Programming for Performance (BSCS2011)

ASSIGNMENT 1: File I/O, Memory, Compression, Proximity Search

1 GENERAL INFORMATION and OVERVIEW

1.1 Learning Objectives

1. Be able to write, compile, test, and debug, simple C++ programs.
2. Become familiar with files and streams in C++.
3. Learn how to use new and delete for memory allocation/deallocation.
4. Get experience using the std library (a bit).
5. Understand and implement some basic data compression and search algorithms.

1.2 Deadline

Solutions must be uploaded to Moodle by Friday, 3 April, 23:59 (1 minute before midnight). You will lose 10% of the total marks for each 24 period that passes after the above deadline. The final time you can submit a late assignment is Monday, 6 April, 23:59. Failure to submit your solutions before the final time (Monday, 6 April, 23:59) will attract a mark of 0 and will result in you being removed from the course on 7 April.

1.3 Rules (read carefully!):

1. Your submissions should be all your own work. No copying from peers or elsewhere. No cheating.
2. Apart from code you have developed yourself for the purpose of this course, use only the standard C++ libraries. No third-party libraries that you have downloaded from the Internet (or otherwise obtained) are allowed.
3. You should submit all your code in a single .zip archive, containing a single directory that contains all your code (subdirectories are allowed if you like). Your archive should contain a file called “README.Simon” that explains in detail how your code can be compiled and tested (i.e. run).
4. You may develop and test your code on whatever system you like, however, the code you submit must compile and run on the server **turso.cs.helsinki.fi**. You should make sure of this prior to submission. You should all have ssh access to this server from melkki.cs.helsinki.fi or melkinkari - contact Simon if you do not, for some reason.

1.3 Assignment Overview

In this assignment you are required to implement:

1. Variable byte (VByte) encoding and decoding routines.
2. Proximity intersection for sets.

VByte encoding is a famous compression scheme for integers that is used for efficiency in many software systems, including search engines. In the 1st programming task of Assignment 1 you will implement and use VByte to encode sets of integers. A brief description of VByte encoding is given at the end of this document and it will also be discussed in the first lecture on 12 March.

For the 2nd main task in Assignment 1, you will write code to compute proximity intersections on sets. A proximity intersection of set A with set B for bounds [x,y] is a set that contains all elements b of B for which there exists an element z in A such that b is in the range [z-x..z+y]. For example, if A = {3, 4, 18, 28, 129}, B = {0, 2, 29, 129, 131}, and [x,y] = [2,1], the proximity intersection is {2, 29, 129}. 2 is included in the result because it is at most 2 positions before 3 (and 4). Spend some time verifying the other elements in the result set yourself. Proximity intersections are at the heart of the proximity search operators offered in some search engines, and are also central to motif search, a common task in bioinformatics.

1.4 Marks

There are a total of 30 marks for this assignment:

- 15 marks for the VByte encoding task,
- 10 marks for the proximity intersection task, and
- 5+ marks for an “Exploratory Task” that you should argue for (as part of your report).

This last set of marks is obtained by you exploring different language features, doing things cleverly, or generally trying things out and discussing what you found. Marks will be awarded based on the difficulty and general interest of what you have tried, and the persuasiveness of your argument for getting the marks. Note that you can obtain more than 5 (and up to a maximum of 10) marks this way (and thus more than 30 marks for the assignment overall). Some suggestions for what you *could* try to obtain these marks are in Section 2, below. This is just a guide though, you should feel free to be creative.

2 PROGRAMMING TASKS

Complete the coding tasks below. The marks you obtain for each task will be given based on the code you produce (its quality, correctness, readability, comments) and the short report you write (10 pages total at the *absolute most*). In each task below, some specific things are asked for inclusion in your report, however your report can discuss any reflections or factoids related to the assignment you found interesting or want to mention.

Before You Start

Especially if you are new to C/C++ it is highly recommended that before you begin this assignment you go through the guide *Tie Koodereksi*, which is available in Finnish:
https://tie.koodariksi.fi/cpp/
and now in English:
https://tie.koodariksi.fi/cppe/ (bug finding in this translation is welcome ;-) contact Simon).
Read through the guide, compile some small programs, and try out the little code snippets in the guide. Other online introductory guides may also be helpful. Spend an intense day or so warming up this way before throwing yourself at the programming tasks of this assignment.

If you’re developing on Linux or Mac, it is also advised to at least quickly investigate the valgrind (especially --tool=memcheck) utility, and gdb, gprof, and make (to at least know what they are). There is an abundance of online documentation for these very useful tools.

Data

For the assignment you are provided with a set of files named F0, F1, F2, ..., each containing unsigned 64-bit integers (note: ***not*** in text format). They can be found here: https://www.cs.helsinki.fi/u/puglisi/PFP2020/assignment1/

Think of each file as containing a set of integers. The integers in the files are 64-bits (i.e. 8 bytes) in length, unsigned, and are not sorted in any particular way. The first seven integers in F0 are: 87513582 101675707 94584721 108822661 146209743 116123837 130973540.

Task 1: VByte Encoding/Decoding (15 out of 30 marks)

Do the following:

1. Write a program that accepts as a parameter a file name (e.g., an Fi from the set of files above), and produces a new file containing the integers from Fi compressed as VByte codes. The output file corresponding to input file "Fi" should be called "Fi.vb".
2. Write a program to decompress a file of VByte-encoded integers and output a file of 64-bit integers. The output file corresponding to the input file "Fi.vb" should be named "Fi.vb.dec".
3. Write a program that for an original file Fi, reads in the 64-bit integers, sorts them into ascending order, takes differences between adjacent elements in sorted order, and outputs the differences as VByte-encoded integers. The output file corresponding to input file "Fi" should be called "Fi.sorted.vb".

For each of your programs, measure the total time taken to process all files, recording both wallclock and user+system time. What do these numbers mean and why are they different? While you were developing your encoding and decoding routines, were you able to improve (or worsen) runtimes? If so, what changes made a difference?

How well does compression perform? How do the sizes of the .vb and .sorted.vb files compare to the original files?

Look around on the web and find a way to measure the peak memory usage of each of your programs. Explain the method you used and where you found it online (if you found it there). Record the peak memory usage of your programs and include them in your report, with some comments about variation between programs, if any.

Task 2: Proximity Intersection (10 out of 30 marks)

Write a program that takes three parameters:

1. An integer lower bound
2. An integer upper bound
3. A text file of pairs of integers, one pair per line. The integers correspond to VByte compressed files: integer x refers to file Fx.vb produced from the original file Fx.

The program should first read in all the pairs from the file to memory. It should then loop over the pairs, and for pair (x,y) it should read in files Fx.vb and Fy.vb, compute the proximity intersection of the two sets, and store the size of the result. Once all pairs have been processed, output the list of sizes (in the same order that they were computed) and the total time taken for all intersections (avoid including time used for reading the initial list of pairs).

Task 3: Exploration Task Suggestions (5 to 10 out of 30 marks)
Here are some suggestions you can choose from, with a rough guide for marks. Remember they are just suggestions: feel free to come up with your own ideas. The descriptions below are left deliberately vague (so that you still need to think a little bit).

1. "I used classes and inheritance for ..., because …" (1 mark). Exploring different C++ language features even if they're not absolutely necessary.
2. Program to generate random sets of test data for intersection the task (2-3 marks).
3. Write and compare different versions of your VByte encoder (4 marks). The first one reads the entire input file into memory, encodes it (storing the codes in memory), and then finally writes the codes out. The second version reads the input file one integer at a time and outputs the code for each integer as soon as each byte is computed. How do the runtimes and peak memory usage differ and why do you think this is?
4. Write an iterator for VByte codes (4 marks).
5. Implement an intersection algorithm for uncompressed 64-bit integers and uses, e.g., binary search (from std library) to speedup intersection (3-6 marks).
6. Implement an intersection algorithm that works on the sorted.vb files, decoding VByte and reversing the differences during intersection. Compare the times to those for the original files and also for .vb files (5 marks).
7. Speed up intersection time (maybe) by preprocessing the VByte encoded sets and storing samples of absolute values that enable you to skip over sections that cannot possibly lead to matches (7 marks).

3 BRIEF DESCRIPTION of VBYTE ENCODING

VByte encoding is perhaps best understood through examples. VByte represents an integer x as a sequence of one or more bytes. If x < 128 then the VByte encoding for x is simply a single byte containing the value (128+x). For example, the VByte code for x = 23 (which is < 128) is 151, or 10010111 in binary.

Bigger numbers (x ≥ 128) are encoded using more than one byte. For example, for x = 500, which is 111110100 in binary, VByte would output the following bytes (shown in binary):

01110100
10000011

Consider the two bytes above (in decimal they are 116 and 131, respectively). Looking at the underlined bits of each byte, we see precisely the bits that make up the original number x = 500 = 111110100. The lower order 7 bits of x are underlined in the first byte, and the remaining bits are underlined in the second byte.

Now look at the leftmost bit of each byte (in **bold**), which is called the *stop bit*. In the first byte we have a **0** and in the second byte a **1**. The **0** in the first byte indicates that the encoded number (x = 500) is greater than 127 and that some more of its bits are contained in the next byte. So we read the next byte. It's stop bit is **1**. This indicates that after this byte there are no more bits needed to reconstruct x.

So we have found the bits for x, but how do we reconstruct x itself? The following calculation does the job: $x = B_0 + (B_1 - 128)*128$, where B_0 is the first byte above, and B_1 is the second. Spend some time verifying this for yourself.

The scheme works for any size number. For example, the number 20,000,000 (20 million), which is 10011000100101101000000000 in binary, would have the following VByte encoding:
00000000
01011010
01000100
10001001

So, after inspecting the nth byte of a VByte encoded number x we have 7n bits of x and know whether or not we need to inspect the next byte to decode some more of x (stop bit = **0**) or not (stop bit = **1**). Spend some time writing down the equation to reconstruct x from the four bytes above that encode 20,000,000, as we did for the encoding of 500.

Your first action for this task should be to write some pseudocode for VByte encoding and decoding. Once you think about it, you'll see that we don't need to look at bits directly at all: integer division and modulus (for encoding) and multiplication and addition (for decoding) are all that is required. To get you started, here is some pseudocode for encoding.

```
VByteEncodeInteger(i): (INPUT: one integer, OUTPUT: one or more bytes)
1 while true
2       b = i mod 128
3       if i < 128 then
4               OutputByte(b+128) and BREAK
5       OutputByte(b)
5       i ← i div 128
```

If you get *really* stuck in understanding VByte, contact Simon - but do invest some time thinking about it before deciding if you really need to.

4. TURSO RESOURCES

As specified in the Rules, the code you submit is required to compile and run on turso.cs.helsinki.fi. You are also welcome to use this environment for developing your code. Tools such as g++, grpof, perf, and valgrind are available there. You should be able to ssh into turso from melkki.cs.helsinki.fi and melkinkari.cs.helsinki.fi, and possibly from other servers (let Simon know early if you're having trouble).

A message from IT support:
$HOME has a very tight quota, but that is on purpose.

Students should use $WRKDIR for all their needs, this is the primary Lustre based FS which provides adequate performance. All others are very slow due to being NFS shared over gigabit ethernet, saturated over to hundred nodes.

$PROJ is for user applications etc, mostly for researchers to use for projects.

Documentation for using turso resources is available here:
https://wiki.helsinki.fi/display/it4sci/Scientific+Software+Use+Cases#ScientificSoftwareUseCases-0.0AVeryShortCoursetoABatchScheduling
The Ukko2 user guide deals with more complex circumstances:
https://wiki.helsinki.fi/display/it4sci/Ukko2+User+Guide