Practical 9: Run Length

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”

Pseudocode

To implement RLE you need to first loop through the characters in the input string. You use an int counter that counts the number of times you have seen the same character in a row. Once you encounter a different character, you then output the value of the counter you’ve just seen and the number (your count variable) that you have been counting. Then repeat until you get to end of the input string. If you want to optimize your compression, you could choose not to output a count if there is only one instance of the character (i.e., “a” instead of “a1”).

Steps

1. Read in your input string
2. Start at the first character in the input string
3. Create a counter
4. Keep incrementing as long as you keep seeing the same character in a row
5. Once you encounter a new character, output the previous character and its count
6. Repeat the same steps with the next character until you reach the end of the string
7. Output the compressed string

```
PS C:\Users\patri\algorithms20290-2021-repository-PSeskauskas\src> javac Practical9/RunLengthEncoding.java
PS C:\Users\patri\algorithms20290-2021-repository-PSeskauskas\src> java Practical9/RunLengthEncoding aaaabbbbb
Original String: aaaabbbbb
Compressed String: a4b5
```

Step 2 Use the RunLength.java implementation provided 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). To refresh your memory, the **BIN files included** contain binary code (i.e. 0s and 1s). You’ll need to work in the Terminal (Mac), CommandPrompt (Windows), Shell etc., or use the input options in Eclipse for a lot of this exercise.

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.text
- q32x48.bin
- q64x96.bin

Binary Compression

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

Use the command: java **BinaryDump** 40 < 4runs.bin
Note down the bits.
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)

Use the command: java RunLength - < 4runs.bin | java BinaryDump
Note down the bits.

48 bits
Calculate the compression ratio: compressed bits / original bits
48/40 = 120% Compression

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

Use the command: java RunLength **-** < 4runs.bin > 4runsrle.bin
Use Binary Dump to check the bits: you can create this command yourself now
32 bits
32/40 = 80% compression

<u>ASCII Compression</u>

1. Let's run through some of the same steps but with a text file
   Use the command: java BinaryDump 8 < abra.txt

How many bits do you get?
96 bits

2. Let's see what we can get to with compression
   java RunLength - < abra.txt | java BinaryDump 8

How many bits did you get?
432 bits

What is the compression ratio?
432/96 = 450% Compression Ratio

Why do you think you got this? What is happening?
The number of bits of abra.txt after using Run Length Encoding increased by 450% instead of compressing it. Therefore, the abra.txt text file is not suitable for Run Length Encoding due to the increase of the number of bits.

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

Text file contains the string: "0000000000"

Original bits: 80 bits

Run Length compression bits: 184 bits

Compression to new file: 168 bits

Compression Ratio: 168/80 = 210%

<u>Bitmap Compression</u>
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: Use BinaryDump to find out how many bits the bitmap file q32x48.bin has
Note down your answer
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 RunLength - < q32x48.bin > q32x48rle.bin

Now use the BinaryDump function to count the bits in the compress file ('q32x48rle.bin').
1144 bits

Step 3: Calculate the compression ratio
1144/1536 = 74.48% Compression ratio

Q3: Perform the Steps 1 and 2 on the higher resolution bitmap file q64x96.bin
Note down the original bits and compressed bits.
Calculate the compression ratio?
Original File: 6144 bits
Compressed File: 2296 bits
Compression Ratio: 2296/6144 = 37.37% Compression Ratio

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 first bitmap image (q32x48.bin) has a compression ratio of 74.48%, whereas the second bitmap image (q64x96.bin) has a compression ratio of 37.37%. This is due to the fact that the second bitmap image has a higher resolution compared to the first bitmap image. Meaning the length and width have both been doubled, since 32 x 2 = 64 and 48 x 2 = 96. This means the total number of uncompressed bits has quadrupled from 1536 to 6144. Therefore, these changes lead to the number of bits of the compressed files to increase from 1144 to 2296. A 2x increase. Therefore, the result is a lower compression ratio, nearly half of the first bitmap.