CS2020: Data Structures and Algorithms (Accelerated)

Problems 14–15

Released: Tuesday, March 8th 2011, Due: Wednesday, March 16th 2011, 13:59

Overview. You have two tasks this week. The first task involves hashing (from Dr Seth). The second task involves manipulating the two graph data structures that we have learned recently: the Adjacency Matrix and the Adjacency List (from Steven).

Collaboration Policy. As always, you are encouraged to work with other students on solving these problems. However, you **must** write up your solution **by yourself**. In addition, when you write up your solution, you **must** list the names of every collaborator, that is, every other person that you talked to about the problem (even if you only discussed it briefly). Any deviation from this policy will be considered cheating, and will be punished severely, including referral to the NUS Board of Discipline.

Problem 14. (Longest Common Substring)

For this problem, you will implement a version of the *Longest Common Substring* algorithm described in class, and use it to analyze the evolution of DNA.

Included in this problem set is a file *LongestCommonSubstring.java* that contains the framework for implementing your solution. It consists of a *main* routine that reads in two files (*string1.txt* and *string2.txt*), and calls the function `LCS` on the two strings. (We assume that each text file contains a single line of text.) This function should return the longest common substring of the two strings. It then prints out the length of the substring. Your task will be to implement the function `LCS`.

Problem 14.a. First, implement the function `existsSubstring(String A, String B, int L)`. If there exists a common substring of length L in the strings A and B, then the function returns the index in string B of the common substring. (That is, it returns a value i such that the substring $B[i, i + L - 1]$ is also a substring of A.) If there does not exist a common substring of length L, then it returns -1. For this part, implement a solution that uses hash tables (i.e., a `HashMap`) and runs in $O(n^2)$ time.

Problem 14.b. Now implement the `LCS`(String A, String B) function itself. You may implement `LCS` as a linear search, i.e., as a function that calls `existsSubstring` with n different values of L. (You will achieve better performance, however, if you impement `LCS` as a binary search procedure on the largest value of L.)

Problem 14.c. Biologists believe that the human chromosome 2 is actualy derived from two chimpanzee chromosomes: 2a and 2b. Attached to this problem set is a file `chromosome2.zip` which contains the following files. For chromosome 2:

- chr2_first_1000.txt, chr_2_last_1000.txt: the first and last 1000 base pairs of chromosome 2.
- chr2_first_10000.txt, chr_2_last_10000.txt: the first and last 10000 base pairs of chromosome 2.
- chr2_first_100000.txt, chr_2_last_100000.txt: the first and last 100000 base pairs of chromosome 2.
- chr2_first_1000000.txt, chr_2_last_1000000.txt: the first and last 1000000 base pairs of chromosome 2.

For chromosome 2a:

- chr2a_first_1000.txt, chr_2a_last_1000.txt: the first and last 1000 base pairs of chromosome 2a.
- chr2a_first_10000.txt, chr_2a_last_10000.txt: the first and last 10000 base pairs of chromosome 2a.
- chr2a_first_100000.txt, chr_2a_last_100000.txt: the first and last 100000 base pairs of chromosome 2a.

- chr2a_first_1000000.txt, chr_2a_last_1000000.txt: the first and last 1000000 base pairs of chromosome 2a.

For chromosome 2b:

- chr2b_first_1000.txt, chr_2b_last_1000.txt: the first and last 1000 base pairs of chromosome 2b.
- chr2b_first_10000.txt, chr_2b_last_10000.txt: the first and last 10000 base pairs of chromosome 2b.
- chr2b_first_100000.txt, chr_2b_last_100000.txt: the first and last 100000 base pairs of chromosome 2b.
- chr2b_first_1000000.txt, chr_2b_last_1000000.txt: the first and last 1000000 base pairs of chromosome 2b.

(Note that the capital and lower-case letters differentiate coding and non-coding portions of the chromosome; you may ignore the differences for this exercise. If your string comparison function is case-sensitive, it will automatically take this into account.)

By comparing these chromosomes, can you show that $2a$ and $2b$ did in fact merge to form 2? The faster your implementation, the more base pairs you can compare.

If you implement a rolling hash, it takes (on a fast computer) less than five minutes to find the longest common substring for the files containing a million base pairs. How many base pairs can you compare (in a reasonable time)?

Problem 15. (Graph Data Structure Manipulation)

You are given a template codes: `Graph.java` and `ii.java` that already contains Steven's implementation of an **Adjacency List** and **Integer pair**, respectively. You are also given another text file `g1.txt`, `g2.txt`, and `g3.txt` that will be read by `Graph.java`. Your task is to implement four methods that are currently left 'blank' inside `Graph.java`. There are also additional questions for each of these methods. Please answer all these questions as comments in `Graph.java`.

The four methods that you have to implement are:

- Write a Java method `public int[][] convert()` to convert the default **adjacency list** already implemented in `Graph.java` as `private Vector < Vector < ii > > AdjList;` into an **adjacency matrix**. What is the time complexity of your algorithm? Will your conversion algorithm works for each `g1.txt`, `g2.txt`, and `g3.txt`? Why?
- Write a Java method `public int[][] transpose(int AdjMatrix[][])` that will take in an **adjacency matrix** and transpose it. The transpose of a directed graph G is another directed graph on the same set of vertices with all of the edges reversed compared to the orientation of the corresponding edges in G. That is, if G contains an edge `(u,v)` then the transpose of G contains an edge `(v,u)` and vice versa. What is the time complexity of your algorithm? Apply your algorithm to `g1.txt` and then to `g2.txt`. Is there any phenomenon that you can you see? Elaborate!
- Write a Java method `public int countDegrees(int mode, int vtx)` that takes in the `mode` (0 for computing in-degrees, 1 for computing out-degrees) and a vertex number `vtx` then count the number of in (or out – depending on the `mode`) degrees of a certain vertex `vtx`. For this part, please work with Steven's original **adjacency list** implementation. What is the time complexity of your algorithm? Which `mode` is 'computationally harder' to compute? (in-degree or out-degree)? Why?
- Write a Java method `public boolean isAcyclic()` that checks if the graph is acyclic. A graph is said to be cyclic if there exists a path from vertex `u` to vertex `v`, and then there exists a 'back edge' `(v,u)`; otherwise, the graph is acyclic. For this part, please also work with Steven's original **adjacency list** implementation. What is the time complexity of your algorithm? Which input file is acyclic? `g1.txt` or `g2.txt`?

To facilitate faster grading (so that your tutor can concentrate on checking your algorithm instead of your code), please do not touch the `main` method of `Graph.java` so that the input/output format are standard across all submissions.