Spring 2019 CS 32

Project 4
Compression Obsession

Time due: 11:00 PM Thursday, June 6

Introduction

Small-Mart, one of the world's largest retailers, wants to hire you to help them manage their inventory. A centralized file contains information about every item Small-Mart sells. This file is updated each day. All around the world, there are devices that need to receive a copy of the file each day: computers and cash registers in the stores, vending machines, handheld devices, etc., so each day, Small-Mart sends a copy of the inventory file to all these devices. There are 10 million such devices, and the inventory file is about 100 kilobytes long, so about 1 terabyte (i.e., 1000 gigabytes = 100KB * 10 million) is sent each day. Given that their internet provider charges Small-Mart roughly $1 per gigabyte, it costs about $1,000 per day to provide up-to-date inventory information to the devices. In addition to the inventory files, Small-Mart occasionally sends updates of other files to the devices as well (for example, new versions of applications as they are updated).

Marty Small, one of Small-Mart's smartest managers (and no relation to the company founder), recently had an idea. He realized that the file that Small-Mart sends each day changes very little from the file it sent the day before. For example, consider these two sample inventory files sent by Small-Mart to the devices on April 10 and April 11. Each file contains multiple records; each record has an item number, an item name, and the number of those items in the inventory:

April 10 version of the inventory file:

81609,Feather Duster,198,92246,Lawn Chair Set,50,03854,Carrano C++ book,183,27408,Monsters, Inc. DVD,89

April 11 version of the inventory file:

66284,Screwdriver,1000,81609,Feather Duster,195,92246,Lawn Chair Set,50,03490,Bedspread,87,27408,Monsters, Inc. DVD,89,40411,Hair Spray,380

As you can see, both the earlier and later version of the inventory file have many similarities. There are some new additions in the April 11 version of the file, such as the entry

03490,Bedspread,87

There are also some changes to existing records between the two versions:

81609,Feather Duster,198

changes to

81609,Feather Duster,195

And some of the entries in the original file have been removed (e.g., it looks like Small-Mart sold all of the Carrano C++ books in one day, since there is no entry for it in the second file).

Given the significant similarities between successive versions of the inventory file, Marty realized he could save Small-Mart hundreds of dollars every day. How? Marty realized that since a given day's inventory file is so

similar to the previous day's inventory file, that it doesn't make sense to ship the entire 100K inventory file to each salesperson every day. Instead, Small-Mart could just send a smaller file containing the list of changes (called a *diff file*) to each device to indicate what has changed from yesterday's inventory file to today's file. Each device could then run an *incremental update* program to use this diff file to convert their existing copy of the inventory file to the new version of the inventory file.

Let's number each character in both of inventory files and then see an example.

April 10 version of the inventory file:

```
            1         2         3         4
   01234567890123456789012345678901234567890123456789
  0 81609,Feather Duster,198,92246,Lawn Chair Set,50,0
 50 3854,Carrano C++ book,183,27408,Monsters, Inc. DVD
100 ,89
```

April 11 version of the inventory file:

```
            1         2         3         4
   01234567890123456789012345678901234567890123456789
  0 66284,Screwdriver,1000,81609,Feather Duster,195,92
 50 246,Lawn Chair Set,50,03490,Bedspread,87,27408,Mon
100 sters, Inc. DVD,89,40411,Hair Spray,380
```

Given these two versions of the inventory file, let's see what a diff file might look like. Remember, the diff file contains a set of instructions on how to convert an older version of the file (e.g., the April 10 version of the file) into a newer version of the file (e.g., the April 11 version). This diff file would normally be created at Small-Mart headquarters and then sent to each of the 10 million devices, which already have the full April 10 version of the inventory file:

1. Create a new, empty output file
2. Add 23 characters to the newer file: `66284,Screwdriver,1000,`
3. Copy 23 characters from offset 0 in the older file to the newer file.
4. Add 1 character to the newer file: `5`
5. Copy 27 characters from offset 24 in the older file to the newer file.
6. Add 16 characters to the newer file: `490,Bedspread,87`
7. Copy 28 characters from offset 75 in the older file to the newer file.
8. Add 21 characters to the newer file: `,40411,Hair Spray,380`

Given this set of instructions, an incremental update program on a device could follow the above set of instructions and convert their April 10 version of the file (which is already on their computer) into the April 11 version of the file. How would this work?

After following the first instruction, the incremental update program would create a new empty file.

After following the second instruction, that newer file would contain

```
66284,Screwdriver,1000,
```

After following the third instruction, the newer file would contain

```
66284,Screwdriver,1000,81609,Feather Duster,19
```

After following the fourth instruction, the newer file would contain

```
66284,Screwdriver,1000,81609,Feather Duster,195
```

After following the fifth instruction, the newer file would contain

```
66284,Screwdriver,1000,81609,Feather Duster,195,92246,Lawn Chair Set,50,03
```

And so on.

To perform this conversion, only two different instructions are required: a Copy instruction and an Add instruction. The Copy instruction specifies that a certain number of characters should be copied from a particular offset in the older version file to the end of the newer file. The Add instruction is used to add entirely new content to the newer file when this content cannot be located and copied from the older file.

Of course, the eight diff instructions shown above are in a human-readable format that is quite wordy. We can make our diff file much smaller by removing all of the human-readable text and using a more compact encoding. Shown below is a 93-character diff file containing all of the instructions above; note that this diff file is 33% smaller than the full April 11 version of the file:

```
A23:66284,Screwdriver,1000,C23,0A1:5C27,24A16:490,Bedspread,87C28,75A21:,40411,Hair Spray,380
```

In the example above, each add instruction is represented by an `A`, followed by the number of characters to add, followed by a colon character and the actual characters to append. Each copy instruction is represented by a `C` followed by the number of characters to copy, a comma, and the offset in the original file from which to start copying the characters. It is possible to convert any file to any other file using just these two types of instructions.

So to recap, each day, in its corporate office, Small-Mart can create the latest version of the full inventory file (e.g., the April 11 version). Then, Small-Mart can use a special tool called a *diff creator* to create a diff file (made up of Add and Copy instructions) that contains the instructions for converting yesterday's version (the April 10 version) of the file to today's version (the April 11 version) of the file. Then, Small-Mart can send this diff file to the 10 million devices instead of sending the full (much larger) April 11 inventory file.

Upon receiving the diff file, each of the 10 million devices then runs a program called an *incremental updater* that takes yesterday's full April 10 inventory file (which is already on the device, produced yesterday) and the just-received diff file, and follows the instructions in the diff file to create today's April 11 full inventory file. On April 12, the device will receive a diff file that it will use along with this full April 11 file to produce a full April 12 inventory file. Since the diff files are just a fraction of the full inventory files' sizes, this saves Small-Mart considerable networking costs.

Of course, this diff approach can be used to update all types of files, not just inventory files. For instance, consider the following two files A and A', where A' is a derived version of A.

```
            1         2         3
    01234567890123456789012345678901234
A  : ABCDEFGHIJBLAHPQRSTUVPQRSTUV
A' : XYABCDEFGHIJBLETCHPQRSTUVPQRSTQQELF
```

Here's a diff file to convert A into A'. Verify for yourself that it works:

```
A2:XYC12,0A3:ETCC13,13A5:QQELF
```

Now it may not be obvious, but there are actually many possible correct diff files that can be created to convert a first version of a file A into a second version A'. For example, here's another correct diff file for the example above:

```
A3:XYAC9,1A6:BLETCHC12,14A5:QQELF
```

And here's another correct, but not very useful, diff file:

```
A35:XYABCDEFGHIJBLETCHPQRSTUVPQRSTQQELF
```

This diff file simply contains the entire contents of A' and instructs the incremental updater to write all 35 characters of this data out to the output file. It's a bad solution because it's actually larger in size than A'! It would be a little cheaper to just send A' instead of this diff file.

Your Assignment

You have been hired by the Chief Frugality Officer (CFO) of Small-Mart to create two functions:

- `createDiff`: This function takes the contents of two files, A and A', and produces a diff file containing instructions for converting A into A'. Each day, Small-Mart will use this function at their corporate headquarters to create a new diff file that they will then send to all the devices.
- `applyDiff`: This function takes the content of a file A and a diff file, and will apply the instructions in the diff file to produce a new file A'. Each day, every device will use this function to update the previous day's inventory file.

The `createDiff` function has the following interface:

```
void createDiff(istream& fold, istream& fnew, ostream& fdiff);
```

You may name the parameters something else, but there must be three parameters of the indicated types in the indicated order. (The File I/O tutorial explains `istream` and `ostream`.) The parameters are

- an already-opened input source (for yesterday's full file, say)
- an already-opened input source (for today's full file, say)
- an already-opened output destination to which you will write the instructions for converting the first input to the second.

The `applyDiff` function has the following interface:

```
bool applyDiff(istream& fold, istream& fdiff, ostream& fnew);
```

You may name the parameters something else, but there must be three parameters of the indicated types in the indicated order. The parameters are

- an already-opened input source (for yesterday's full file, say)
- an already-opened input source (the diff file)
- an already-opened output destination to which you will write the file resulting from applying the diff file to the first input.

The `applyDiff`function returns true if the operation succeeds. If it fails because the diff file is malformed (e.g., there's a character other than an `A` or `C` where an instruction is expected, or an offset or length is invalid), the function returns false. If the function returns false, the caller can make no assumption about what may or may not have been written to the output destination (so you're free to, for example, just return false as soon as you detect an error, without regard to what you may have already written).

Except for the output required to be written to its third parameter, each of the two required functions must not cause anything to be written to `cout` or any file. They may write to `cerr` anything they want (presumably for debugging purposes); our testing program will ignore anything you write to `cerr`.

Your functions must be able to create diff files and apply diff files for files up to 100 kilobytes (102,400 bytes) in length, and if you wish, may be able to handle longer files as well.

The diff file that your implementation of `createDiff` produces must be in this format: A diff file is a sequence of zero or more *instruction*s, where an *instruction* is either

- an *Add instruction* of the form `ALen:text` in which *len* is a sequence of one or more digits representing the number of characters to add, and *text* is a character sequence of length *len*, the text to add; or
- a *Copy instruction* of the form `CLen,offset` in which *len* is a sequence of one or more digits representing the number of characters to copy, and *offset* is a sequence of one or more digits representing the (zero-based) offset in the original file from which to start copying.
- a *Do-nothing instruction* consisting of either a single newline character (`'\n'`) or a single carriage return character (`'\r'`).

An instruction with a *len* of 0 (such as `A0:` or, for an old file with at least 13 characters, `C0,12`) is valid and successful when executed by `applyDiff`, even though it has no effect since no characters are to be added or copied. Of course, it would be foolish for `createDiff` to produce such an instruction, since it needlessly makes the diff file longer. If a copy instruction specifies a length of 0 and an offset beyond the last character in the old file (e.g., `C0,12` for an old file with 12 or fewer characters), you may either consider it valid (and have it do nothing) or you may consider it invalid (and have `applyDiff` return false), your choice. Notice that an instruction like `A005:Hello` is valid (and has the same effect as `A5:Hello`).

The Do-nothing instructions exist solely to make working with diff files easier if you create or examine them under Windows or macOS or Linux, whose text editors may or may not append a newline or carriage return-linefeed at the end of the file. To execute a Do-nothing instruction, `applyDiff` does nothing. For example, a five-character diff file containing `C6,3•` (where • represents a newline character) causes the same effect that a four-character diff file containing `C6,3` would. Of course, it would be foolish for `createDiff` to produce a Do-nothing instruction, since it needlessly makes the diff file longer.

Your program must build and run successfully under g32 and either Visual C++ or Xcode.

When run under g32fast, your `createDiff` function must not run for longer than 15 seconds for any file of up to 100 kilobytes. Your `applyDiff` function must not run for longer than 5 seconds for any file of up to 100 kilobytes.

Here's an example of a main routine that performs a test of the functions, checking that the diff file produced by `createDiff` from an older file and a newer file can be used by `applyDiff` with the older file to produce a file identical to the newer file.

```
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include <iterator>
#include <cassert>
using namespace std;

bool runtest(string oldName, string newName, string diffName, string newName2)
{
    if (diffName == oldName  ||  diffName == newName  ||
        newName2 == oldName  ||  newName2 == diffName  ||
        newName2 == newName)
    {
        cerr << "Files used for output must have names distinct from other files" << endl;
        return false;
    }
    ifstream oldFile(oldName, ios::binary);
    if (!oldFile)
    {
        cerr << "Cannot open " << oldName << endl;
        return false;
    }
    ifstream newFile(newName, ios::binary);
    if (!newFile)
    {
        cerr << "Cannot open " << newName << endl;
```

```
            return false;
        }
        ofstream diffFile(diffName, ios::binary);
        if (!diffFile)
        {
            cerr << "Cannot create " << diffName << endl;
            return false;
        }
        createDiff(oldFile, newFile, diffFile);
        diffFile.close();

        oldFile.clear();   // clear the end of file condition
        oldFile.seekg(0);  // reset back to beginning of the file
        ifstream diffFile2(diffName, ios::binary);
        if (!diffFile2)
        {
            cerr << "Cannot read the " << diffName << " that was just created!" << endl;
            return false;
        }
        ofstream newFile2(newName2, ios::binary);
        if (!newFile2)
        {
            cerr << "Cannot create " << newName2 << endl;
            return false;
        }
        assert(applyDiff(oldFile, diffFile2, newFile2));
        newFile2.close();

        newFile.clear();
        newFile.seekg(0);
        ifstream newFile3(newName2, ios::binary);
        if (!newFile)
        {
            cerr << "Cannot open " << newName2 << endl;
            return false;
        }
        if ( ! equal(istreambuf_iterator<char>(newFile), istreambuf_iterator<char>(),
                     istreambuf_iterator<char>(newFile3), istreambuf_iterator<char>()))
        {

            cerr << newName2 << " is not identical to " << newName
                 << "; test FAILED" << endl;
            return false;
        }
        return true;
    }

    int main()
    {
        assert(runtest("myoldfile.txt", "mynewfile.txt", "mydifffile.txt", "mynewfile2.txt"));
        cerr << "Test PASSED" << endl;
    }
```

(Note: We create the streams in binary mode (`ios::binary`), which is harmless under macOS and Linux, but necessary under Windows to make the offsets work correctly, since Windows represents line ends for text files with two characters even though it delivers just one newline character to and from your C++ program.)

(Note: We close the output streams to ensure that all output to the file is completed before we open it for input. Because `createDiff` reached the end of file on the `oldFile` input stream, we need to clear the end of file condition on the stream and reset the stream so that we read from the beginning again. We do the same for `newFile2` before comparing the files.)

Here's another main routine that tests the functions. It uses the types `istringstream` (an input source where the characters read come from a string instead of a file), and `ostringstream` (an output destination where the characters written are later available as a string).

```
#include <iostream>
#include <sstream>  // for istringstream and ostringstream
#include <string>
#include <cassert>
using namespace std;

void runtest(string oldtext, string newtext)
{
    istringstream oldFile(oldtext);
    istringstream newFile(newtext);
    ostringstream diffFile;
    createDiff(oldFile, newFile, diffFile);
    string result = diffFile.str();
    cout << "The diff file length is " << result.size()
         << " and its text is " << endl;
    cout << result << endl;

    oldFile.clear();   // clear the end of file condition
    oldFile.seekg(0);  // reset back to beginning of the stream
    istringstream diffFile2(result);
    ostringstream newFile2;
    assert(applyDiff(oldFile, diffFile2, newFile2));
    assert(newtext == newFile2.str());
}

int main()
{
    runtest("There's a bathroom on the right.",
            "There's a bad moon on the rise.");
    runtest("ABCDEFGHIJBLAHPQRSTUVPQRSTUV",
            "XYABCDEFGHIJBLETCHPQRSTUVPQRSTQQELF");
    cout << "All tests passed" << endl;
}
```

Neither of the main routines above check that the diff file is validly formatted. You can run this Windows program or this Mac program or this Linux program to see if a diff file is validly formatted.

We will provide you with several different old and new files for you to test your program with, in a Windows version and a Mac and Linux version. For each of pair of old and new files, your program ***must*** create diff files that are *at least 5% smaller* than the new version of the file. We will test your program on other files as well, so it can't work *just* for these sample files. Below we show how big *our* solution's diff file is for each of these sample pairs of files:

	Windows file sizes in bytes			Mac and Linux file sizes in bytes		
	old	new	our diff file	old	new	our diff file
Small-Mart inventory	105	141	95	104	140	94
Weird Al's Dr. Seuss	533	606	69	510	579	69
War and Peace	77285	77333	399	76634	76682	393
Some strange files	100764	100764	8746	100440	100440	8746

Algorithm Requirements

You may be wondering how to write a function to create a diff file. This is definitely a non-trivial problem, and most senior job interview candidates can't come up with a viable algorithm (i.e., a conceptual approach) within a one-hour interview.

Here's a general, high-level algorithm that can be used to build a diff file. It's not ideal, however, and you'll have to come up with improvements of your own to make it work well:

1. Read in the entire contents of the old file into a string. Read the entire contents of the new file into another string.
2. For all consecutive N-character sequences in the old file's string, insert that N-character sequence *and* the offset F where it was found in the old file's string, into a table (e.g. hash table or binary search tree). You might use 8 for N, or maybe 16.
3. Once you have filled up your table with all N-byte sequences from the source file, start processing the new file's string, starting from offset j=0, until j reaches the end of the string.
    a. Look up the N-byte sequence which starts at offset j ([j,j+N-1]) in the new file's string in the table you created from the old file's string.
    b. If you find this sequence in the table, you know that that sequence is present in both the old and new versions of the file.
        i. Determine how long the match goes on (it might be just N bytes long, or it might extend past the first N matching bytes to be many bytes longer).
        ii. Once you have determined how long the match is (call this L), write a Copy instruction to the diff file to copy L bytes from offset F from the source file.
        iii. Go back to step 3a, continuing at offset j = j + L in the new file's string.
    c. If you don't find the current sequence (new file's string [j,j+N-1]) in the table you created, you know that the first version of the file doesn't contain the current N byte sequence.
        i. Write an instruction to the diff file to Add the current character.
        ii. Increment j by one to continue past the character used in the add instruction.
        iii. Go back to step 3a, where we'll try to find the next N-byte sequence in our table.

Of course, this is a simple version of the algorithm, and many improvements are possible. For example, this version of the algorithm will result in only single-character add instructions: If the new file contains the new text `BLAH` (not present in the old text), then the above algorithm would produce four Add instructions (`A1:BA1:LA1:AA1:H`) instead of a single (and more compact) Add instruction that adds all four new characters at once (`A4:BLAH`).

To obtain the highest score possible and create the smallest diff files, you'll have to improve on the algorithm substantially. Be creative! In addition, this naïve algorithm also has troubles with certain types of files (such as the strange*.txt sample files that we provide). You'll have to figure out why and make sure this doesn't cause problems in your solution.

For this project, we are constraining your implementation:

- You are required to use either a hash table or a binary search tree (your choice of which) for your table data structure.
- The only containers from the C++ standard library you may use are `vector`, `list`, `stack`, and `queue` (and `string`). You'll thus have to implement your BST or hash table yourself instead of using a `map` or `unordered_map`, but you can, say, use the `list` type to help you implement a hash table. Although we're limiting your use of *containers* from the library, you are free to use library *algorithms* (e.g., `sort`). You are also free to use `std::pair` from `<utility>` and `std::hash` from `<functional>` if you wish.

Compared to creating a diff file, the algorithm for applying a diff file is much easier. You may use the following functions to aid you in extracting instructions from the diff file if you wish:

```
bool getInt(istream& inf, int& n)
{
```

```
    char ch;
    if (!inf.get(ch)  ||  !isascii(ch)  ||  !isdigit(ch))
        return false;
    inf.unget();
    inf >> n;
    return true;
}

bool getCommand(istream& inf, char& cmd, int& length, int& offset)
{
    if (!inf.get(cmd))
    {
        cmd = 'x';  // signals end of file
        return true;
    }
    char ch;
    switch (cmd)
    {
      case 'A':
        return getInt(inf, length) && inf.get(ch) && ch == ':';
      case 'C':
        return getInt(inf, length) && inf.get(ch) && ch == ',' && getInt(inf, offset);
      case '\r':
      case '\n':
        return true;
    }
    return false;
}
```

Turn it in

What you will turn in for this assignment is a zip file containing these files and nothing more:

1. One or more source files that contain the code for the two required functions and any supporting types and functions you use to implement them. Your source code should have helpful comments that explain any non-obvious code. You may include a main routine if you wish; if you do, we will rename it to something harmless, never call it, and instead use our own main routine.

How nice! You don't have to turn in a report file this time.

Grading

Your `applyDiff` function is worth 22% of the points for this project.

The correctness of your `createDiff` function is worth 54% of the points, subject to these caveats: You will receive no points for any test case that produces a diff file that is not at least 5% smaller than the file that is the second argument to `createDiff`, and you will lose half of the correctness points you earn if you use any library containers other than those we allowed. (In other words, do not use library containers that are implemented using trees or hash tables, such as `set`, `map`, `unordered_set`, `unordered_map`, etc.)

The size of the diff files that `createDiff` produces is worth 24% of the points. The smaller your diff files, the more points you'll earn. For each test case for which you produce a correct diff file, of the diff file size points possible for that case:

- You earn 25% of those points if the diff file you produce is at least 5% smaller that the file that is the second argument to `createDiff`.
- You earn 50% of those points if the diff file you produce is between 5% and 20% larger than the diff file that *our* solution produces.

- You earn 75% of those points if the diff file you produce is between 0% and 5% larger than the diff file that *our* solution produces.
- You earn 100% of those points if the diff file you produce is smaller than the diff file that *our* solution produces.