ITP 435 Assignments

PA1: RLE / Compressing Strings

Compressing Strings

Recall that in test driven development, you write your unit tests first. The idea is it makes you think about all the edge cases *before* you begin coding the actual solution.

We use the Catch unit testing library. `StudentTests.cpp` has some tests to start out with. For this PA, you will mostly add new unit test sections under "RLE Compression" and "RLE Decompression." Let's focus on compression for now.

> **NOTE**
> If you have issues implementing the compression algorithm, for a change of pace try the decompression instead, as most students find the decompression simpler to implement.

Compression Test Cases

The format of each compression test case is the same. You have a test vector, which is the input to RLE, and the expected vector, which is your expected output. Let's look at the declaration of the test vector in the first example:

```
std::vector<int8_t> test = {
    'a','a','a','b','b','b','c','c','c',42,42,42,
    'a','a','a','b','b','b','c','c','c',42,42,42,
    'a','a','a','b','b','b','c','c','c',42,42,42,
    'a','a','a','b','b','b','c','c','c',42,42,42,
    'a','a','a','b','b','b','c','c','c',42,42,42,
    'a','a','a','b','b','b','c','c','c',42,42,42,
    'a','a','a','b','b','b','c','c','c',42,42,42,
    'a','a','a','b','b','b','c','c','c',42,42,42,
};
```

This is uses what's called an initializer-list syntax to allow us to specify the elements we want in the vector without using a loop.

The expected vector similarly uses that syntax to define the output:

```
std::vector<int8_t> expected = {
    3,'a',3,'b',3,'c',3,42,
    3,'a',3,'b',3,'c',3,42,
    3,'a',3,'b',3,'c',3,42,
    3,'a',3,'b',3,'c',3,42,
    3,'a',3,'b',3,'c',3,42,
    3,'a',3,'b',3,'c',3,42,
    3,'a',3,'b',3,'c',3,42,
    3,'a',3,'b',3,'c',3,42,
};
```

Now write your own test cases for compression. For full credit, you need **7 more** test cases for a total of 8 compression test cases. Here are a few ideas:

- A long positive run
- A long negative run
- Alternating positive/negative runs
- A very long positive run that goes greater than the max run size

As discussed in class, the easiest way to make additional test case sections is to copy the existing one and just changed the input/expected vectors. Keep in mind that your test cases should fail right now. If they pass then your test case is wrong!

NOTE

For consistency with our test files, assume that a single value run is a positive run of size 1.

Compression Code

In `RleData.h`, there is a `struct` called `RleData`. This has a vector called `mData` that you'll use to store the compressed data.

First, implement `RleData::Compress`. This function takes in input vector and compresses it with our RLE algorithm, saving it into the `mData` member array.

At the start of the function you should call `clear()` on `mData` just in case the struct is being reused.

Next, you can use the `reserve()` function to reserve enough memory for double the size of the input vector (which is the pathological worst-case largest output). Reserving the memory in advance will prevent you from reallocating the vector over and over as you add elements to it.

Now you need to implement the meat of the function – the algorithm to compress according to RLE. As mentioned in class, most students find it simplest to implement this in two passes. In the first pass, assume you only support positive runs and store the result in a temporary vector. Then in the second pass, find any neighboring runs of size 1 and change them into a negative run. Don't forget the max run size is +/- 127.

Decompression Tests

Now write some unit tests for decompression. As with the compression tests, I've given you one test to start with.

For full credit, write at least **3** more tests, for a total of 4 decompression tests. It's ok if you invert your compression tests.

Decompression Code

Implementing the decompression is significantly easier than compression. Notice that `RleData::Decompress` has two parameters instead of one. It still has the input vector but it also has the expected size of the decompressed data. This way, `Decompress` can reserve the exact amount of memory necessary.

The function needs to loop through the input array and convert it as appropriate. You know that the value at index 0 contains run information, so you can add that run to `mData`, and then move onto the next one.

Once implemented, you should pass all your decompression tests.

TEST IT!
You should push your code to GitHub now (as you should at the end of every part). GitHub actions will run the graded unit tests. If you pass all the tests for part 1, you will have a score of 24/75. Passing all of them means there's a good chance you wrote your code properly.

Once you've pushed this code, you're ready to move on to part 2.