Final exam debrief and answers (Dec 12th 2017)

PLEASE TREAT THIS DOCUMENT AS CONFIDENTIAL TO THE STUDENTS OF INFO6205, FALL 2017

First, an apology. The test cases for Q8 did not work properly and so some of you thought your answers were good when in fact they weren't. It also meant that I had to go through and score each question individually :(

Oh, and of course I have to apologize for the cramped conditions of the exam in the next-door room. I still don't know why our "slot" was taken for that time but I will know to check the classroom booking system next time.

Next, a word about my exams. I don't know what other professors do but if I set an *open book* exam, I have to try to make the questions challenging and unique. This is because, if I simply asked you to compare, say, merge sort and quick sort (many of you thought I had done just that in Q5), then you would simply be able to copy the answer from the book. Or, even easier, from a web site somewhere. Furthermore, I regard my exams, like the quizzes and assignments, as good practice for your job interviews. I want you to learn to think on your feet and not just be a parrot!

Finally, using HackerRank has its pluses and minuses for an exam like this. It turns out that the minuses greatly outweigh the pluses and, I look forward to using the new Blackboard for setting exams in the future.

Here are my answers and comments on the questions:

You mostly did well on Q1 (bags) and Q2 (red-black trees).

Q1: *Which of the following is an appropriate application of the Bag data structure?*

- A bag is a good choice to store the adjacent vertices of a vertex in a graph

Q2: *Which of the following statements about a red-black tree are true?*

- A red-black tree is a type of balanced search tree.
- A red-black tree of order 3 is logically equivalent to a 2-3 binary tree.
- A red-black tree of order 4 is logically equivalent to a B-tree of order 4

i.e. All of the above.

Q3: *Sometimes you will ask a colleague to help you with a problem and, as you are explaining the problem to him/her, the light bulb in your brain switches on and you no longer need their help.*

Remember how I emphasized that algorithms have nothing to do with any specific language? There are certain principles that transcend the choice of language. A junior colleague, who writes all his code in Scala has come to show you some code that is performing poorly. It is part of a Genetic Algorithm framework and this particular code has to do with females choosing the most attractive male from a "lek" (https://en.wikipedia.org/wiki/Lek_mating).

He thinks the problem is in the following method:

```
def bestMate(female: Y): Y = (males map { m
=> (m, mateable.attraction(female,
m)) }).sortBy(_._2).head._1
```

You know enough Scala to know that the _1 method yields the first element of a tuple, _2 the second, etc. And of course head yields the first element in a list.

The code works perfectly, but is slow and your colleague comes to you as the local "algorithms" expert.

(A) Explain to your colleague what he's doing wrong.

(B) Assuming that, on average, there are 1000 males in the "lek", by what factor do you expect to speed up this bit of the code?

Mixed results here. I had assumed that students would be at least somewhat familiar with the concept of *map* as it is defined in Java8 (Streams). Even so, it should not have obscured the fundamental issue which is that *sort* is unnecessary. Many students didn't seem to realize that what I was looking for in part B was a number. Why would the question specify a value for *n*

unless a number was required? Many students received 2/10 for this question.

BTW, for those of you who attended the final lecture/project session on the previous Thursday, I actually mentioned this very situation as an example of *reduction* where the reduction is not efficient.

A. (5 points) What he's doing wrong is simply that he's sorting the list, i.e. ***O(n log n)*** and then taking only the best. Clearly, the work required to put the *other n-1* elements in order is a total waste. A priority queue which yields the *n* best from a list *could* be a little more efficient, perhaps as good as *n + log n*. But best of all if all you need is one best item is just to take the *max,* i.e. *n* operations.

B. (5 points) On average, quicksort will take *2 n log n* operations. Max takes *n* operations. Therefore the speed-up of this part of the code is estimated to be a factor of *2 log n*. If *n* is 1000, this comes out to about 14. Mentioning *n log n* is worth 3 points even if the number is incorrect. I don't think anyone got a number even in the ballpark of 14 :(

Q4: *Implementing a symbol table by hashing is one of the most prevalent methods. Which of the following statements about a hash table are true?*

There was a lot more variation here. The correct answers were:

- Separate chaining is a method which can be reasonably efficient whether a table is fairly empty or almost full, but it tends to be somewhat wasteful of space.
- Linear probing is a method which can be very efficient for low density tables but which gets very inefficient when the table gets full.
- It is desirable that the hash function distributes the keys uniformly.
- A hash function is, ideally, suited to its key as a hash function which performs well on, say, Double values might not perform so well as a hah function on dates.
- The standard Java hashing function (hashCode) must be consistent with, i.e. based on the same "primary key" as, the equals method.

The incorrect answers were:

- Like a balanced search tree, it is important that the key to be hashed should have an order defined.
- The hash function must be chosen such that it is impossible to have collisions.
- One advantage of a Hash Table (over other symbol tables) is that it never has to be resized.

We had to manually deduct 1.5 for incorrect answers as HackerRank only gives the score for the correct answers and ignores the incorrect answers.

Q5: *It's tempting to think that the only difference between merge sort and quick sort is that in one case we process the whole first, then*

partition and recursively work on the partitions; while in the other case, we do it the other way around.

But there's a deeper difference than that, at least in my opinion. And it is essentially the reason why merge sort is well-behaved and predictable in its performance, while quick sort is not.
Imagine you are at an interview and briefly describe this significant difference. Remember: deep breath... think... then speak/write.

[Just like at an interview, I'm not looking for an essay. Express the difference as succinctly as possible.]

This question asks *why* quicksort is less predictable in its performance than mergesort. It didn't ask you to contrast and compare mergesort with quicksort.

The reason is the unpredictable pivoting on quicksort. Mergsort always partitions straight down the middle, but quicksort partitions at the *pivot* which, depending how much effort (cost) you put in to finding it, maybe be very unsymmetrical.

Q6: *There are three obvious ways to store graph information:*

1. *a list of edges*
2. *a vertex-by-vertex matrix*
3. *a list of vertices, each with its adjacency list*

Assuming a typical graph where $E = cV^2/2$ [E is the number of edges, V is the number of vertices, and c is some number ($c < 1$) that represents the average density of the graph and is typically much smaller than 1], order these three data structures according to the efficiency of space utilization (starting with the most efficient).

Answer: 1,3,2

Q7: *Given the same conditions as in the previous question, order the methods according to the number of operations required to*

determine if two vertices are connected by an edge, starting with the fewest.

Answer: 1,3,2

For both questions, we went through your answers manually and, if your answer could be transformed into the correct answer with a simple swap, you got half credit (2.5).

Q8: *Write the code to implement a linear-probing hash table.*

The question details mentioned *k,* and the format of input, etc. Many of you forgot all about *k,* the purpose of which was to prevent you from simply copying the algorithm from somewhere (which I think all of you did, including, those that modified for *k*). To earn full marks on this, you had to replace *i+1* with *i+k,* not *i+1+k*. Some of you incorrectly tried to add k to m as in *(i+1) % (m+k)*. A few of you forgot that *put* and *get* both have to use the same stepping function. Some of you, even those who got the *k* part right, forgot that if the key matches on a *put*, then you still have to update the value, not just return. This error was caught by the test suite. As I mentioned, apologies for not catching the *k* error in the tests. I actually was on the point of putting in the correct unit test when I heard about the classroom problem and had to come straight over.

The good news was that some of you actually got more marks than HackerRank awarded because it can only judge based on running the tests.

Q9: The Commonwealth of Massachusetts is creating a software system to maintain a web site for each of the 351 cities and towns in the state, keyed by 5-digit zip code (don't worry about the fact that many of them have multiple zip codes--just assume that there's only one zip per town/city). The idea is to use separate chaining with 128 buckets. There are 100,000 possible 5-digit zip codes which will require something like 17 bits. So, the programmer decides to simply strip off the least-significant 10 bits in order to get the hash.

Explain why this is a bad idea, giving a short analysis of the problem/solution (no more than a few lines of text).

To help you with this question (since you are obviously not as familiar with MA zip codes as I am) here is a representative sample of MA zip codes:

1001	Agawam
1007	Belchertown
1013	Chicopee
1033	Granby
1060	Northampton
1085	Westfield
1103	Springfield
1230	Great Barrington
1453	Leominster
1602	Worcester
1741	Carlisle
1854	Lowell
2108	Boston
2138	Cambridge
2360	Plymouth
2493	Weston
2780	Taunton

When you design a hash function for a Hash Table with N "buckets"/slots, you generate an integer of many bits (often 32) and you must *then* either use modulo N or shift (in the case were $N=2^p$) on this integer in order to determine which of the N buckets we should target. If N is a power of two, then these two operations are the same. In this case, $p = 7$ so that $N = 128$. That is given. Here, the programmer, uses the identity function as the first part of the hash (i.e no scrambling at all) so all of the real information is contained in something like the 11 least significant bits—you infer that by looking at the data. Then, stripping off (right-shifting) the least significant 10 bits will leave 7 bits without much variation. In fact, if I've done the arithmetic right, only two of the 128 buckets will be used, which will almost defeat the entire purpose of having a Hash Table!

Although the question didn't specifically ask you for a solution, it typically did enhance the clarity of your answer if you suggested a remedy.

Many of your answers were very vague on this and revealed a worrying lack of understanding of the mechanisms of hash tables!