COSC 2030 Computer Science II
Spring 2018

Lab 6: Uses of Sorted Sequences

Monday, 19 March 2018

When trying to answer questions about data contained within collections, those questions are often much easier to answer if the collection is sorted. In this lab you will be working with collections of data that have already been sorted for you, and your task is to generate linear time algorithms that answer certain questions.

Lab Assignment:

1. Provided in your repo under the code folder are the following files: SortedDriver.cpp, RandomUtilities.h, ContainerPrinting.h, and Timer files.

   You will be editing the SortedDriver.cpp file which produces instances of two problems.

   The first problem is to find the most isolated number in a collection. The most isolated number is the number whose nearest neighbor is farthest away. The nearest neighbor of a number is another number in the collection that is closest in value to the number. If you imagine the number line, then the most isolated number will be the number who has the most distance between its two neighbors out of any other particular number from our problem set. The other number may be larger, equal to, or smaller than the number. For this instance, ties may be resolved using the neighbor with the lowest index.

   For example, if the collection of numbers is {4.7, 9.3, 2.8, -6.2, 4.7, 1.8}, then the nearest neighbor of -6.2 is 1.8, the nearest neighbor of 1.8 is 2.8, of 2.8 is 1.8, of both 4.7's is 4.7, and of 9.3 is 4.7. The most isolated number in this collection is -6.2. Every other number has a nearer nearest neighbor. The nearest neighbors are easier to find if the collection of numbers is sorted.

   TODO: Compile and execute the code.

   Try a few small (< 10) sizes for the collection of numbers. Any small positive integer will work for the seed.

   The code to calculate the most isolated number is a stub. It is very fast, but it always returns -123.456 so it is almost always wrong.

   If you enter 0 for the size of the collection of numbers the program advances to the next section.

   The second problem is a variation on the set difference problem. Given two collections of words, the problem is to count the number of words in the first collection that do not appear in the second collection.

Try some small sizes for the word lists and small word lengths. The alphabet is any string, like abcd. (The word 'alphabet' here is often used in Computer Science, especially the theoretical side, to denote a specific set of letters that a problem will use).

The code to calculate the number of unmatched words is a stub. It always returns -1.

2. **TODO:** Write a function that determines the most isolated number in a sorted vector of numbers, vector<double>.

   The isolation of a number is the distance to the nearest other number.

   Your code should have time complexity Θ(n), where n is the size of the list. This means your algorithm must run in linear time.

   Test your function using size = 8 and seed = 123. The result should be 1.07565.
   Test your function using size = 8 and seed = 115. The result should be 3.3912.

3. Suppose you are given two sorted lists of short words, list<string>.

   TODO: Write a function that determines how many words in sorted list A do not occur in sorted list B.

   Your code should have time complexity Θ(n), where n is the sum of the sizes of the lists of strings. Note that two strings can be compared using the usual comparison operators, such as `==`, `<`, and `>=`.

   Test your function using parameters: <12, 2, abc, 123>. The result should be 4.
   Test your function using parameters: <8, 3, abcd, 123>. The result should be 6

4. Make a production run using <1000000, 552> for number input. The result should be -394924. My code constructed the problem in about 4.4 seconds. It required less than 0.3 seconds to find the result.

   Make a production run using <60000, 5, abcdefg, 557> for word input. The result should be 1681. My code constructed the problem in about 8.9 seconds. It required less than 0.3 seconds to find the result.

   You can test the run time of your programs using the included timer files. Run experiments and note the amount of time certain sizes of lists take to get an approximate idea of the execution time of your algorithms.

For Mac/Linux: The compilation command you need to use is “g++ -std=c++11 SortedDriver.cpp unixTimer.cpp”

Turn In:

- Code Files
- Screenshot of program output (titled output.png)

Do not rename any of the files, but add your name and date to the comments at the top of SortedDriver.cpp. Submit your code through Github and then submit the URL or your repo to wyocourses.

Criteria:

Grade Point 1	Nearest Neighbor found	5 points
Grade Point 2	Found in linear time	5 points
Grade Point 3	Correctly find all words in list A not in list B	5 points
Grade Point 4	Found in linear time	5 points