Implement the *Burrows–Wheeler data compression algorithm*. This revolutionary algorithm outcompresses *gzip* and *PKZIP*, is relatively easy to implement, and is not protected by any patents. It forms the basis of the Unix compression utility bzip2.

The Burrows–Wheeler data compression algorithm consists of three algorithmic components, which are applied in succession:

1. *Burrows–Wheeler transform.* Given a typical English text file, transform it into a text file in which sequences of the same character occur near each other many times.
2. *Move-to-front encoding.* Given a text file in which sequences of the same character occur near each other many times, convert it into a text file in which certain characters appear much more frequently than others.
3. *Huffman compression.* Given a text file in which certain characters appear much more frequently than others, compress it by encoding frequently occurring characters with short codewords and infrequently occurring characters with long codewords.

Step 3 is the only one that compresses the message; it is particularly effective because Steps 1 and 2 produce a text file in which certain characters appear much more frequently than others. To expand a message, apply the inverse operation in reverse order: first apply the Huffman expansion, then the move-to-front decoding, and finally the inverse Burrows–Wheeler transform. Your task is to implement the Burrows–Wheeler and move-to-front components.

Binary input and binary output. To enable your programs to work with binary data, use BinaryStdIn and BinaryStdOut, which are described in *Algorithms, 4th edition*. You can use HexDump, to display the binary output when debugging: it takes a command-line argument *n* reads bytes from standard input; and writes them to standard output in hexadecimal, *n* per line.

```
~/Desktop/burrows> cat abra.txt
ABRACADABRA!

~/Desktop/burrows> java-algs4 edu.princeton.cs.algs4.HexDump < abra.txt
41 42 52 41 43 41 44 41 42 52 41 21
96 bits
```

Note that in ASCII, '`A`' is 41 (hex), '`B`' is 42 (hex), and '`!`' is 21 (hex).

Huffman compression and expansion. Huffman (Program 5.10 in *Algorithms, 4th edition*) implements the classic Huffman compression and expansion algorithms.

```
~/Desktop/burrows> java-algs4 edu.princeton.cs.algs4.Huffman - < abra.txt | java-algs4 edu.princeton.cs.algs4.HexDump 16
50 4a 22 43 43 56 a8 40 00 00 01 8f 96 8f 94
128 bits
```

```
~/Desktop/burrows> java-algs4 edu.princeton.cs.algs4.Huffman - < abra.txt | java-algs4 edu.princeton.cs.algs4.Huffman +
ABRACADABRA!
```

Do *not* write any code for this step.

Move-to-front encoding and decoding. The main idea of *move-to-front* encoding is to maintain an ordered sequence of the characters in the alphabet by repeatedly reading a character from the input message; printing the position in the sequence in which that character appears; and moving that character to the front of the sequence. As a simple example, if the initial ordering over a 6-character alphabet is `A B C D E F`, and we want to encode the input `CAAABCCCACFF`, then we would update the move-to-front sequence as follows:

move-to-front	in	out
A B C D E F	C	2
C A B D E F	A	1
A C B D E F	A	0
A C B D E F	A	0
A C B D E F	B	2
B A C D E F	C	2
C B A D E F	C	0
C B A D E F	C	0
C B A D E F	A	2
A C B D E F	C	1
C A B D E F	C	0
C A B D E F	F	5
F C A B D E		

If equal characters occur near each other many times in the input, then many of the output values will be small integers (such as 0, 1, and 2). The resulting high frequency of certain characters (0s, 1s, and 2s) provides exactly the kind of input for which Huffman coding achieves favorable compression ratios.

- *Move-to-front encoding.* Your task is to maintain an ordered sequence of the 256 extended ASCII characters. Initialize the sequence by making the *i*th character in the sequence equal to the *i*th extended ASCII character. Now, read 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.

```
~/Desktop/burrows> java-algs4 MoveToFront - < abra.txt | java-algs4 edu.princeton.cs.algs4.HexDump 16
41 42 52 02 44 01 45 01 04 04 02 26
96 bits
```

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

```
~/Desktop/burrows> java-algs4 MoveToFront - < abra.txt | java-algs4 MoveToFront +
ABRACADABRA!
```

Name your program `MoveToFront.java` and organize it using the following API:

```
public class MoveToFront {

   // apply move-to-front encoding, reading from standard input and writing to standard output
   public static void encode()

   // apply move-to-front decoding, reading from standard input and writing to standard output
   public static void decode()

   // if args[0] is "-", apply move-to-front encoding
   // if args[0] is "+", apply move-to-front decoding
   public static void main(String[] args)

}
```

Performance requirements. The running time of both move-to-front encoding and decoding must be proportional to $n \, R$ (or better) in the worst case and proportional to $n + R$ (or better) on inputs that arise when compressing typical English text, where n is the number of characters in the input and R is the alphabet size. The amount of memory used by both move-to-front encoding and decoding must be proportional to $n + R$ (or better) in the worst case.

Circular suffix array. To efficiently implement the key component in the Burrows–Wheeler transform, you will use a fundamental data structure known as the *circular suffix array*, which describes the abstraction of a sorted array of the *n* circular suffixes of a string of length *n*. 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	ABRACADABRA!	!ABRACADABRA	11
1	BRACADABRA!A	A!ABRACADABR	10
2	RACADABRA!AB	ABRA!ABRACAD	7
3	ACADABRA!ABR	ABRACADABRA!	0
4	CADABRA!ABRA	ACADABRA!ABR	3
5	ADABRA!ABRAC	ADABRA!ABRAC	5
6	DABRA!ABRACA	BRA!ABRACADA	8
7	ABRA!ABRACAD	BRACADABRA!A	1
8	BRA!ABRACADA	CADABRA!ABRA	4
9	RA!ABRACADAB	DABRA!ABRACA	6
10	A!ABRACADABR	RA!ABRACADAB	9
11	!ABRACADABRA	RACADABRA!AB	2

We define `index[i]` to be the index of the original suffix that appears *i*th in the sorted array. For example, `index[11]` = `2` means that the *2nd* original suffix appears 11*th* in the sorted order (i.e., last alphabetically).

Your job is to implement the following circular suffix array API, which provides the client access to the `index[]` values:

```
public class CircularSuffixArray {

   // circular suffix array of s
   public CircularSuffixArray(String s)

   // length of s
   public int length()

   // returns index of ith sorted suffix
   public int index(int i)

   // unit testing (required)
   public static void main(String[] args)

}
```

Corner cases. Throw an `IllegalArgumentException` in the constructor if the argument is `null`. Throw an `IllegalArgumentException` in the method `index()` if `i` is outside its prescribed range (between 0 and *n* − 1).

Unit testing. Your `main()` method must call each public method directly and help verify that they work as prescribed (e.g., by printing results to standard output).

Performance requirements. On typical English text, your data type must use space proportional to $n + R$ (or better) and the constructor must take time proportional to $n \log n$ (or better). The methods `length()` and `index()` must take constant time in the worst case.

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 for compression. The Burrows–Wheeler 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 `the` in English text, then, most of the time, the letter preceding it is either `t` or `e`. If you could somehow group all such preceding letters together (mostly `t`s and some `e`s), then you would have a propitious opportunity for data compression.

- *Burrows–Wheeler transform.* The Burrows–Wheeler transform of a string *s* of length *n* is defined as follows: Consider the result of sorting the *n* circular suffixes of *s*. The Burrows–Wheeler transform is the last column in the sorted suffixes array `t[]` preceded by the row number `first` in which the original string ends up. Continuing with the "`ABRACADABRA!`" example above, we highlight the two components of the Burrows–Wheeler transform in the table below.

i	Original Suffixes	Sorted Suffixes	t	index[i]
0	ABRACADABRA!	!ABRACADABRA	11	
1	BRACADABRA!A	A!ABRACADABR	10	
2	RACADABRA!AB	ABRA!ABRACAD	7	
3	ACADABRA!ABR	ABRACADABRA!	*0*	
4	CADABRA!ABRA	ACADABRA!ABR	3	
5	ADABRA!ABRAC	ADABRA!ABRAC	5	
6	DABRA!ABRACA	BRA!ABRACADA	8	
7	ABRA!ABRACAD	BRACADABRA!A	1	
8	BRA!ABRACADA	CADABRA!ABRA	4	
9	RA!ABRACADAB	DABRA!ABRACA	6	
10	A!ABRACADABR	RA!ABRACADAB	9	
11	!ABRACADABRA	RACADABRA!AB	2	

Since the original string `ABRACADABRA!` ends up in row 3, we have `first` = `3`. Thus, the Burrows–Wheeler transform is

```
3
ARD!RCAAAABB
```

Notice how there are 4 consecutive `A`s and 2 consecutive `B`s—these clusters make the message easier to compress.

```
~/Desktop/burrows> java-algs4 BurrowsWheeler - < abra.txt | java-algs4 edu.princeton.cs.algs4.HexDump 16
00 00 00 03 41 52 44 21 52 43 41 41 41 42 42
120 bits
```

Also, note that the integer 3 is represented using 4 bytes (`00 00 00 03`). The character '`A`' is represented by hex `41`, the character '`R`' by `52`, and so forth.

- *Burrows–Wheeler inverse transform.* Now, we invert the Burrows–Wheeler transform and recover the original input string. If the *j*th original suffix (original string, shifted *j* characters to the left) is the *i*th row in the sorted order, we define `next[i]` to be the row in the sorted order where the (*j* + 1)*st* original suffix appears. For example, if `first` is the row in which the original input string appears, then `next[first]` is the row in the sorted order where the 1*st* original suffix (the original string left-shifted by 1) appears; `next[next[first]]` is the row in the sorted order where the *2nd* original suffix appears; `next[next[next[first]]]` is the row where the *3rd* original suffix appears; and so forth.
  - *Inverting the message given t[], first, and the next[] array.* The input to the Burrows–Wheeler decoder is the last column `t[]` of the sorted suffixes along with `first`. From `t[]`, we can deduce the first column of the sorted suffixes because it consists of precisely the same characters, but in sorted order.

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

Now, given the `next[]` array and `first`, we can reconstruct the original input string because the first character of the *i*th original suffix is the *i*th character in the input string. In the example above, since `first` = `3`, we know that the original input string starts with '`A`' (and ends with '`!`'). Since `next[first]` = `7`, the next original suffix appears in row 7; thus, the next character in the original input string is '`B`'. Since `next[next[first]]` = `11`, the next original suffix appears in row 11; thus, the next character in the original input string is '`R`'.

  - *Constructing the next[] array from t[] and first.* Amazingly, the information contained in the Burrows–Wheeler transform suffices to reconstruct the `next[]` array, and, hence, the original message! Here's how. It is easy to deduce a `next[]` value for a character that appears exactly once in the input string. For example, consider the suffix that starts with '`C`'. By inspecting the first column, it appears *8th* in the sorted order. The next original suffix after this one will have the character '`r`' as its last character. By inspecting the last column, the next original appears 5*th* in the sorted order. Thus, `next[8]` = `5`. Similarly, '`D`' and '`!`' each occur only once, so we can deduce that `next[9]` = `2` and `next[0]` = `3`.

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

However, since '`R`' appears twice, it may seem ambiguous whether `next[10]` = `1` and `next[11]` = `4`, or whether `next[10]` = `4` and `next[11]` = `1`. Here's the key rule that resolves the apparent ambiguity:

> *If sorted row i and j both start with the same character and i < j, then next[i] < next[j].*

This rule implies `next[10]` = `1` and `next[11]` = `4`. Why is this rule valid? The rows are sorted, so row 10 is lexicographically less than row 11. Thus, the ten unknown characters in row 10 must be less than the ten unknown characters in row 11 (since both start with '`R`'). We also know that between the two rows that end with '`R`', row 1 is less than row 4. But, the ten unknown characters in row 10 and 11 are precisely the first ten characters in rows 1 and 4. Thus, `next[10]` = `1` and `next[11]` = `4` or this would contradict the fact that the suffixes are sorted.

Check that the inverse transform recovers any transformed message.

```
~/Desktop/burrows> java-algs4 BurrowsWheeler - < abra.txt | java-algs4 BurrowsWheeler +
ABRACADABRA!
```

Name your program `BurrowsWheeler.java` and organize it using the following API:

```
public class BurrowsWheeler {

   // apply Burrows-Wheeler transform,
   // reading from standard input and writing to standard output
   public static void transform()

   // apply Burrows-Wheeler inverse transform,
   // reading from standard input and writing to standard output
   public static void inverseTransform()

   // if args[0] is "-", apply Burrows-Wheeler transform
   // if args[0] is "+", apply Burrows-Wheeler inverse transform
   public static void main(String[] args)

}
```

Performance requirements. The running time of your Burrows–Wheeler transform must be proportional to $n + R$ (or better) in the worst case, excluding the time to construct the circular suffix array. The running time of your Burrows–Wheeler inverse transform must be proportional to $n + R$ (or better) in the worst case. The amount of memory used by both the Burrows–Wheeler transform and inverse transform must be proportional to $n + R$ (or better) in the worst case.

Analysis. Once you have `MoveToFront.java` and `BurrowsWheeler.java` working, compress some text files. Then, test it on some binary files. Calculate the compression ratio achieved for each file and report the time to compress and expand each file. (Here, compression and expansion consists of applying `BurrowsWheeler`, `MoveToFront`, and `Huffman` in succession.) Finally, determine the order of growth of the running time of each of your methods, both in the worst case and on typical English text files.

Web submission. Submit a .zip file containing Submit `MoveToFront.java`, `BurrowsWheeler.java`, `CircularSuffixArray.java`, any other supporting files (excluding `algs4.jar`). You may not call any library functions except those in `java.lang`, `java.util`, and `algs4.jar`.