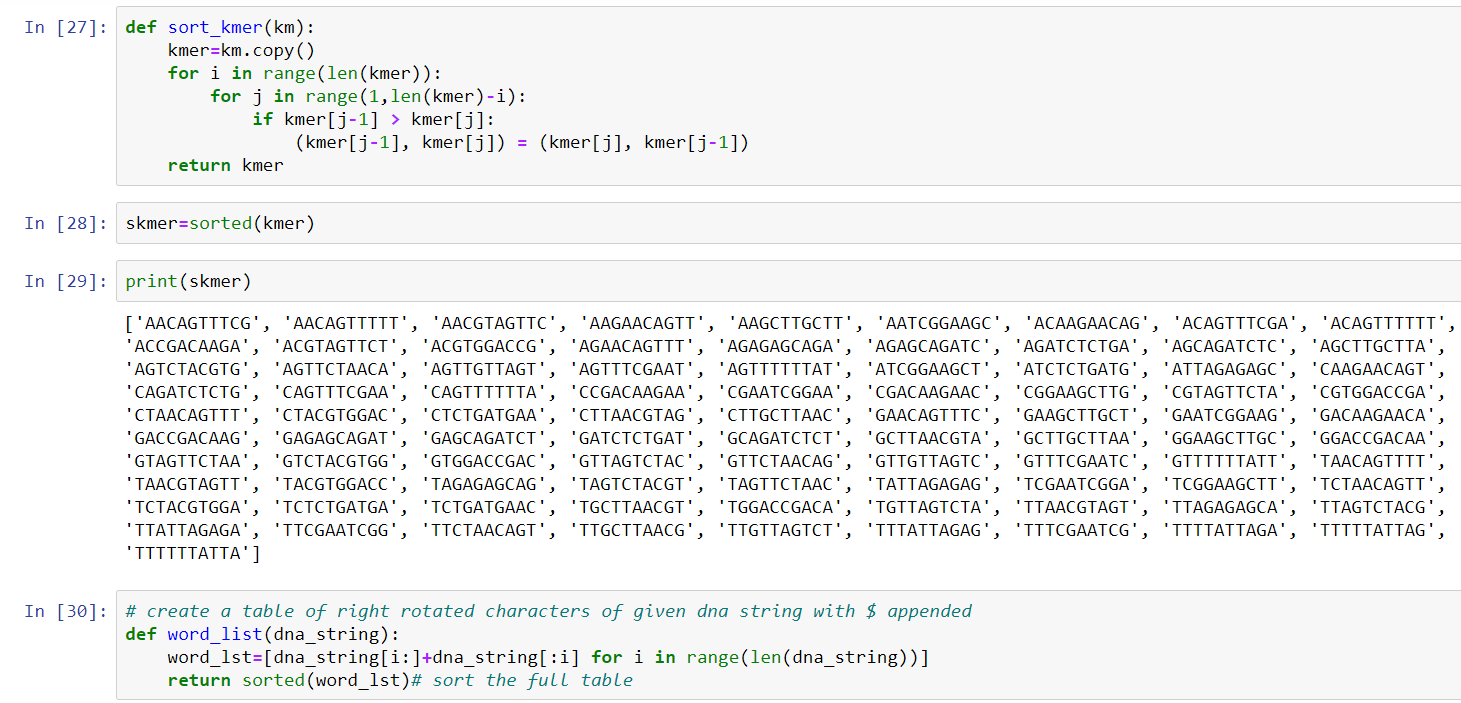
```

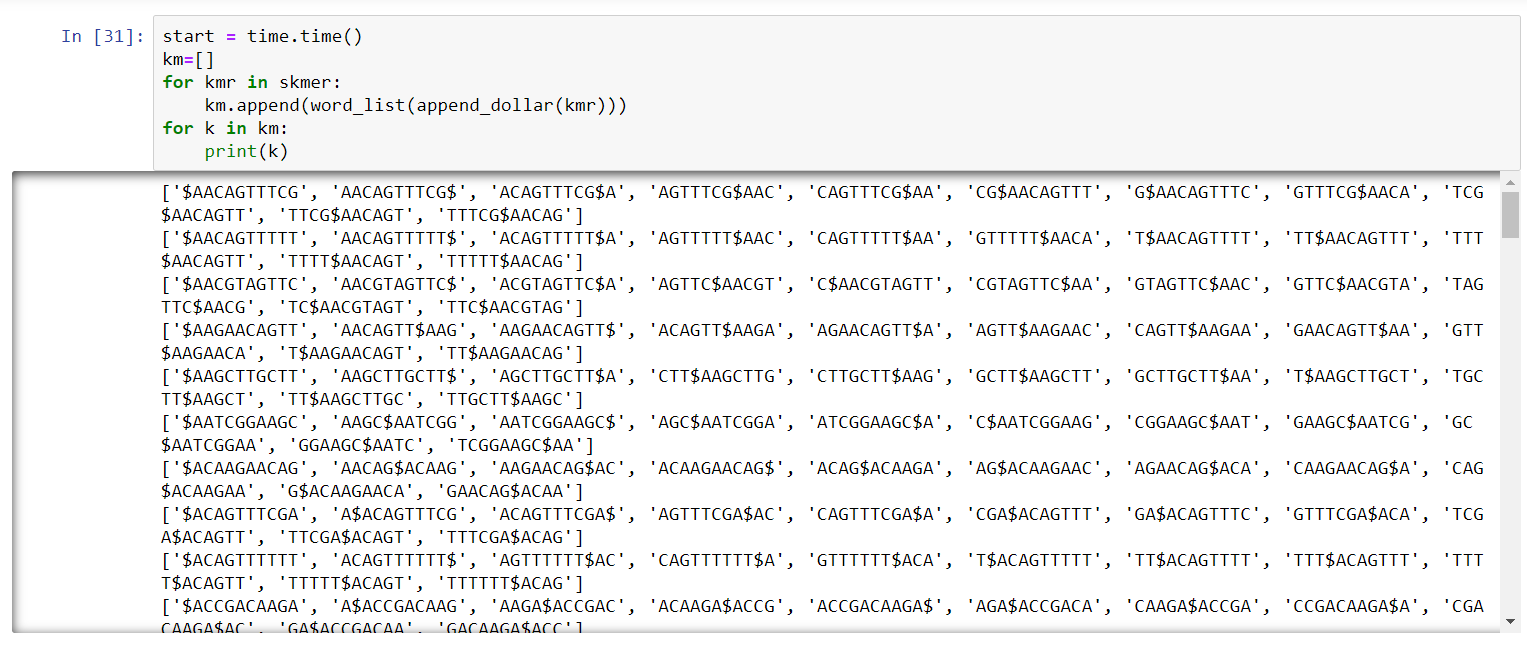
[1] " abbccc"

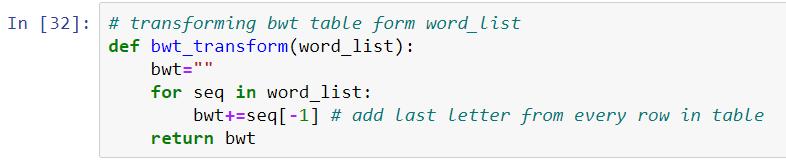
IMPLEMENTATION IN PYTHON

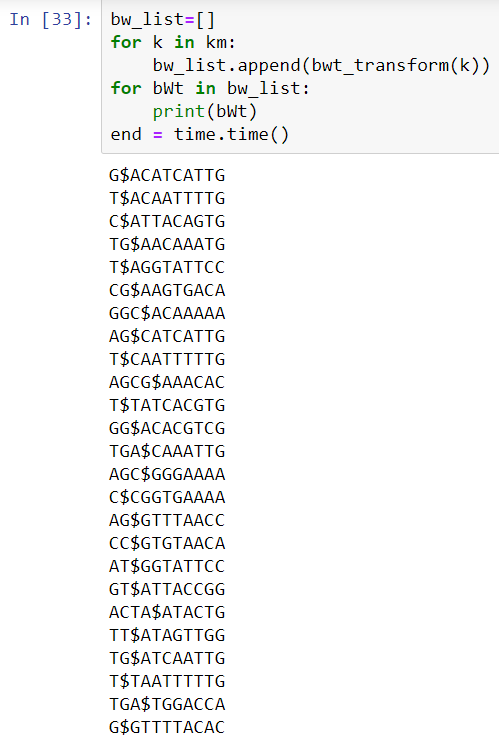


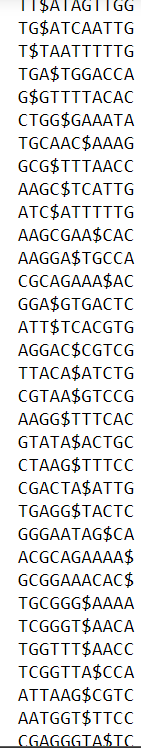


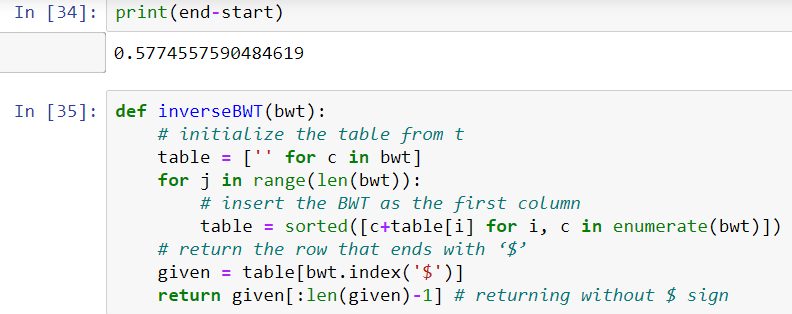






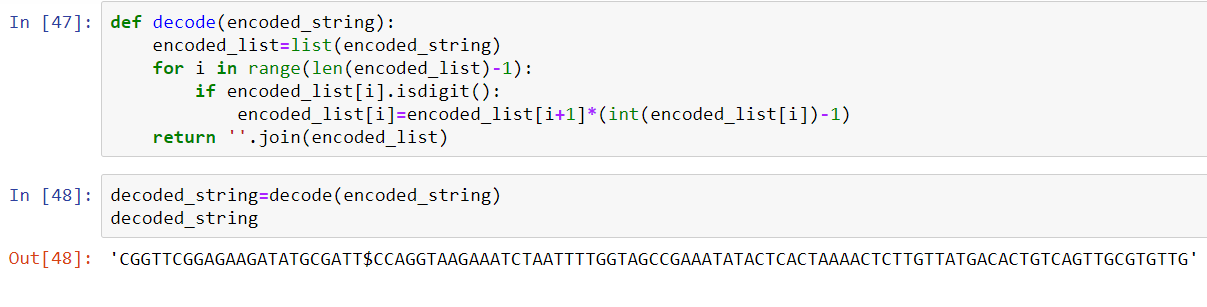


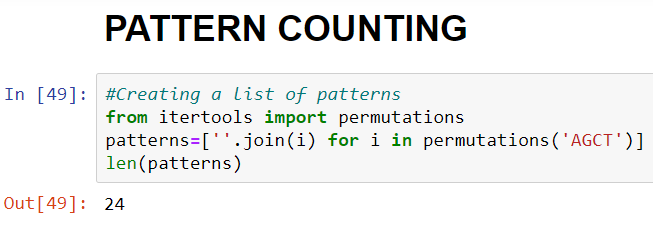


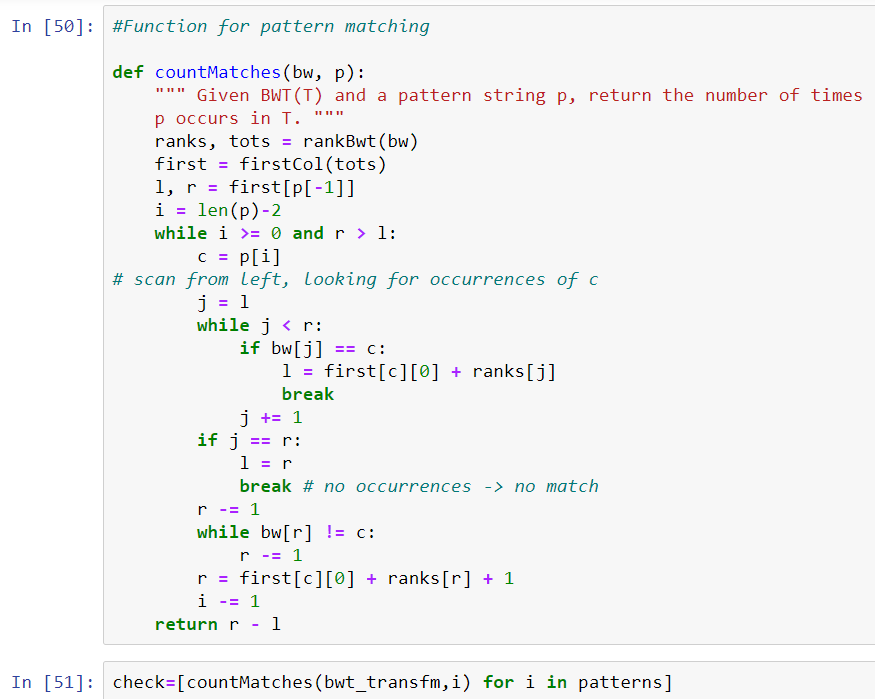


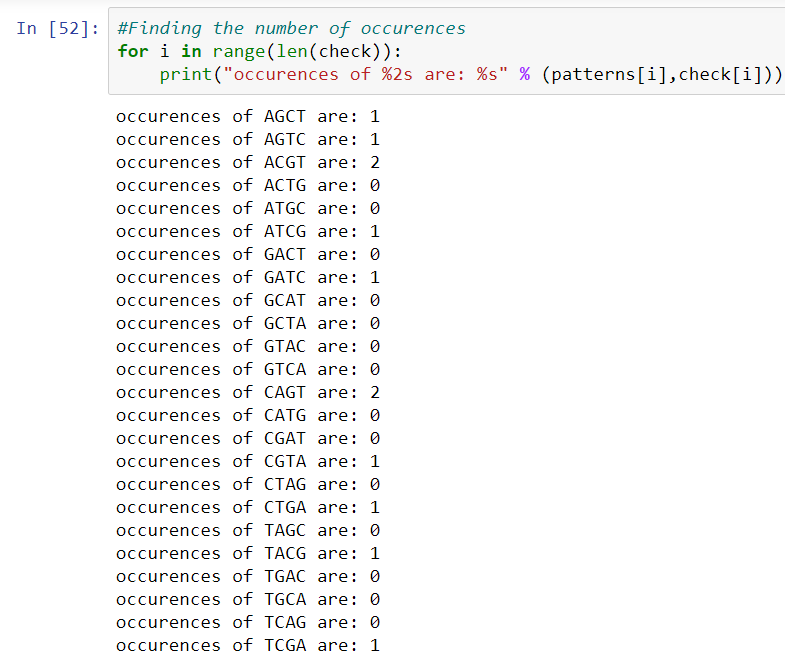












References

<https://www.coursera.org/lecture/algorithms-on-strings/burrows-wheeler-transform-GAA6S>

<https://www.sciencedirect.com/science/article/pii/S1570866713000397>

<https://www.geeksforgeeks.org/burrows-wheeler-data-transform-algorithm/>

<https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/bwt.pdf>

<http://csbio.unc.edu/mcmillan/Comp555S16/Lecture22.pdf>

<https://web.stanford.edu/class/cs262/archives/notes/lecture4.pdf>

<https://www.cs.tufts.edu/~ablumer/BWTSlides.pdf>