Compressor/Decompressor for .NET platform (C#)

How to use the program:

The program is of the command-line type and supports two operations – “compress” and “decompress”. The files to be compressed\decompressed must be in the same directory with the program.

A program command has the following structure:

```
[command name] [original file name] [archive file name]
```

and it requires the full name of an input file. For instance, if we want to decompress the archive “archive.pdf.gz” and name the output file as “file”, then the correct command will be:

```
decompress archive.pdf.gz file
```

that will result in the file “file.pdf”.

After decompression, the output file inherits the extension decoded in an input file name that appears before the “.gz” extension. If an input file does not have an extension then the program omits it as well and thus the user can explicitly specify it in the output file name. For instance:

```
decompress fileWithNoExtension.gz file.txt
```

will result in the file “file.txt”, while

```
decompress archive.pdf.gz file.txt
```

will result in “file.txt.pdf”.

Compressor:

The most advanced idea is to do the following trick: we split an input file into pieces (blocks) and then compress each block separately thus getting a set of archives. Then, for each archive but the first one we cut off the headers (first 10 bytes + optional headers that we would need to recognize) and the footers (8 byte). Appending these cut archives to the first one without the footer we obtain one large GZip archive and we only need to append a new footer.

The footer of any GZip file consists of a CRC-32 checksum with appended size of the original file modulo 2^32. We can easily calculate the resulting CRC-32 checksum for an archive consisting of appended archives using exclusive OR operation on the checksums of those inner archives.

However, this approach does not work due to the fact that GZip uses DEFLATE algorithm for compression which is a mixture of LZ-77 algorithm and Huffman codes. Thus, the algorithm goes through the whole sequence of bits of the file being compressed and, according to the bits in the so-called “look ahead” buffer, compresses data. Also, using adaptive Huffman coding, it builds a code tree that highly depends on the bits that have been already processed. Thus, “gluing” together several encoded blocks we need to know the exact positions where those pieces were combined to delete the current Huffman code tree and to explicitly

tell the algorithm that it must not put the first bits of the next block into its look-ahead buffer during decompression.

Since we do not have all tools needed for update of Huffman code trees we will not be able to use this approach.

Another approach is more straightforward and does not require any intervention in the gzip archive structure. Here, we split the input file into blocks of size 1MB and compress each block in parallel independently thus getting a set of GZip archives. We put these compressed parts into the output file and provide it with footer.

The footer is in fact just a textual file that contains archive sizes in the order they appear in the final archive paired with indices of the archives. After all, we compress this footer file to keep the final archive as small as possible and append it to the main archive.

The last step is a computation of the size of the footer archive and appending it as a 32-bit number to the archive.

Thus, the structure of the final archive looks as following:

There is one thing to notice. The function BitConverter.GetBytes(long number) that we use for conversion of the footer size into an array of bytes employs big-endian ordering while the whole file uses little-endian ordering. Of course it does not influence our file since we can take this fact into account during decompression, however, to be consistent with the common file structure, we reverse the byte array gotten from the BitConverter.GetBytes(long number) function.

The original file extension can be stored in the footer archive alongside with the offsets of the inner archives. However, we will store it in the filename. Thus, for example, if the file to be compressed is named “Song.mp3” and we choose the name “Melody“ for the output file, the full name of the resulting archive will be “Melody.mp3.gz”. If the input file does not have an extension then the program omits it as well.

Decompressor:

Having the file structure presented above, we can easily parse an input file.

First of all, since the length of the “Footer archive size” block is known, we read last 32 bits of the input archive and convert it into Int32 (long) format. Then, using this number we read the input file further getting the footer archive that we can read as an ordinary textual file. During reading, we build a dictionary that keeps indexes of the inner archives alongside with their sizes. Also, we store the order which the inner archives

appear in. Then, we build the output file getting sequentially parts of the compressed file and decompressing them.

Performance:

We will compare the program with a commercial product – WinRar (".rar" extension) in its "Normal mode". The hardware domain that we will use for tests is:

- Processor: Intel(R) Core(TM) i7-8750 CPU @ 2.20GHz 2.21 GHz
- Memory: 16 GB RAM
- Operating System: Windows 10 Pro, version 1803, build 17134.765
- System type: 64 bit operating system, x64-based processor

The results are given in the following table:

File	Operation	WinRar	Veeam Job Interview Project
9.91 MB, ".mp3"	Compress	0 s.	0 s.
	Decompress	0 s.	0 s.
1.37 GB, ".mkv"	Compress	65 s.	22 s.
	Decompress	6 s.	9 s.
2.60 GB, ".bin"	Compress	139 s.	23 s.
	Decompress	16 s.	12 s.
23.4 GB, ".bin"	Compress	1748 s.	600 s.
	Decompress	578 s.	585 s.

We do not compare compression ratio of both programs since they use different compression algorithms.