Project 5 - Data Compression

Handed out: **November 26, 2012**
Due: **December 9, 2012, 11:59pm**

Description

The Burrows-Wheeler data compression algorithm is a revolutionary algorithm that 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.

It consists of three different 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 more frequently than others.
3. **Huffman compression**. Given a text file in which certain characters appear more frequently than others, compress it by encoding frequently occuring characters with short codewords and rare ones with long codewords.

What you need to do...

Out of the 3 components you need to implement the first 2 mentioned and the third one will be provided using the existing code created by the book authors (Huffman.java). You will make use of the usual algs4.jar (Documentation) and stdlib.jar (Documentation).

For the components you need to implement, you are required to submit two main programs called MoveToFront.java and BurrowsWheeler.java, as well as any helper classes used (implementation details below).

To succesfully compress a test file you need to run the 3 components in succession: `BurrowsWheeler`, `MoveToFront` and `Huffman`. This process should generate a compressed file and, in order to recover the original content, you will execute the opposite process: Huffman, MoveToFront and BurrowsWheeler. Parameters provided to each program (described later) will indicate which of the two ways is being requested (compression/decompression).

Testing

Three test examples are provided here. Two of them are text files and one of them is an image file. For each of the test cases, 5 files are provided:

- The original files that need to be compressed with extensions .txt or .gif
- The final result file, with extension .bwt.mtf.huff which is the result of applying all three steps
- Three files that represent the partial results of applying each of the 3 isolated components (the extension represents which one was used)

As a suggestion, if you want to visualize the detailed content of each test file, you can use HexDump.java (described later).
The results generated by your programs should be equal to the tests provided. You can use the command

`diff` file1 file2

to test if they are equal.

Detailed Description

Binary input and binary output. To enable your programs to work with binary data, you will use the libraries BinaryStdIn.java and BinaryStdOut.java (from stdlib). To display the binary output, you can use HexDump.java, which takes a command-line argument *N*, reads bytes from standard input and writes them to standard output in hexadecimal, *N* per line. This is just a helper class to allow you to easily visualize results.

```
% more abra.txt
ABRACADABRA!

% java HexDump 16 < abra.txt
41 42 52 41 43 41 44 41 42 52 41 21
12 bytes
```

Note that 'A' is 41 (hex) in ASCII.

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.

- *Burrows-Wheeler encoding.* Treat the input string as a cyclic string and sort the *N* suffixes of length *N*. Here is how it works for the text message "ABRACADABRA!". The 12 original suffixes are ABRACADABRA!, BRACADABRA!A, ..., !ABRACADABRA, and appear in rows 0 through 11 of the table below. Sorting these 12 strings yields the sorted suffixes. Ignore the next[] array for now - you will only need it for decoding.

```
 i     Original Suffixes          Sorted Suffixes       t     next
--    -----------------------    -----------------------     ----
 0    A B R A C A D A B R A !    ! A B R A C A D A B R A      3
 1    B R A C A D A B R A ! A    A ! A B R A C A D A B R      0
 2    R A C A D A B R A ! A B    A B R A ! A B R A C A D      6
*3    A C A D A B R A ! A B R    A B R A C A D A B R A !      7
 4    C A D A B R A ! A B R A    A C A D A B R A ! A B R      8
 5    A D A B R A ! A B R A C    A D A B R A ! A B R A C      9
 6    D A B R A ! A B R A C A    B R A ! A B R A C A D A     10
 7    A B R A ! A B R A C A D    B R A C A D A B R A ! A     11
 8    B R A ! A B R A C A D A    C A D A B R A ! A B R A      5
 9    R A ! A B R A C A D A B    D A B R A ! A B R A C A      2
10    A ! A B R A C A D A B R    R A ! A B R A C A D A B      1
11    ! A B R A C A D A B R A    R A C A D A B R A ! A B      4
```

  The Burrows Wheeler transform t[] is the last column in the suffix sorted list, preceded by the row number where the original string **ABRACADABRA!** ends up.

```
3
ARD!RCAAAABB
```

  Notice how there are 4 consecutive **A**s and 2 consecutive **B**s - this makes the file easier to compress.

```
% java BurrowsWheeler - < abra.txt | java HexDump 16
00 00 00 03 41 52 44 21 52 43 41 41 41 41 42 42
16 bytes
```

  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 decoder.* Now we describe how to undo the Burrows-Wheeler transform and recover the original message. If the *j*-th original suffix (original string, shifted *j* characters to the left) is the *i*-th row in the sorted order, then next[i] records the row in the sorted order where the *(j+1)*st original suffix appears. For example, the 0th original suffix **ABRACDABRA!** is row 3 of the sorted order; since next[3] = 7, the next original suffix **BRACADABRA!A** is row 7 of the sorted order. Knowing the array next[] makes decoding easy, as with the following Java code:

```
int[] next = { 3, 0, 6, 7, 8, 9, 10, 11, 5, 2, 1, 4 };
int N = next.length;
String t = "ARD!RCAAAABB";
int i = 3;
for (int count = 0; count < N; count++) {
  i = next[i];
  System.out.write(t.charAt(i));
}
```

  Amazingly, the information contained in the Burrows-Wheeler transform is enough to reconstruct next[], and hence the original message! Here's how. First, we know all of the characters in the original message, even if they're permuted in the wrong order. This enables us to reconstruct the first column in the suffix sorted list by sorting the characters. Since 'C' only occurs once in the message and the suffixes are formed using cyclic wrap-around, we can deduce that 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
--    ----------------------        ----
 0    ! ? ? ? ? ? ? ? ? ? ? A         3
 1    A ? ? ? ? ? ? ? ? ? ? R
 2    A ? ? ? ? ? ? ? ? ? ? D
*3    A ? ? ? ? ? ? ? ? ? ? !
 4    A ? ? ? ? ? ? ? ? ? ? R
 5    A ? ? ? ? ? ? ? ? ? ? C
 6    B ? ? ? ? ? ? ? ? ? ? A
 7    B ? ? ? ? ? ? ? ? ? ? A
 8    C ? ? ? ? ? ? ? ? ? ? A         5
 9    D ? ? ? ? ? ? ? ? ? ? A         2
10    R ? ? ? ? ? ? ? ? ? ? B
11    R ? ? ? ? ? ? ? ? ? ? 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 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 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 decoder recovers any encoded message.

```
% java BurrowsWheeler - < abra.txt | java BurrowsWheeler +
ABRACADABRA!
```

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

```
public class BurrowsWheeler {
    // apply Burrows-Wheeler encoding, reading from standard input
    // and writing to standard output
    public static void encode()

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

    // if args[0] is '-', apply Burrows-Wheeler encoding
    // if args[0] is '+', apply Burrows-Wheeler decoding
    public static void main(String[] args)
}
```

Move-to-front encoding and decoding. The main idea of *move-to-front* encoding is to maintain an ordered sequence of all of the characters that can appear in a message (e.g. all ASCII characters), and repeatedly read in characters from the input message, print out the position in which that character appears, and move that character to the front. Note, that the *move-to-front* action refers to the reordering of the sequence of valid characters (not a reordering of the message). 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 **CAAABCCCACCF**, then we would update the move-to-front sequences 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
```

For example, at step number 5 the program will read a B. Since at that moment B is the third element from the left, it has index 2 (starting from a zero index) so the out value corresponds to 2. If the same character occurs next to each other many times in the input, then many of the output values will be small integers, such as 0, 1, and 2. The extremely high frequency of certain characters makes an ideal scenario for Huffman coding.

For implementation purposes

`java MoveToFront - < somefile`

indicates we are in the process of compressing a file, while

`java MoveToFront + < somefile`

indicates we are decompressing.

- *Move-to-front encoding.* Your task is to maintain an ordered sequence of the 256 extended ASCII characters (with actual values from 0 to 255). Upper case letters like A,B,C... have ASCII values 65, 66, 67... These correspond to hexadecimal values of 41, 42 and 43. (This is a list of the ASCII values). In the previous example we provided a list of just 6 characters, however, for the assignment you need to initialize a list of 256 characters by making the *i*th character in the sequence equal to the *i*th extended ASCII character. Position number 65 will store "A", 66 will store "B", for instance.

  After initializing the list of character, read in each 8-bit character ch from standard input one at a time, output the index in the array where ch appears, and move ch to the front. A sample result is shown next for a file that contains "**ABRACADABRA!**":

  ```
  % java MoveToFront - < abra.txt | java HexDump 16
  41 42 52 02 44 01 45 01 04 04 02 26
  12 bytes
  ```

  Note that your program itself DOES NOT output the characters "41 42 52...". These are the hexadecimal values obtained thanks to piping the output to HexDump. Here we can observe that the program first outputs one byte with value 65 (42 hexadecimal) which indicates position 65 in the character list followed by positions 42 and 52 (hexadecimal). At the beginning of the program all positions match the actual ASCII code, so position 41 represents "A", 42 "B" and 52 "R". When we analyze the fourth value we notice the effect of having reordered the list. After the first three characters are processed positions 0, 1 and 2 correspond to characters R, B and A because they have been moved to the front.

  When the fourth character is processed, "A" for the second time, it is no longer located at position 42 (hex) but at position 2. That is why the fourth byte output represents a 2. After "ABRA" comes a "C" character, which would initially be located at position 43 (hex) but is located at 44 when it is processed, due to the reordering of the list.

  Move-to-front decoding. Initialize an ordered sequence of 256 characters, where extended ASCII character *i* appears *i*-th 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 *i*-th character in the sequence, and move that character to the front. Check that the decoder recovers any encoded message.

  ```
  % java MoveToFront - < abra.txt | java 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)
}
```

Huffman encoding and decoding. Huffman.java implements the classic Huffman compression and expansion algorithms (described in the textbook). This program receives one command line parameter: a "-" if it needs to compress and a "+" if it needs to decompress.

```
% java Huffman - < abra.txt | java HexDump 16
50 4a 22 43 43 54 a8 40 00 00 01 8f 96 8f 94
15 bytes
% java Huffman - < abra.txt | java Huffman +
ABRACADABRA!
```

You do not need to implement anything. You should only use the program as is in order to complete the process

Submission

Checklist

Before submitting your project make sure that you comply with the following (while on data.cs.purdue.edu):

- Submit, at least, files MoveToFront.java and BurrowsWheeler.java. Huffman.java, HexDump.java, stdlib.jar and algs4.jar are not necessary
- Make sure that your program outputs binary information (Do not be confused by the above instructions when " | java HexDump 16")
- All the programs receive a command line argument: "-" for compression and "+" for decompression
- The input is received through standard input. In other words you execute programs in the following manner "java yourProgram < inputfile"
- Make sure you can manage binary inpout and output by using BinaryStdIn.java and BinaryStdOut.java
- Check that you can correctly generate all the test output files given
- If necessary, you can use the -Xmx option when running java to increase the amount of memory used by the program.

Submit your solution on or before **December 9, 2012**. turnin will be used to submit this assignment. The submission procedure is the same as for the first project. Inside your working directory for this project on data (e.g., ~/cs251/project5), create a folder in which you will include all source code used and libraries needed to compile and run your code. DO NOT use absolute paths in your files since they will become invalid once submitted. Optionally, you can include a README file to let us know about any known issues with your code (like errors, special conditions, etc). After logging into data.cs.purdue.edu, please follow these steps to submit your assignment:

- Enter the working directory for this project

```
% cd ~/cs251/project5
```

- Make a directory named <your_first_name>_<your_last_name> and copy all the files needed to compile and run your code there.
- While still in the working directory of your project (e.g., ~/cs251/project4) execute the following turnin command

```
% turnin -c cs251 -p project5 <your_first_name>_<your_last_name>
```

Keep in mind that old submissions are overwritten with new ones whenever you execute this command. You can verify the contents of your submission by executing the following command:

```
% turnin -v -c cs251 -p project5
```

Do not forget the -v flag here, as otherwise your submission would be replaced with an empty one.

Based on an assignment developed by Bob Sedgewick and Kevin Wayne.