Techniques For Big Data, Reading

If you remove the duplicate checking in the previous lab assignment in this module, you can convince yourself that the main contributor to runtime is that -- duplicate checking. Let's see why:

As we saw in a previous reading in this module, duplicate checking involved a list of already-seen combinations of term+section. Whether you used your StaticArray or DynamicArray template, and whether you used objects or concatenated strings in the list, that list grew from size zero at the start of parsing (when no combinations had already been seen) to 70,000 or so value stored at the end -- on the average, 35,000 values to look through before deciding (in most cases) that the newly found term+section had not yet been seen. Do that 70,000 times and you get 2,450,000,000 comparisons associated with duplicate checking alone.

By comparison, the list of subject codes and their totals was only about 100 in size, because that's how many different subject codes are in the TXT file. The inner-for loop of the sorting code block does 100 or so compares the first time it runs and 1 the last time, for an average of about 50. The loop runs 100 or so times, so that's 5,000 compares, and maybe half that many swaps. That's *way* less than the number of comparisons done in duplicate checking.

So if we want to find a way to improve the runtime in a big way, the place to look is duplicate checking, and the solution will have to include doing way fewer compares.

Doing Fewer Compares

We do so many comparisons because we have that one, long list of already-seen term+section combinations. So when a record for Fall 2010 gets read from the database, we have to skip past every already-seen that's not Fall 2010 before we have an opportunity to compare the section number. What if there was a was to not even have to *look* at the already-seen's that were not for Fall 2010? What if instead of one big pile of already-seen's, we have multiple piles -- one for each term, and in those piles we list the already-seen section numbers for that term?

To see if that has promise, let's look at the numbers. There are about 50 terms in the database -- 3 per year since the year 2000. Each could have its own pile of sections. Sharing 70,000 records among 50 piles puts about 1,400 in each. Now the process should involve, on the average, 25 compares to find the right pile (because sometimes it's the first pile we check and sometimes it's the last -- the average is in the middle). Once the pile is located, it's on average 700 sections to look through (because the piles start empty and end up with about 700 -- again the average is in the middle). That's 25+700 compares -- *way less* than 2 billion! It has promise!

Or we could have 10,000 piles of already-seen's -- one for each section number (because the highest section number is 9999). Each would list the terms in which that section number occurred. It seems like more comparisons that the previously discussed solution, but if you covert the section number into a numeric index, you could go right to its pile without a for-loop...

Applying Data Structures

Conceptually the solutions discussed above make sense. And there are certainly other solutions that may work as well or better. But how to program that? When the list of already-seen's was 70,000 long, and we used our StaticArray or DynamicArray template for the list, we really did not even need our fancy data structures. We could just as easily used regular static or dynamic arrays, and improved the runtime a little bit.

The 70,000-value array stored concatenated strings or objects with two string attributes. If anything, it was easy to program. But now we're going to need multiple arrays, each labeled with a term or section number. The arrays are easy enough -- concatenation is no longer needed, as the lists only contain a section number or term. We need something like this:

Fall 2013	Spring 2010	Summer 2009	Fall 2015	Fall 2001
`8234`	`1122`	`5623`	`0123`	`9012`
`1265`	`4512`			`9015`
`0324`				`8712`

Each time a line is read from the file, we search this structure for a matching term. If no match, we add a new box at the end. If the match *is* found, we search its pile for a matching section number. If found, it's a duplicate -- skip it! If not, add it to the already-seen list for that semester.

The overall data structure looks like an array that we fill starting with index 0, and hope its capacity is large enough to handle all the terms we'll see. The values stored in the array are made up of two attributes -- a string for the term and an array for the list of section numbers. Two attributes means an object values stored in the array are made up of two attributes -- a string for the term and an *array* for the list of section numbers. Two attributes means an object that looks like this:

```
struct SectionsForTerm
{
  string term;
  int numberOfSectionsSeen; // to track where we are in the following "array"
  an "array" of ints or strings
};
```

An idea data structure for the array is our **DynamicArray<string>**, because of it's ability to self-adjust its own capacity as needed:

```
struct SectionsForTerm
{
  string term;
  int numberOfSectionsSeen; // to track where we are in the following "array"
  DynamicArray<string> seenSectionNumber; // or int instead of string
};
```

Then we just need an array of such objects in the main program:

```
  int numberOfTermsSeen = 0; // to track where we are in the following "array"
  StaticArray<SectionForTerm, 200> alreadySeen; // 200 should be plenty for the number of terms
```

Program Design

Before trying to implement a solution like the one described above, it's a good idea to write down your "cast of characters". Things are a bit more complicated with objects and data structure of objects that have data structures as attributes. Here's what we have for the term-based lists solution:

- **alreadySeen[i].term** is an already-seen term, like Fall 2016, at array index `i`
- **numberOfTermsSeen** is for range of the for-loop for `int i`
- **alreadySeen[i].seenSectionNumber[j]** is a section number seen for for the term **alreadySeen[i].term** at for-loop index `j`
- **alreadySeen[i].numberOfSectionsSeen** is how many sections we've seen for the term **alreadySeen[i].term** -- it's the range of the for-loop for `int j`, which is nested inside the `int i` for-loop

Once you decide what you're going to call things in your program, you're ready to start coding.

Double-Duty For numberOfTermsSeen

Not only does `int numberOfTermsSeen` track how many terms we've seen, it is *also* the index of where to store the next term in the **alreadySeen** array. It's the next unused position in that array, waiting for you to find a new term that's not been seen before. Be sure to increment **numberOfTermsSeen** *after* using it as an index.

Double-Duty For alreadySeen[i].numberOfSectionsSeen

Likewise, not only does the `struct SectionsForTerm`'s `int numberOfSectionsSeen` attribute track how many non-duplicate sections we've seen for that term, it is *also* the index of where to store the next section in the **seenSectionNumber** array for that term. It's the next unused position in that array, waiting for you to find a new section that's not been seen before. Be sure to increment the **numberOfSectionsSeen** attribute *after* using it as an index.