Advanced Data Management for Data Analysis

Assignment 3

19.10.2021

Due: ***Monday, 15 November 2021, 09:15 CET***

Notes:

- Work in groups of (max.) 4 students.
- Produce a PDF document describing the functionality, implementation and use of your program.
- Accompany the document with a compressed archive containing your program's entire source code as well as any accompanying scripts (if any).
- Name your submission file
  ADM2021_A3_<student_id_1>_<…>_<student_id_4>.zip
  (with student IDs sorted in ascending order!)
- Submit via BrightSpace

Points:

This assignment is worth a total of 100 points, 10 points for the encoding and decoding for each of the five encoding schemes (equally split between the actual source code and its documentation, i.e., the report), plus max. 10 bonus points (see details below). The final score (grade) will be points divided by 10 to fit in the 0-10 grade system.

Please read the entire assignment instructions carefully and make sure you follow them properly.

Your task is to write ***(A) one*** or ***(B) multiple*** (details below) stand-alone program(s) that apply some of the data compression techniques we discussed in class to given data sets. While C or C++ are preferred / recommended, you can also use an other programming language of your choice, provided I can compile and run your program on a standard Fedora 34 Linux distribution.

The data sets are provided as single-column CSV (comma-separated values) files using line-end as record separator, i.e., text files that contain one value per row. Please be aware that the files are originally created on a Linux system, i.e., merely a single line-feed character (LF; ASCII: 10 (dec) / 0A (hex)) is used as end-of-line character, rather than Windows-typical carriage-return plus line-feed (CRLF; ASCII 13,10 (dec) / 0d,0a (hex)). The data type (see below) is encoded in the file name.
Please find the data sets at
https://homepages.cwi.nl/~manegold/ADM/ADM-2021-Assignment-3-data-T-SF-1.zip

In the assignment, we consider

- a total of 5 (five) compression techniques:
  - "**bin**": uncompressed binary format (for integer types, only), [1]
  - "**rle**": run-length encoding,
  - "**dic**": dictionary encoding,
  - "**for**": frame of reference encoding (for integer types, only),
  - "**dif**": differential encoding (for integer types, only);
- a total of 5 (five) input data types:
  - "**int8**": 8-bit (1-byte) integer,
  - "**int16**": 16-bit (2-byte) integer,

1 The "uncompressed binary format" ("**bin**") is not a compression technique as such, but merely means writing the (numerical) data as "machine-readable" raw bytes (as represented internally in the computer) rather than serialized into "human-readable" strings (text) as the data is provided.

- “**int32**”: 32-bit (4-byte) integer,
- “**int64**”: 64-bit (8-byte) integer,
- “**string**”: character string of arbitrary length.

(A)

In case you choose to implement ***everything in one single program*** (called **program.<*ext*>**, where **<*ext*>** is the standard file extension of the programming language you use), your program must accept four mandatory command line arguments (in this order):

1. “**en**” or “**de**” to specify whether your program should **en**code the given data or **de**code data that your program has encoded.
2. The compression technique to be used: “**bin**”:, “**rle**”, “**dic**”, “**for**”, or “**dif**”.
3. The data type of the input data: “**int8**”, “**int16**”, “**int32**”, “**int64**”, or “**string**”.
4. The **name (or entire path) of the file** to be en- or de-coded.

(B)

In case you choose to implement ***each compression/decompression technique for each data type in a separate program***, your ***44 programs*** must be called **program-**(**en**|**de**)**-**(**bin**|**rle**|**dic**|**for**|**dif**)**-**(**int8**|**int16**|**int32**|**int64**|**string**)**.<*ext*>** (where **<*ext*>** is the standard file extension of the programming language you use) and except the **name (or entire path) of the file** to be en- or de-coded as sole command line argument.

For encoding, your program must read the data from the given text (CSV) file from disk, and write the data back to disk in the requested encoding format (if applicable for the given data type). The output file name should be the input file name suffixed with the encoding acronym, i.e., “.**bin**”, “.**rle**”, “.**dic**”, “.**for**”, “.**dif**”. [2] [i]

For decoding, your program must read the encoded data from the given encoded file, and output the plain decoded data (text), i.e., the same format as the original provided data, to the console (stdout).

For each encoding, it is sufficient to use standard byte-aligned data types, i.e., code widths of 1/2/4/8 byte (8/16/32/64 bit). Improving the compression ratio by using sub-byte-aligned/wide codes and “bit packing” would yield bonus points (one for encoding and decoding per encoding scheme, i.e., max. 10 in total), if done correctly.

You need to accompany your program with a report (in PDF; called **report.pdf**) that documents how your code works, i.e., how you implemented each of the encoding techniques, how your program needs to be compiled, and how to run your program.
Additionally, your report must

- list the sizes of all (provided) input files and their respective output files (as produced by your program) for all applicable encoding techniques, and compare and discuss the different compression ratios;
- report how long your program(s) took to encode each provided file using each applicable compression technique as well as how long it took to decode each encoded file.

Pack the source code of your program(s) and your report in a single archive for submission:
ADM2021_A3_<student_id_1>_<…>_<student_id_4>.zip
(with student IDs sorted in ascending order!).

---

2 You can also opt for “simply”(?) writing the compressed data to the console (stdout) (to then be re-directed to a file using shell comman line functionality / syntax), rather than directly to a file. In either case, you need to handle binary data-/file-formats correctly.

i

Just in case, here are small indicative examples in Python & C to showcase the difference between "human-readable" text files and "machine-readable" binary files:

```
========
$ python
--------
Python 3.8.5 (default, Jul 31 2020, 00:00:00)
[GCC 10.2.1 20200723 (Red Hat 10.2.1-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> i=123
>>> t=open('int.txt','w')
>>> t.write('%d\n' % i)
4
>>> t.close()
>>> b=open('int.bin','wb')
>>> b.write(i.to_bytes(1, byteorder='big', signed=True))
1
>>> b.close()
>>>
========
$ file int.???
--------
int.bin: very short file (no magic)
int.txt: ASCII text
========
$ du -b int.???
--------
1     int.bin
4     int.txt
========
$ ls -l int.???
--------
-rw-rw-r--. 1 manegold manegold 1 Oct 5 20:22 int.bin
-rw-rw-r--. 1 manegold manegold 4 Oct 5 20:22 int.txt
========
$ cat int.txt
--------
123
========
$ hexdump -C int.txt
--------
00000000    31 32 33 0a    |123.|
00000004
========
$ hexdump -C int.bin
--------
00000000    7b             |{|
00000001
========
$ echo $[0x7b]
--------
123
========
```

```
========
$ cat z.c
--------
#include <stdlib.h>
#include <stdio.h>
int main (void) {
      signed char i = 123;
      FILE *t,*b;
      t = fopen("INT.txt","w");
      fprintf(t,"%d\n",i);
      fclose(t);
      b = fopen("INT.bin","wb");
      fwrite(&i,1,1,b);
      fclose(b);
      return(0);
}
========
$ gcc z.c -o z
--------
========
$ ./z
--------
========
$ file INT.???
--------
INT.bin: very short file (no magic)
INT.txt: ASCII text
========
$ du -b INT.???
--------
1     INT.bin
4     INT.txt
========
$ ls -l INT.???
--------
-rw-rw-r--. 1 manegold manegold 1 Oct 5 20:52 INT.bin
-rw-rw-r--. 1 manegold manegold 4 Oct 5 20:52 INT.txt
========
$ cat INT.txt
--------
123
========
$ hexdump -C INT.txt
--------
00000000    31 32 33 0a    |123.|
00000004
========
$ hexdump -C INT.bin
--------
00000000    7b             |{|
00000001
========
$ echo $[0x7b]
--------
123
========
```