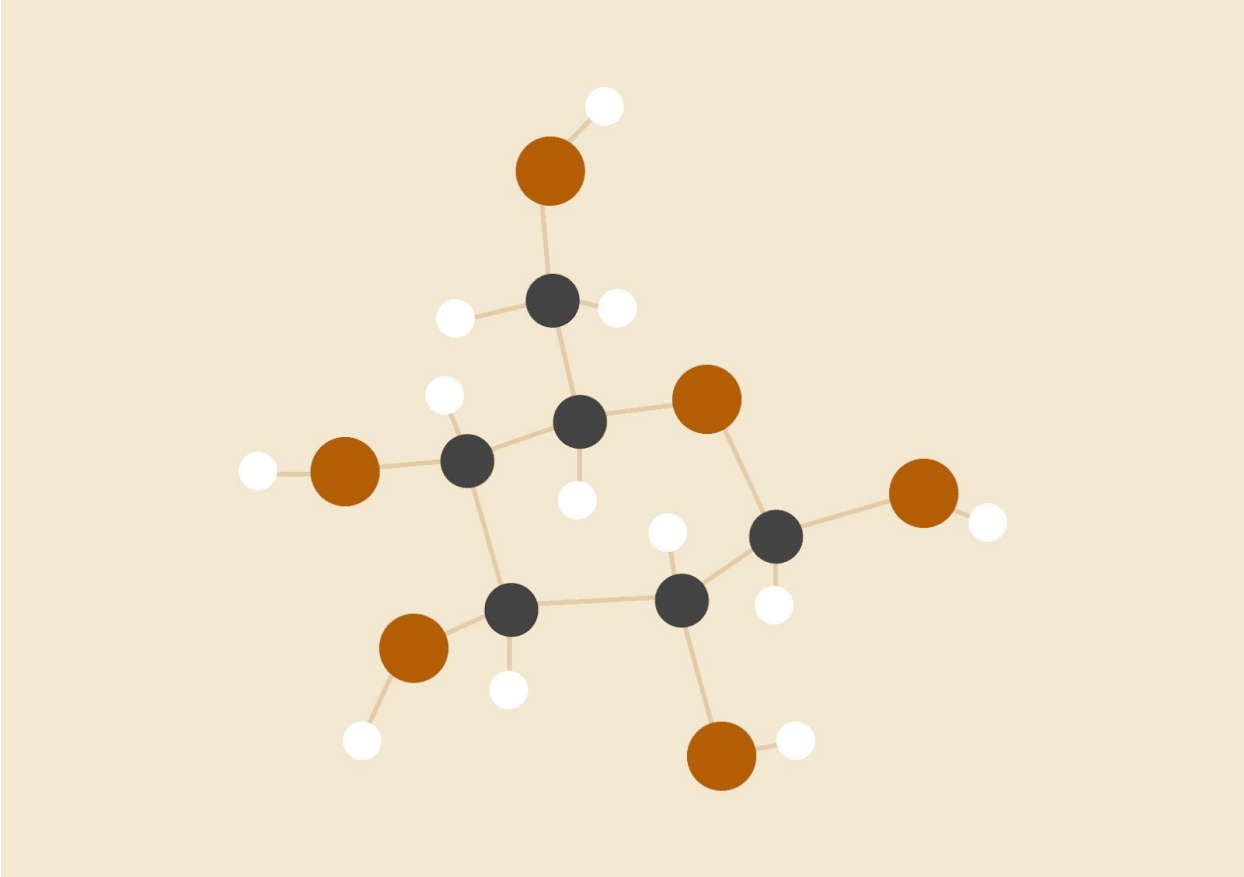
PROJECT #1

Complete Report



Azaria Fowler

October 1st, 2019
CSE 310 TTh 10:30am

PROBLEMS ENCOUNTERED

I had two major problems while working on the complete project. While working on merge sort for the second sorting algorithm for encode, I ran into some problems with pointers. While working on decode, I had a huge problem on how to parse the encoded line to generate the last string.

Merge Sort for Encode

For my implementation of merge sort, I had this section of code:

```
for (int k = l; k < r; k++) {
     if ((*firstHalf[i]) <= (*secondHalf[j])) {
          array[k] = firstHalf[i];
          i++;
     } else {
          array[k] = secondHalf[j];
          j++;
     }
}
```

The comparison seemed to be working, however, `array[k] = secondHalf[i];` did not. This obviously had something to do with my misunderstanding of pointer, but I couldn't understand why my array of type vector<string*> would not be able to assign another string* to its index. There was also a problem where I would run into a segmentation fault whenever I wanted to run merge sort of two arrays bigger than length 1.

With the help of Yiran Lawrence Luo on Piazza, we decided to just completely reconstruct a new approach to merge sort. Instead of sorting the array in place, I had to make both merge sort and merge return arrays. Lawrence helped me with the following pseudocode mentioned in this git issue. I obviously added some changes, like for example, I would have to physically create the arrays before I passed them into this function.

Another big problem I ran into was newlines. I thought merge sort would handle newlines, but apparently, the system doesn't actually store a newline. It just noticed it and returned me nothing. So, I ran into a couple segmentation faults trying to run merge sort on a newline. I quickly fixed it with an if-else statement for if the vector was empty or not, and it worked just fine!

Parsing Encoded Line for Last

So, this problem took up a majority of my coding process. Most of it had to do with my misunderstanding of input buffer and how newlines and terminating characters were not the same thing. Basically, I initially had a while loop that read the index of the encoded line and then another while loop nested inside that read each token of the encoded line for construction of last. However, two problem instances caused me problems:

when there was a new line and where there was a space in the encoded line. Both would appear to 'halt' my outer while loop and return me wacky decoded results. I solved this problem in two phases:

Phase One: Recognizing the Encoding Pattern

First of all, I really had no idea how I fixed the halting while loop problem. However, I learned a lot about stringstream through the process. Apparently, `cin >> indexString` will leave a bunch of \n characters in your input buffer. So, to cut all the crap, I still continued with the `getline(cin, inputString)`.

For the inner loop problem, I used stringstream, which allowed me to get the line all at once and parse through it for the needed information.

```
string encodedLine;
getline(cin, encodedLine);
stringstream ss(encodedLine);

for (int i = 2; i < encodedLine.size(); i += 4) {
     ss >> count;
     ss.get();
     letter = ss.get();

     for (int j = 0; j < count; j++) {
          last.push_back(letter);
          }
     }
```

The difficult part was figuring out how to:

1. Extract the count with consideration that the count could be more than one digit. Therefore, a simple indexing of the string will not work.
2. Extract the character with consideration that the character could be a space. Therefore, doing `ss >> letter` would throw off everything.

The way I figured this out was first realizing what had to be true for *all* occurrences, and then take advantage of that.

First, The string `encodedLine` will always be in this pattern: *count [space] letter*

Count may have multiple digits, so I cannot read just one character. However, when I use operator `>>`, it will give me the entire token up until a space. When the stringstream `ss` reads *count*, it stops right before *[space]*. Therefore, if I throw away that *[space]* , I will arrive at *letter*. Now, I can't just use `ss` again, because it ignores white space and therefore might ignore an actual whitespace we need to read. But, I know the length of the character will always be one (unlike *count*), so I can just do `letter = ss.get()`.

This worked for the decoding algorithm when I used insertion for both encoding and decoding.

Phase Two: Eliminating sstream

Inconveniently, it came to my attention that while sstream was compliant with the project guidelines as it is

part of the standard input/output library for C++, it was not allowed to be used in the project according to the professor. So, after lots of thinking, I realized I needed to come up with a more complicated way to ensure standard input calls would always capture the appropriate data in the encoded line.

Once again, I needed to figure out what was true for *all* encoded lines and use that to my advantage.

Each encoded line had the same pattern, but also included a newline at the end:

*count [space] letter [space] count [space] letter **[newline]***

Since I know the following:

1. `cin >> count` takes a token of variable character length, ignoring the whitespaces, but ends before the next whitespace is read.
2. `letter = cin.get()` takes one character, including whitespaces, but ends before the next character is read. Also, `cin.get()` doesn't have to be stored in a variable.

I know I must call `cin >> count` to account for double digits, and `letter = cin.get()` to get the letter. The problem is when do I terminate reading the encoded line?

Previously, my code said:

```
while (cin >> count) {
     cin.get();
     letter = cin.get();
     ...
}
```

So, if I had the following encoded line and the horizontal bar (|) indicated where the input reader has stopped:

```
2
10 c 3 f|
2
3 x 56 Q
```

When I call `cin >> count` in the while loop:

```
2
10 c 3 f
2|
3 x 56 Q
```

The buffer reading line will continue reading to the next line and store the next token; the index of the original string of the next encoded line.

So, I must check if the next *character* is a newline by using `cin.get()`. However, *when* would I call that?

Before I would call the while loop again, I must check if the next element is a newline with `cin.get()`. If it is, I will break out the loop, hence it is at the end of line. However, if it is not, then I really only got a space and I can continue on with the loop. There are no other possible characters I could have gotten.

Two Cases:

- `10 c| 3 f`
  `cin.get() = ' '`
- `10 c 3 f|`
  `cin.get() = '\n'`

However, what happens if I only have a newline? Specifically if the encoded line is:

```
0


2
3 x 56 Q
```

Well, if I didn't change the code, then `cin >> count` will run, thereby storing the *index* of the original string of the next encoded line into *count*. When I do `letter = cin.get()`, the new line for the *index* will be stored. Now, last contains four new lines in this example. The point is, it is wrong, and it's because new lines have to be checked before I enter the while loop.

What I know to be true is that a newline is very unique.

- An encoded line will always have one *index* and at least one *count* and one *letter*.

  index

  count *[space]* *letter**[newline]***

- A newline will only have one *index* and one character— a new line.

  index

  [newline]

So, before I enter the loop, I must check with `cin.get()` if this is a newline scenario by checking if the presumed "count" is a newline. Obviously, for a legit encoded line, count cannot be a newline, so if it is, I can safely assume that the "encoded line" is just of length one and consists of one newline character. In order to be compliant with my merge sort algorithm, I didn't alter last at all, thus signalling that the encoded line is simply a new line. However, if it is actually a legit number, we should rightfully put it back into the input stream with a function called `cin.putback()` so we will be able to use it again for when we start running our `while(cin >> count)` loop on the encoded line pattern.

KNOWN BUGS OR INCOMPLETE IMPLEMENTATIONS

There aren't any bugs to my knowledge when it comes to the implementation of the project milestone. Obviously it is incomplete in terms of the whole scope of the project, but I believe my implementation can handle all appropriate test cases to my knowledge.

SIGNIFICANT INTERACTIONS

1. **Yiran Lawrence Luo**

One of my problems dealing with merge sort was solved by Lawrence. He helped me look at merge sort in a different, yet simpler way that best suited my needs and data.

EXTERNAL SOURCES

1. **Input Streams**

Narek. "How to cin Space in C++?" *Stack Overflow*, 1 Aug. 1960,

https://stackoverflow.com/questions/2765462/how-to-cin-space-in-c/2765613."Std::Istream::Get."

Cplusplus.com, http://www.cplusplus.com/reference/istream/istream/get/."Std::Istream::Putback."

Cplusplus.com, http://www.cplusplus.com/reference/istream/istream/putback/."Std::Stringstream."

Cplusplus.com, http://www.cplusplus.com/reference/sstream/stringstream/."

Stringstream in C and Its Applications." *GeeksforGeeks*, 9 May 2018,

https://www.geeksforgeeks.org/stringstream-c-applications/.

2. **Piazza**

I asked two questions on Piazza. One of them pertaining to my merge sort problem and another one, unanswered, pertaining to the utilization of sstream in our projects.