Practical 9: Run Length Compression

Name: Rajit Banerjee
Student Number: 18202817

Data compression reduces the size of a file to save *space* when storing it and to save *time* when transmitting it. While Moore's law guides that the number of transistors on a chip doubles every 18-24 months, Parkinson's law tells us that data expands to fill available space. The text, images, sound, video, and so on that we create every day is growing at a far greater rate than storage technology.
As an aside and illustration: Wikipedia provides public dumps of all its content for academic research and republishing. To compress these large files, it uses bzip and SevenZip's LZMA (more on this later) and can take a week to compress of 300GB of data.

Today’s practical focuses on:

1. Build a run length encoded function that takes any string inputs and outputs the compressed output.
2. Make use of the helper files and the provided RunLength.java file to compress the files provided.

Step 1 Implement Run Length Encoding

Write your own Java function that takes in a string as a command line argument and returns a compressed string that uses Run Length Encoding (RLE).

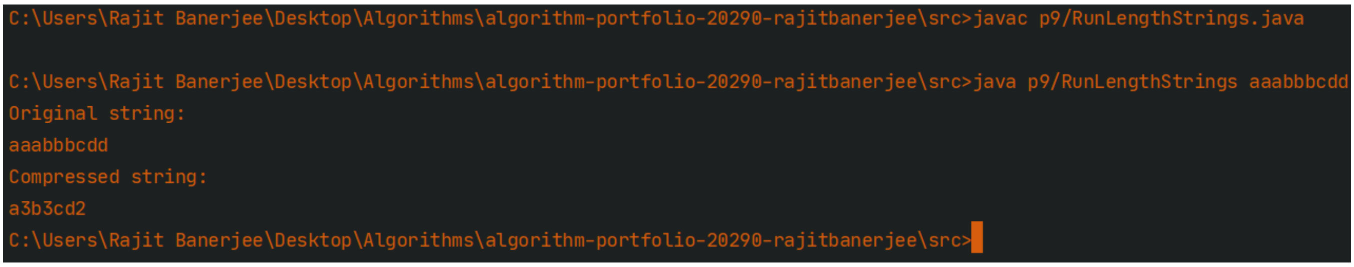
So, for example if the input (argument) into your program is:

“aaaabbbbb”

Then your program should return

“a4b5”

Implemented in class RunLengthStrings.java in package p9.
Dependencies: util/StdOut.java



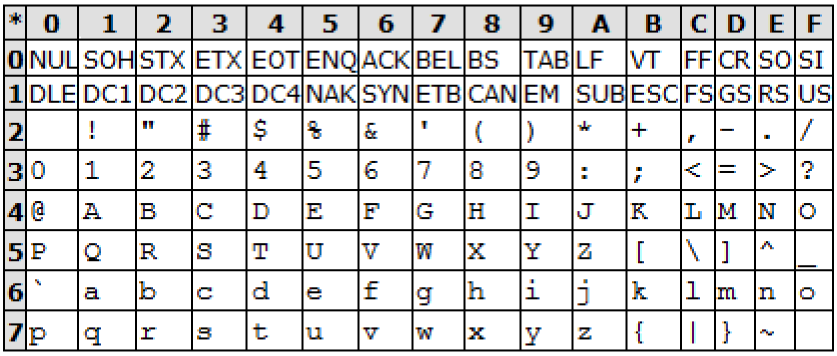
```
C:\Users\Rajit Banerjee\Desktop\Algorithms\algorithm-portfolio-20290-rajitbanerjee\src>javac p9/RunLengthStrings.java

C:\Users\Rajit Banerjee\Desktop\Algorithms\algorithm-portfolio-20290-rajitbanerjee\src>java p9/RunLengthStrings aaabbbcdd
Original string:
aaabbbcdd
Compressed string:
a3b3cd2
C:\Users\Rajit Banerjee\Desktop\Algorithms\algorithm-portfolio-20290-rajitbanerjee\src>
```

Step 2 Use the RunLength.java implementation that works with the binary input and output libraries provided

In the next step we are going to work at the bit level to measure the amount of compression we can attain by applying Run Length Encoding on a series of files (text, binary and bitmaps).

Included for your use in the repo are the following java files:

- BinaryStdIn - to work at the fundamental level of compression, we want to work at the bit level. BinaryStdIn is included to read in 1 bit at a time.
- BinaryStdOut - a corollary of BinaryStdIn but for write bits out 1 at a time
- BinaryDump - how can we examine the contents of the bits or bitstreams that we are working with (particularly while you are working on a program)? BinaryDump outputs the input in binary.
- HexDump - this is the same as BinaryDump but in hex code which is more compact if you can read it using the table below



*	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

- RunLength (see below) - implements Run Length Encoding (as you did in Strings) but in binary and builds on the previous libraries provided above

static void **compress**()

Reads a sequence of bits from standard input; compresses them using run-length coding with 8-bit run lengths; and writes the results to standard output.

static void **expand**()

Reads a sequence of bits from standard input (that are encoded using run-length encoding with 8-bit run lengths); decodes them; and writes the results to standard output.

static void **main**(**String**[] args)

Sample client that calls compress() if the command-line argument is "-" an expand() if it is "+"

Input files included in the repo include**:** 4runs.bin, abra.txt, q32x48.bin, q64x96.bin

Binary Compression

1. Begin by first outputting the number of bits in the binary file ‘4runs.bin’

Command used: java util/BinaryDump 40 < p9/4runs.bin

Output:

```
0000000000000001111111000000011111111111
40 bits
```

2. Now let’s try to compress this file with Run Length Encoding and see what we get (we’ll combine RunLength with BinaryDump to see how much compression we achieve)

Command used: java util/RunLength - < p9/4runs.bin | java util/BinaryDump 40

Output:

```
00001111000001110000011100001011
32 bits
```

Calculate the compression ratio: compressed bits / original bits

Compression ratio = 32/40 = 80%

3. Next, we’ll output this compressed file to a new binary file and check we have the same compression ratio.

Command used: java util/RunLength - < p9/4runs.bin > p9/4runsrle.bin

Check the bits: java util/BinaryDump 40 < p9/4runsrle.bin

```
00001111000001110000011100001011
32 bits
```

Compression ratio = 32/40 = 80%

So, the compression ratio is as expected.

ASCII Compression

1. Let's run through some of the same steps but with a text file

   Command used: java util/BinaryDump 8 < p9/abra.txt

How many bits do you get?

96 bits

2. Let's see what we can get to with compression

   Command used: java util/RunLength - < p9/abra.txt | java util/BinaryDump 8

How many bits did you get?

416 bits

What is the compression ratio?

Compression ratio = 416/96 = 433.33%

Why do you think you got this? What is happening?

Using run length encoding on the String "ABRACADABRA!" increased the number of bits (433.33%), instead of compressing it, thus showing that abra.txt is not suitable for run length encoding.

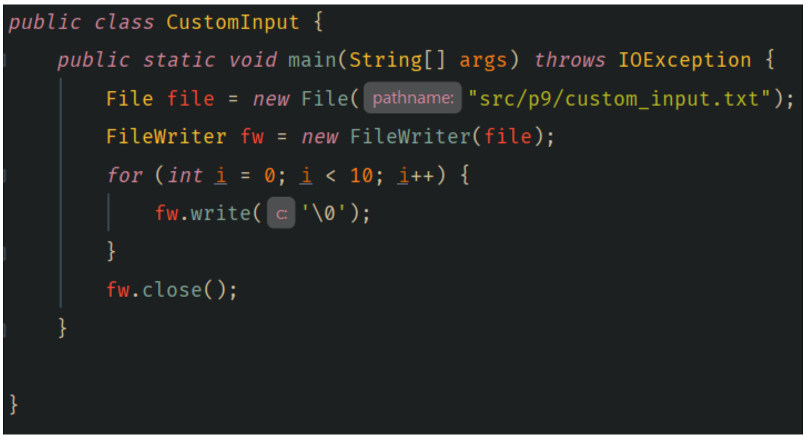
This is happening because this particular String contains ASCII characters, and its bitstream doesn't have enough long runs of 1s or 0s. The longest run available is only 5 0s. The original 96-bit long bitstream has multiple short runs, which end up being expanded when our run-length encoding is applied with 8-bit run lengths. E.g. a single 0 followed by single 1 is now encoded using 8 bits each: 00000001, followed by 00000001, thus contributing to the absurd compression ratio of 433.33% that we observe above.

3. Create your own text file that does lend itself to RunLength Encoding and perform the same steps as above, reporting your compression ratio.

This particular form of Run Length Encoding (for binary input, using 8-bit run lengths) is not generally suitable for text files containing ASCII characters. This is because ASCII characters in 8-bit binary codes don't contain long runs of 1s or 0s.

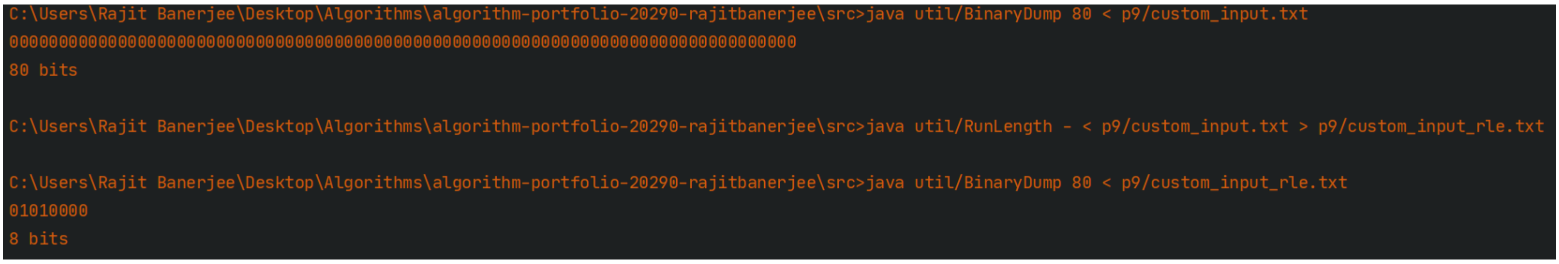
However, for the purpose of this question, I have written a program to generate a custom_input.txt file which only contains 10 null characters '\0' (ASCII code: 00000000). Here, we have bitstream run of 80 0s, which is suitable to be compressed using binary run length encoding.

Program to generate the text file containing only null characters.



```
public class CustomInput {
    public static void main(String[] args) throws IOException {
        File file = new File( pathname: "src/p9/custom_input.txt");
        FileWriter fw = new FileWriter(file);
        for (int i = 0; i < 10; i++) {
            fw.write( c: '\0');
        }
        fw.close();
    }

}
```

Compress the file using binary run length encoding:



```
C:\Users\Rajit Banerjee\Desktop\Algorithms\algorithm-portfolio-20290-rajitbanerjee\src>java util/BinaryDump 80 < p9/custom_input.txt
00000000000000000000000000000000000000000000000000000000000000000000000000000000
80 bits

C:\Users\Rajit Banerjee\Desktop\Algorithms\algorithm-portfolio-20290-rajitbanerjee\src>java util/RunLength - < p9/custom_input.txt > p9/custom_input_rle.txt

C:\Users\Rajit Banerjee\Desktop\Algorithms\algorithm-portfolio-20290-rajitbanerjee\src>java util/BinaryDump 80 < p9/custom_input_rle.txt
01010000
8 bits
```

Compression ratio = 8/80 = 10%

Bitmap Compression

Run Length encoding is widely used for bitmaps because this input data is more likely to have long runs of repeated data (i.e. pixels).

Step 1: Find out how many bits the bitmap file q32x48.bin has.

1536 bits

Step 2: Use Run Length function to compress the bitmap file q32x48.bin

Use the command to compress and output the compressed file to a new file:

java util/RunLength - < p9/q32x48.bin > p9/q32x48rle.bin

Now use BinaryDump to count the bits in the compressed file (‘q32x48rle.bin’).

1144 bits

Step 3: Calculate the compression ratio

Compression ratio = 1144/1536 = 74.48%

Step 4: Perform the Steps 1 and 2 on the higher resolution bitmap file q64x96.bin

Original bits = 6144

Compressed bits = 2296 bits

Compression ratio = 2296/6144 = 37.37%

Step 4: Compare the compression ratio of the first bitmap image to this second compressed bitmap image. What do you think is the reason for this difference?

The lower resolution bitmap file (q32x48.bin) has a compression ratio of 74.48%, whereas the higher resolution bitmap file (q64x96.bin) has about half the compression ratio of the former, at 37.37%.

This is because in the higher resolution file, both the length and width of the bitmap have been doubled.

Hence the following have increased:

- total number of bits, from 1536 to 6144 (i.e. 4 times)
- number of runs and run lengths of the 1s and 0s

These changes increase the number of bits in the compressed file from 1144 to 2296, approximately 2 times.

The result is therefore a much lower (about half) compression ratio.