Was only able to finish correctness for the Move-To-Front. Updated in this, however the correctness proofs for Circular Suffix Array and Burrows Wheeler are not complete. Can complete at a further date. *This document contains an explaination of algorithmic design and solutions to ... problems. First will be the statement of the question followed by the mathematical formulation, the Brute-Force solution, our solution, proof of correctness for our solution and any slight improvements we could do.*

1 Move-To-Front

This algorithm has two parts: encoding and decoding.

The main idea of move-to-front encoding is to maintain an ordered sequence of the characters in the alphabet, and repeatedly read in a character from the input message, print out the position in which that character appears, and move that character to the front of the sequence. The task is to maintain an ordered sequence of the 256 extended ASCII characters. Initialize the sequence by making the ith character in the sequence equal to the ith extended ASCII character. Now, read in each 8-bit character c from standard input one at a time, output the 8-bit index in the sequence where c appears, and move c to the front.

The main idea of move-to-front decoding is to initialize an ordered sequence of 256 characters, where extended ASCII character i appears ith in the sequence. Now, read in each 8-bit character i (but treat it as an integer between 0 and 255) from standard input one at a time, write the ith character in the sequence, and move that character to the front. Check that the decoder recovers any encoded message.

1.1 Mathematical Formulation

In the Move-To-Front algorithm we will be taking an input of size N characters and either encoding or decoding them using an alphabet (standard ASCI) of size $R = 256$.

1.2 Solution

Move-To-Front *encode*( ) initializes a Linked List of all the asci-characters. Then each character is read in, call it c, from standard input one at a time. The character is then outputed as its correspoding 8-bit index as it appears in the sequence. Once this is done the character is removed from it's current spot and is moved to the front of the Linked List.

Algorithm 1 Move-To-Front Encode

```
procedure ENCODE( )
    alphabet ← Character LinkedList storing all 256 asci-characters
    while there is still input do
        c ← the next character
        index ← the current index of c in the alphabet
        alphabet.REMOVE(index)
        output the 8-bit index in sequence where c appears
        alphabet.ADDFIRST(c)
```

Move-To-Front *decode*( ) initializes a Linked List of all the asci-characters. Then each character is read in, call it index, from standard input one at a time. Then each one is written out as the ith character in the sequence, and move that character to the front.

Algorithm 2 Move-To-Front Decode

```
procedure DECODE( )
    alphabet ← Character LinkedList storing all 256 asci-characters
    while there is still input do
        index ← the index corresponding to stored char
        c ← alphabet.REMOVE(index) // the char from that index
        output c
        alphabet.ADDFIRST(c)
```

1.3 Correctness

Proposition 1. *Our move-to-front encode() and decode() methods will produce the correct outputs required.*

Proof.

Without loss of generality (W.L.O.G.) let us say our alphabet (of length m) is A_0 for the initial state and the string we read in can be represented as s with length n. We will denote the given index where s_i in A_i as A_{s_i} and the resulting alphabet is A_{i+1} where $A_i = A_{i+1} \iff s_i = s_{i-1}$.

We begin by finding the first character s_0 within A_0 which gives us the number A_{s_0} that will be the first to be outputted. We then note that s_0 will be moved to the front, i.e. $A_{s_0} = 0$, giving us the new alphabet A_1. A_1 will be identical to A_0 except all A_{s_k} s.t. $k<s_0$ will be located at A_{s_k+1} and $A_{s_0} = 0$. We can generalize this with changing 0 to i such that for every input s_i, we change from A_i to A_{i+1} s.t. $A_{s_k} = A_{s_k+1}\ \forall\ k<s_i$ and $A_{s_i} = 0 \implies$ producing $A_{i+1} = \{A_{s_i}, A_{s_0}, A_{s_1}, ..., A_{s_i-1}, A_{s_i+1}, A_{s_i+2}, ..., A_{s_{m-1}}\}$. Note here that s_m where $m \neq i$ does not correlate to the string, but the index of a symbol in A_i.

This is the exact process that is asked to execute in the problem description and is executed in both *encode*() and *decode*(). The only difference between the two is: in *encode*() we are taking a sequence of characters, $s = s_0s_1...s_{n-1}$ and outputting the corresponding A_{s_i} for each $s_i \in s$; in *decode*() we are taking a sequence of character codes, $c = c_0c_1...c_{n-1}$ and outputting the character corresponding to the code. i.e. $c_0 = A_{s_0}, c_1 = A_{s_1}, ...c_{n-1} = A_{s_{n-1}}$ and output $= s_0s_1...s_{n-1} = s$ □

Proposition 2. *The move-to-front encode and decode methods will accurately be able to encode and decode the* ***same*** *message.*

Proof.
At both the encoding and decoding stages, we start out with the same unaltered linked list of ASCI characters (represented as ints). As explained above, both will go through the exact same process, however reversing their inputs and outputs. Therefore putting in the input □

1.4 Analysis

For the following analysis, we will say that N is the size of the given input. We also will use $R = 256$ to denote our alphabet size i.e. ASCI characters.

Proposition 3. *The encode() and decode algorithms will both have time and space complexity in proportion to* ***O(R + N)***

Proof. This is because for both, we begin by constructing our alphabet of all 256 Asci Characters **(R)**. Then for each character input (**N** of them):

- **O(1)** get the current index of the character, initially this will be worst case R, however after the first instance is found, the same letter should take a next to constant operation to find.
- **O(1)** write out the number (if encoding) or character (if decoding)
- **O(1)** remove and move character to the front of the alphabet

Now as for the Memory usage, all we use is the character array of the input (**N**), and the linked list of our alphabet (of size **R**)

Giving us a complexity of **O(R + N)** for both time and memory.

□

1.5 An Example

Going through the example of ABRACADABRA! which would be inputted as: ARD!RCAAAABB after going through the Circular Suffix Array. We note that there are only 6 characters: {!, A, B, C, D, R}. I will go through how their ASCI Characters change in the table below as well as their final output.

Encoding													
	Original	Index of Character After Seeing											
Character	Value	A	R	D	!	R	C	A	A	A	A	B	B
!	21	22	23	**24**	00	01	02	03	–	–	–	04	04
A	**41**	00	01	02	03	03	**04**	**00**	**00**	**00**	00	01	01
B	42	–	43	44	–	–	45	–	–	–	**45**	**00**	00
C	43	–	44	45	–	**45**	00	01	–	–	–	02	02
D	44	–	**45**	00	01	02	03	04	–	–	–	05	05
R	52	**52**	00	01	**02**	00	01	02	–	–	–	03	03
Output	**41**	**52**	**45**	**24**	**02**	**45**	**04**	**00**	**00**	**00**	**45**	**00**	–

∴ Our encoded message is: **41 52 45 24 02 45 04 00 00 00 45 00**
To decode this, we will go through the same process except for instead of writting down a number we will be writting down the correspoding character. Therefore we will have the following:

Decoding												
Encoded String:	**41**	**52**	**45**	**24**	**02**	**45**	**04**	**00**	**00**	**00**	**45**	**00**
Characters	Index of Characters											
!	21	22	23	**24**	00	01	02	03	–	–	–	04
A	**41**	00	01	02	03	03	**04**	**00**	**00**	**00**	–	01
B	42	–	43	44	–	–	45	–	–	–	**45**	**00**
C	43	–	44	45	–	**45**	00	01	–	–	–	02
D	44	–	**45**	00	01	02	03	04	–	–	–	05
R	52	**52**	00	01	**02**	00	01	02	–	–	–	03
Output	**A**	**R**	**D**	**!**	**R**	**C**	**A**	**A**	**A**	**A**	**B**	**B**

∴ Our decoded message is: **ARD!RCAAAABB** so we have returned to our original string!

2 Circular Suffix

Given a string, this algorithm constructs a circular suffix array who's ith entry is the index of the original suffix that appears ith in the sorted array. Holistically, we will enumerate every suffix of the word in a circular fashion and sort them, storing the index of where they had originally been. (see example)

2.1 Mathematical Formulation

In the Circular Suffix algorithm we will be taking an input of size N characters and creat an array of size N which will contain the original indicies ($charAt(i)$) of the suffixes in sorted order. For reasons which will become appearent in the analysis section, we will also note here that for the average word we will do C^* compares to determine the relationship between circular suffixes.

2.2 Solution

When we construct the Circular Suffix Array, we will first check to see that the input is valid. If it is then we will continue and make an array of indicies s.t. index[i] = i. From here we will sort these indicies based off of our own comparator. This comparator will use the character array aspect of our input to execute comparisons. Given 2 starting indexes, the algorithm will check to see if the two characters are the same, if they are not then it will return the appropriate negative or positive value. If they are the same, then the two pointers will advance (wrapping around to the front if the pointer should ever leave the string) and repeat. If at the end of all N compares the strings are identical, then the comparator will return 0 $\implies$ equal. At the end of this the object will store the generated sorted circular suffix array so therefore calls on length() and index() will be constant.

Algorithm 3 Circular Suffix Array Construction

```
procedure INIT(String s)
    if s is null then
        throw NullPointerException()
    indicies ← Filled out array [0..(N-1)]
    Arrays.SORT(indicies,comparator()) // see next method
procedure COMPARE(Integer i1, Integer i2)
    for i ∈ {0..(N-1)} do
        (char) c1 ← s[(i1 + i) % s.length]
        (char) c2 ← s[(i2 + i) % s.length]
        if c1 ≠ c2 then
            RETURN(c1-c2)
        RETURN(0)
```

2.3 Correctness

Proposition 4. *The Circular Suffix Array will, given an input of size N, sort the sircular suffixes of the input and return an array of their sorted indexes.*

Proof. To come soon... □

2.4 Analysis

For the following analysis, we will say that N is the size of the given input which implies that we will 'generate' N circular suffixes. We will also note here that for the average word we will do C^* compares. (i.e. there will not be a significant time taken to make several comparisons and will act as the MSD sorting does.)

Proposition 5. *The construction algorithm will have both time complexity in proportion to* $\boldsymbol{O(N{\cdot}log(N){\cdot}C^*)}$ *and space in* $\boldsymbol{O(N)}$.

Proof. We describe each part of the analysis as follows: it will take $\mathbf{O(N{\cdot}log(N))}$ comparisons to sort the N indicies using $Arrays.sort()$. This will cost $\mathbf{O(C^*)}$ per comparison. However we can count this as nearly negligable in the typical case (being words in the english dictionary). However the worst case would be a string that is uniform in character i.e. 'aaaaaa...a' would produce a cost of N every comparison. Though there no such words in the english language.

Now as for the Memory usage, we are only using the indicies array of size N, the original string of size N and two integer pointers in our $compare()$ method. This would give us a grand total of $N + N + 2 = \mathbf{O(N)}$

$$\therefore \text{Giving us a complexity of } \mathbf{O(N{\cdot}log(N){\cdot}C^*)} \text{ for time and } \mathbf{O(N)} \text{ for memory.}$$

□

2.5 An Example

As an example, consider the string "*ABRACADABRA*!" of length 12. The table below shows its 12 circular suffixes and the result of sorting them.

i	Original Suffixes	Sorted Suffixes	index[i]
0	A B R A C A D A B R A !	! A B R A C A D A B R A	11
1	B R A C A D A B R A ! A	A ! A B R A C A D A B R	10
2	R A C A D A B R A ! A B	A B R A ! A B R A C A D	7
3	A C A D A B R A ! A B R	A B R A C A D A B R A !	0
4	C A D A B R A ! A B R A	A C A D A B R A ! A B R	3
5	A D A B R A ! A B R A C	A D A B R A ! A B R A C	5
6	D A B R A ! A B R A C A	B R A ! A B R A C A D A	8
7	A B R A ! A B R A C A D	B R A C A D A B R A ! A	1
8	B R A ! A B R A C A D A	C A D A B R A ! A B R A	4
9	R A ! A B R A C A D A B	D A B R A ! A B R A C A	6
10	A ! A B R A C A D A B R	R A ! A B R A C A D A B	9
11	! A B R A C A D A B R A	R A C A D A B R A ! A B	2

3 Burrows Wheeler Transform

The goal of the Burrows-Wheeler transform is not to compress a message, but rather to transform it into a form that is more amenable to compression. The transform rearranges the characters in the input so that there are lots of clusters with repeated characters, but in such a way that it is still possible to recover the original input. It relies on the following intuition: if you see the letters hen in English text, then most of the time the letter preceding it is t or w. If you could somehow group all such preceding letters together (mostly t's and some w's), then you would have an easy opportunity for data compression. Our algorithm for Burrows Wheeler performs two actions, transforming a given string to be encoded and inverse transforming a given string to turn it back into its original form.

3.1 Mathematical Formulation

We take in an input of size N and use an alphabet of size R to either transform or reverse transform it.

3.2 Solution

The transform function of the Burrows Wheeler Transform program first creates a Circular Suffix Array of the given string. Then we record the index of the first original suffix (the unshifted original string). We then iterate through the CSA to record the index of the last character in each sorted suffix. These indices are then used to print out the character at each index of the original string.

Algorithm 4 Burrows Wheeler Transform

```
procedure TRANSFORM( )
    originalWord ← new StringBuilder
    while BinaryStdIn is not empty do
        originalWord ← BinaryStdIn.READCHAR( )
    csa ← new CircularSuffixArray built from originalWord
    for element ∈ csa do
        if csa.INDEXAT(index) == 0 then
            first = index
            BinaryStdOut .WRITE(first)
            break
    for element ∈ csa do
        if csa.index(i) == 0 then
            index ← csa length-1; // index of last character
        else
            index ← csa index at i – 1
        Write originalWord character at index
```

To perform the inverse transformation, we read from Binary standard input the encoded string and initialize an array, t, to the characters corresponding to the last character in each sorted suffix that will be used to reconstruct the string. The t array is then copied and the copy is sorted using bucket sort. We now have the first characters of the sorted suffix array along with the last characters.

Algorithm 5 Burrows Wheeler Inverse Transform

```
procedure INVERSE( )
    first ← BinaryStdIn.READINT( )
    Char[]t ← coded message from BinaryStdIn
    length ← t.LENGTH( )
    Char[]sorted ← bucket sort version of t
    next ← getNextt, length //see below
    for element ∈ sorted do
        BinaryStdOut.WRITE(sorted[index])
        index ← next[index]
```

From here we reconstruct the original string by using a variation of bucket sorting the characters in t. This gives us the *next* array in which will we define next[i] to be the row in the sorted order where the (j + 1)st original suffix appears. By moving through *next* and taking the next[the previous next] we are able to reconstruct the original string.

Algorithm 6 getNext

```
procedure GETNEXT(t, length)
    next ← int[]
    buckets ← LinkedList <Integer >[number of ACSI chars]
    buckets ← chars in t
    for c <number of ACSI chars and i <length do
        while i <length & buckets is not empty do
            next[i] = buckets[c].REMOVE(first)
```

3.3 Correctness

Proposition 6. *The...will...*

Proof. Since... □

3.4 Analysis

We take in an input of size N and use an alphabet of size R to either transform or reverse transform it.

Proposition 7. *The $transform()$ algorithm has Time complexity of O(N·log(N)·C*) and Space of O(N)*

Proof. This is the case since in $transform()$ we read in the string (N), build a CircularSuffixArray (N·log(N)·C*, from above) and output the index of every character in the array (N). The space will be N because all we use is an array to store the string.

Giving us an overarching **O(N·log(N)·C*)** complexity and space of **O(N)**.

□

Proposition 8. *The $inverse()$ transform algorithm has a time complexity of* $O(5N + 2R)$

Proof. This is the case since in $inverse()$ transform, we create an array of size N three times (3N) and make a call to getNext and sortArray. sortArray uses bucket sort to sort the chars and thusly has a time complexity of N + R, where R is the size of the alphabet. getNext uses a variation on bucket sort and has a time complexity of N + R. This gives inverse transform a time complexity of (5N + 2R)

Giving us an overarching **O(N + R)** complexity.

Note: the space complexity will be the same. □

3.5 An Example

As our example, we will begin by encoding the string $ABRACADABRA$!. This is done using the circular suffix array, explained on a previous page, and will produce the index[ ] which signifies the original index of the now sorted suffix. As you can see below, the last letter in each sorted suffix has been **bolded**; these are the letters that we will be outputting. These will be preceeded with the site of the original string, i.e. row $i = 3$

i	Original Suffixes	Sorted Suffixes	index[i]
0	A B R A C A D A B R A !	! A B R A C A D A B R **A**	11
1	B R A C A D A B R A ! A	A ! A B R A C A D A B **R**	10
2	R A C A D A B R A ! A B	A B R A ! A B R A C A **D**	7
3	A C A D A B R A ! A B R	A B R A C A D A B R A **!**	**0**
4	C A D A B R A ! A B R A	A C A D A B R A ! A B **R**	3
5	A D A B R A ! A B R A C	A D A B R A ! A B R A **C**	5
6	D A B R A ! A B R A C A	B R A ! A B R A C A D **A**	8
7	A B R A ! A B R A C A D	B R A C A D A B R A ! **A**	1
8	B R A ! A B R A C A D A	C A D A B R A ! A B R **A**	4
9	R A ! A B R A C A D A B	D A B R A ! A B R A C **A**	6
10	A ! A B R A C A D A B R	R A ! A B R A C A D A **B**	9
11	! A B R A C A D A B R A	R A C A D A B R A ! A **B**	2

The output from this then will the sequence:
00 00 00 03 A R D ! R C A A A A B B

The next stage then is the inverse transform portion where we will be given this same sequence and parse it so that way we note that first = 3 and say an array, t, is [41 52 44 21 52 43 41 41 41 41 42 42] (this corresponds to the unicode value of each letter). We build a next[ ] which will represent which letter comes next by recalling three main things:

1. We have all of the letters so can get the corresponding first letters to each last letter (t[i])

2. These are circular suffixes

3. If sorted row i and j both start with the same character then
   $i < j \implies next[i] < next[j]$

By using these three simple rules, we can deduce that the first time that you see the '!' in t, its correspoding *next* value must be the suffix for which you find the first '!' in the sorted list. Now using this simple notion combined with **3**, we see that first instance of A in t will be the next for the first occurance of A in the sorted and the second for the second and so on.
Our next[ ] will be: **[3 0 6 7 8 9 10 11 5 2 1 4]**

Since we now have a fully filled out next array, all we must do is call next[first] to get the second letter and next[next[first]] for the third and so on. Calling write() on each step will then give us the output: **ABRACADABRA!**

4 Princeton Readme

4.1 Question 1:

Question: List in table format which input files you used to test your program. Fill in columns for how long your program takes to compress and decompress these instances (by applying BurrowsWheeler, MoveToFront, and Huffman in succession). Also, fill in the third column for the compression ratio (number of bytes in compressed message divided by the number of bytes in the message).

Answer: Timing is done in seconds.

File	Encoding Time	Decoding Time	Compression ratio
bible.txt	15.570	5.218	26.05%
muchado.txt	1.218	0.519	35.37%
chromosome11-human.txt	33.146	8.979	27.88%
world192.txt	9.512	5.024	24.44%
mobydick.txt	3.928	2.799	34.74%
moby1.txt	0.359	0.317	45.81%
sedgewick-algc.txt	0.658	0.427	25.15%
pi-1million.txt	2.917	1.049	43.73%
pi-10million.txt	42.474	16.506	43.74%

4.2 Question 2:

Question: Compare the results of your program (compression ratio and running time) on mobydick.txt to that of the most popular Windows compression program (pkzip) or the most popular Unix/Mac one (gzip). If you don't have pkzip, use 7zip and compress using zip format.

Answer:

- original: 9,531,704 bits
- gzip: 3,884,488 bits $\implies$ 40.75% compression rate
- ours: 3,311,696 bits $\implies$ 34.74% compression rate

$\therefore$ Ours compresses 6.01% more than theirs.

4.3 Question 3:

Question: Give the order of growth of the running time of each of the 6 methods as a function of the input size N and the alphabet size R both in practice (on typical English text inputs) and in theory (in the worst case), e.g., N, N + R, N log(N), N^2, or R N. Include the time for sorting circular suffixes in the Burrows-Wheeler encoder.

Answer:

Class	Method	Typical	Worst
BurrowsWheeler	$transform()$	N·log(N)	N^2·log(N)
BurrowsWheeler	$inverseTransform()$	N + R	N·R
MoveToFront	$encode()$	N + R	N·R
MoveToFront	$decode()$	N + R	N·R
Huffman	$compress()$	N + R·log(R)	N + R·log(R)
Huffman	$expand()$	N	N

4.4 Question 4:

Known bugs / limitations

- We are too awesome, truely limits us.
- Our Circular Suffix Array is not very efficient with corner cases where the input has long sections of repetition i.e. "aaa...a"
- Cannot compress dickens.txt (will run out of memory)
  Note: this may be due to cpu, not algorithm.

4.5 Question 5:

Help Received
Professor Han helped with asking questions on how to design the BurrowsWheeler $inverseTransform()$ method as well as answering clarifying questions on how to run some testing scripts and formulate proofs for correctness. TA **Sam Kolvaka** gave us help for using System.err.println(); for testing purposes so as not to mess up the BinaryStdIn.getchar() function.

4.6 Question 6:

Serious Problems Encountered
Our main issues came with learning how to build the coding environment and learning how to use the packages. The next hurdle was how to comprehend all of the abstract

concepts that are associated with this project, namely the $inverseTransform()$ portion of the BurrowsWheeler class.

4.7 Question 7:

Question: If you worked with a partner, assert below that you followed the protocol as described on the assignment page. Give one sentence explaining what each of you contributed.
Answer:
We worked on this project together and split up the work when it came time to writing up this readme.