SIENAcollege
Computer Science

Computer Science 385
Design and Analysis of Algorithms
Siena College
Spring 2019

Problem Set 1
Due: start of class, Friday, February 1, 2019

You may work alone or in a group of 2 or 3 on this assignment. However, in order to make sure you learn the material and are well-prepared for the exams, you should work through the problems on your own before discussing them with your partner(s), should you choose to work in a team. In particular, the "you do these and I'll do these" approach is sure to leave you unprepared for the exams.

All GitHub repositories must be created with all group members having write access and all group member names specified in the `README.md` file by 4:00 PM, Monday, January 28, 2019. This applies to those who choose to work alone as well!

There is a significant amount of work to be done here, and you are sure to have questions. It will be difficult if not impossible to complete the assignment if you wait until the last minute. A slow and steady approach will be much more effective.

Getting Set Up

You will receive an email with the link to follow to set up your GitHub repository, which will be named `ps1-yourgitname`, for this problem set. Only one member of the group should follow the link to set up the repository on GitHub, then others will be granted write access.

Submitting

Please submit a **hard copy** (typeset preferred, handwritten OK but must be legible) for all written questions. Only one submission per group is needed.

Your submission requires that all required code deliverables are committed and pushed to the master for your repository's origin on GitHub. If you see everything you intend to submit when you visit your repository's page on GitHub, you're set.

Complexities You Already Know

? Question 1:

As a review of your work in prerequisite courses, indicate and briefly justify (in words, not mathematically) the "Big O" complexity for each of the operations below. Differentiate among best, average, and worst cases and under what circumstances they occur, where relevant. Specify the basic operation you are counting in each case. You may use any trustworthy resource, but any such resource must be cited. Be sure you understand any answers you need to look up, as you will see some of these or very similar questions as quiz or exam questions soon. (20 points)

- Find the largest value in an unsorted array of n integers.
- Perform a binary search in a sorted array of n integers.
- Add an element to an `ArrayList` that contains n values, using the default `add` method.
- Add an element to a singly-linked list that contains n values using the most efficient possible `add` method.
- Add an element to a sorted `ArrayList` that contains n integers, and which has capacity available to store the new value.
- Insert a new value into a balanced (*e.g.*, AVL) binary search tree that contains n values.
- Remove the root of a min-heap that contains n values.
- Sort an array of n integers using insertion sort.
- Determine if a specific value is currently stored in a sorted array of n integers.
- And one that you probably haven't seen, but should be able to reason out: count the number of times a specific value occurs in a sorted array of n integers.

Programming Task: A New Graph Method

In an undirected graph, such as the `HighwayGraph` from Lab 2, the *degree* of a vertex is the number of adjacent (undirected) edges.

1. Your repository includes a fresh copy of the `HighwayGraph.java` starter you used for Lab 2.
2. Add a new method to the `Vertex` class that computes the degree of the vertex with a given number. So the method call added in `main`:

   ```
   int d = g.vertices[12].degree();
   ```

   would compute and return the degree of vertex 12 and store it in `d`. (4 points)

   Of course, the above could be a good test, but you do not need to include such tests in your final submission.
3. Remove the `System.out.println(g);` line from the given `main` method.
4. Add code in `main` that computes and prints a table of the number of vertices of each degree in the graph. To do this efficiently in terms of memory, do **not** store lists of the vertices of every degree. You only need the counts, so only store the counts. Further, your solution should work for any maximum degree but should be memory efficient. That is, your storage should be $O(maxdegree)$, not $O(|V|)$ or $O(|E|)$. (7 points)

5. Add code in `main` that finds the largest degree of all the waypoints (vertices) in the graph, and then prints out a line containing the name of the input graph file, the largest degree, and lines for each of the vertex labels and their coordinates which have that largest degree. So here, you **will** need to store a list of vertices, but only store it for the largest degree, again so we are not wasting memory on things that are not needed. (6 points)

You should download and test with several of the graphs (include some larger ones, and the big one: `tm-master.tmg`.) on the METAL web site at `http://travelmapping.net/graphs/`. Your code will be tested on a wide variety of graph inputs. You can check correctness on small graphs by drawing them by hand or looking at them in HDX.

The reference solution produces the following output for the region graph for the state of Missouri:

```
Graph MO-region.tmg. |V| = 4658, |E| = 5394
Vertex degree counts:
0: 0
1: 135
2: 3307
3: 830
4: 381
5: 5
All vertices of degree 5
MO15/US24_E/US24BusPar (39.493634,-92.000027)
MO100@TukBlvd&US66HisMan@US66His_E&US66HisStL@MO100 (38.618748,-90.201691)
I-35@12&I-435@52B&MO110Han@I-35(12) (39.206404,-94.49156)
I-435(71)/I-49/I-470/US50/US71 (38.94142,-94.53685)
MO14/US65_N/US65BusOza (37.023593,-93.228714)
```

Your program should match this output format. You do not need to include test results in your submission.

Programming Task: Generating Example Arrays

In future problem sets, you will be asked to perform empirical analyses on the sorting algorithms we will be studying. To do this, you will need to generate input arrays for the sorting algorithms. In order to test the best, worst, and average cases of some of these algorithms, you will need to generate input arrays with various characteristics. For simplicity, we will work with arrays of `int`.

- an array filled with n random values within a given range
- an array filled with n values sorted in ascending order
- an array filled with n values sorted in descending order
- an array filled with n values "nearly" sorted in ascending order

Requirements:

- You may use any programming language, but be aware that you will need to implement the empirical analysis studies later using this code, so you will either need to do those studies in the same language or rewrite these generators later in any new language you choose.
- If you use Java, implement it within a class `IntArrayPopulator` that includes `static` methods to fill a given array of `int` with numbers matching each of the above characteristics, and a provide a `main` method that thoroughly tests these methods for various values of n and ranges. Be sure you can achieve similar functionality if you choose a different language.
- By taking the array to be filled as a parameter, you'll be able to reuse the arrays in your studies rather than re-allocating them each time.
- n will be determined by the length of the array passed to your methods.
- It is important that you generate these efficiently – for example, you should not generate sorted input by generating random input then sorting it. Generate it in sorted order right from the start.
- Your method to fill arrays with nearly sorted values should take a parameter that specifies the fraction of entries that are out of order. For example, if the parameter has a value of 0.05, approximately one of out of each 20 array slots should contain a value that's out of order. There are many reasonable ways to accomplish this.

Grading for the `IntArrayPopulator` will total 20 points: 3 points for the correctness of each of the 4 generator methods (including efficiency), 5 points for sufficient tests, and 3 points for design, documentation, and style.

Written Questions on Graph Representations

For the questions below, consider the graph representations discussed in class for storing directed graphs.

Suppose that the amount of memory required by the adjacency matrix graph representation is exactly $\frac{|V|^2}{8}$ bytes[1], and that the exact amount of memory required by the adjacency list graph representation is exactly $32|V| + 32|E|$ bytes[2].

? Question 2:

If you have a graph with 2^{21} (just over 2,000,000) vertices and each vertex has 4 outgoing edges, exactly how much memory in gigabytes is needed to store the graph using each representation? For each representation, can it fit into the 32GB of main memory which you might find in a high end PC today? (3 points)

? Question 3:

Graphs that have only a few number of edges per vertex are known as sparse graphs. A graph with a high number of edges per vertex is said to be dense. Suppose you have a complete directed graph of 2^{21} vertices. Here every vertex has a directed edge to all the other vertices. This is the "densest" graph there is! Exactly how much memory is needed to store the graph using each representation? Express your answer in gigabytes. (4 points)

? Question 4:

For a graph with 2^{21} vertices, where is the "break even" point, measured by $|E|$, below which the adjacency list representation is more memory efficient, and above which the adjacency matrix representation is more efficient? (4 points)

? Question 5:

What is the average vertex out-degree corresponding to your answer from the previous question? (2 points)

[1] This assumes each Boolean value in the array is stored as a single bit.

[2] This assumes 8 bytes for each reference in the `vertices[]` array, 24 bytes for each `Vertex` object (8 bytes for the `head` reference plus 16 bytes needed by Java to store housekeeping information about each `Vertex` object), and 32 bytes per `Edge` object (8 bytes for the integer `dest`, 8 bytes for the `next` reference, and 16 bytes to store Java housekeeping information about each `Edge` object).