STUDENT NAME: _____

STUDENT ID: _____

McGill University
Faculty of Science
School of Computer Science

FINAL EXAMINATION

COMP-250: Introduction to Computer Science - Fall 2010

December 20, 2010
2:00-5:00

Examiner: Prof. Doina Precup

Sample solutions

1. [50 points] **Short questions**

   (a) True or false: is $f(n) = 50n + \log_2 n$ in $O(n)$? Justify your answer.
   Solution: True, because:

   $$\lim_{n\to\infty} \frac{f(n)}{n} = \lim_{n\to\infty} \left(50 + \frac{\log_2 n}{n}\right) = 50$$

   which is a constant.

   (b) Does an algorithm with $O(n \log n)$ always run faster that an algorithms with $O(n^2)$? Justify your answer.
   Solution: No. $O()$ hides away constants. An algorithm which is $O(n \log n)$ would run faster in the limit, as $n \to \infty$, but the constants may cause slower running time for small amounts of data.

   (c) Give an example of a graph problem in which depth-first search would be preferable to breadth-first search.
   Solution: In problems with very high branching factor, depth-first search is preferable, because it uses less memory. Game tree search was one such example we discussed. Another example is web crawling (as in the Google mimi-project).

   (d) Your friend claims that he know an interesting card trick. If you are thinking of any card number (A, 2, 3, ... 10, J, Q, K), he will *always* guess what card you are thinking about by asking you just 3 questions. Do you believe him? Justify your answer.
   Solution: No. The fastest way to guess is by a binary search procedure. Since there are more than 8 numbers, your friend will need 4 questions for such a procedure.

   (e) True or false: In Java, if a variable is declared static, its value can only be modified by methods of the class in which it is declared.
   Solution: False. If the variable is declared public, any class can modify it. If it is protected (no public or private specifier), then all classes within the package can modify it.

2. [20 points]**Big-Oh**

   For the following algorithms, state what $O()$ is and explain your answer:

   (a) **Algorithm** f1(n)
   $i \leftarrow 1$
   while $i < n$
   for $j = n$ **to** i **do**
   print(j)
   $i \leftarrow i + 2$

   Solution: The running time is: $n + (n-2) + (n-4) + \ldots + 1 = \frac{n+1)n}{2}$ which is $O(n^2)$.

   (b) **Algorithm** f2(n)
   $i \leftarrow 1$
   while $i < n$
   print(i)
   $i \leftarrow i * 2$
   for $j = 1$ **to** i **do**
   print(j)

   Solution: The first loop is $O(\log n)$, the second loop is $O(n)$ (as i will be n at the end of the first loop). So the total complexity os $O(\log n) + O(n) = O(n)$.

3. [40 points] **More Big-Oh**

   Consider the following pseudocode algorithm:

   Algorithm MyAlg(n, k)

   if ($n == 1$ **or** $n == 0$) **return** k
   if (n **mod** $2 == 0$) **return** MyAlg($n/2$, $k + 1$)
   if (n **mod** $2 == 1$) **return** MyAlg($n + 1$, $k + 1$)

   (a) Suppose we call this with initial value $k = 1$. What will the algorithm return?
   Solution: It will return the number of times that n can be divided by 2 (so $\log_2 n$).

   (b) Write a recurrence for the running time of the algorithm. Note that you will need two equations: one for even values of n (i.e. $n = 2i$) and one for odd values of n (i.e. $n = 2i - 1$)
   Solution: We have the following recurrence:

$$\begin{aligned} T(1) &= c \\ T(2m) &= c_1 + T(m) \\ T(2m-1) &= c_2 + T(2m) \end{aligned}$$

(c) Based on the recurrence relation, what is the best case of the algorithm? Estimate the running time ($O()$) in this case, and justify your answer

Solution: The best case is when n is a power of 2. In this case, the even case of the recurrence is always picked, and we get $O(\log n)$ (by the same argument as in binary search).

(d) Suppose now that you get an input n which is NOT the best case. Expand the recurrence, assuming worst-case scenarios as much as possible. Based on this what is $O()$ of the algorithm? Give an answer that is as "tight" as possible.

Solution: The worst case is the odd recurrence. In this case, we have:

$$T(2m-1) = c_2 + T(2m) = c_2 + c_1 + T(m) = 2k + T(m)$$

(assuming k is a constant, $k > c_1$, $k > c_2$). Continuing the expansion, we would still get $O(\log n)$ (as in the previous case.

4. [10 points] **Inheritance**

Consider the following piece of Java code:

```
public class A {
        private char x;
        public A() { x='a'; }
        public void print() {System.out.println(x);}
}

public class B extends A {
        private char y;
        public B() { super(); y='b'; }
        public void print() {super.print(); System.out.println(y);}
}
```

What is the outcome of the following code:

```
B z = new B();
z.print();
```

Solution:

After the first call, z.x ia 'a' and z.y is 'b', so the code will print:
a
b

5. [50 points] **A queuing algorithm**

You are a programmer at a company that offers web services. You are tasked to write a Java class, Server, which is supposed to maintain a set of queues, and put arriving requests into the queues. You have been provided with a Request class, which provides the following method:

void process() //processes a request object

You are also provided with a Queue class, which contains the following methods:

Queue() //constructor that creates an empty queue
void enqueue(Request r) //Put a request into the queue
Request dequeue() // take the object from the front of the queue out
int size() //returns the number of objects in the queue

None of these methods throw any exceptions. If you call dequeue from an empty queue, the program will crash. The queue resize automatically, so it is never "full".

Write on the next page **Java code** for the Server class, which contains the following methods:
Server(int n) - creates n empty queues
void place(Request r) - places the request in the next queue
//For example, if you had 3 queues, the first request would go in the first queue, the second request in the second queue, the third request in the third queue, the fourth request in the first queue, the fifth request in the second queue etc.
void process() - dequeues the first request from the *longest* queue and processes it

Please make sure that you declare all needed variables and you write the code for all the methods.

Solution:

```
public class Server{
    Queue[] q;
    int crt;
    public Server (int n){
        q = new Queue[n];
        for (int i = 0; i<q.length; i++)
            q[i] = new Queue();
        crt = 0;
    }
    public void place(Request r) {
        q[crt].enqueue(r);
        crt = (crt + 1)% q.length;
    }
```

```
    public void process() {
        int max = q[0].size();
        int indmax = 0;
        for (int i = 1; i <q.length; i++) {
            if (q[i].size() > max) {
                max = q[i].size;
                indmax = i;
            }
        }
        if (q[indmax].size()!=0) (q[indmax].dequeue()).process();
        return;
    }
}
```

6. [20 points] **Binary trees** Prove by induction that a binary tree with n nodes has $n-1$ edges. If you want a bigger challenge, prove the same problem but for a general tree.

   Solution: We do here the general case.

   Base case: The tree has 1 node, no edges, and the statement holds.

   Induction step: The root of the tree has k children; each child i is at the root of a subtree with number of nodes n_i. Hence the number of nodes in the tree is: $n = 1 + \sum_{i=1}^{k} n_i$. By the induction hypothesis, the number of edges in a subtree will be $n_i - 1$. Hence the number of edges in the entire tree is:

$$k + \sum_{i=1}^{k}(n_i - 1) = k + (\sum_{i=1}^{k} n_i) - k = n - 1$$

   which concludes the proof.

7. [50 points] **Binary search trees**

   Recall that in a binary search tree, at every node, all elements to the left of the node have a smaller key, and all elements to the right of a node have a larger key.

   (a) [30 points] Write in pseudocode an algorithm which, given a binary search tree, will print all the elements bigger than a specified value, min. Make your algorithm as efficient as possible.

   Solution: Algorithm PrintElements(BinarySearchTreeNode node, int min)
   Input: A binary search tree node, representing the root of a BST
   Output: Does not return anything, but prints all node values that are bigger than min
   if (node == **null**) return;
   if (min< node.key) **then**
   print(node.key)
   PrintElements(node.left, min)
   endif
   PrintElements (node.right, min)

(b) [10 points] What is the best possible running time for your algorithm, as a function of the number of nodes in the tree?

Solution: In the best case, the algorithm only recurses on the right side, all the way to the leaves. If node.right is null and min is greater than the key, then the algorithm terminates immediately, so it runs in $O(1)$.

(c) [10 points] What is the worst possible running time for your algorithm, as a function of the number of nodes in the tree?

Solution: The worst case occurs when min is smaller than the smallest element in the tree. In this case, all keys need to be printed, and the algorithm runs in $O(n)$, where n is the number of nodes in the tree.

8. [30 points] **Secret Santa**

Suppose that you and your friends decided to exchange secret gifts this year. The usual procedure for this is to put the names of all the people in the group into a bag, then draw names at random (if you draw your own name, you put the note back). We will call "Santa assignment problem" the problem of finding an assignment of Santas so that everyone receives exactly one gift. We will use graphs to model this problem. Suppose that n friends are in the group. We will build a graph with n vertices, and put a directed edge (i, j) if person i is willing to give a gift to person j.

(a) Formulate "Santa assignment" as a problem in this graph

Solution: We have a directed graph and want to visit each node exactly once. This is the Hamilton cycle problem.

(b) Suppose that any person is willing to give a gift to any other person. Does this problem always have a solution? If yes, give a solution example. If no, explain why not.

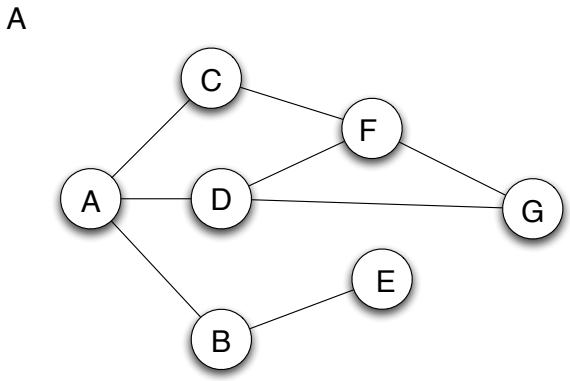
Solution: Yes, we just pick any permutation of nodes (e.g. 1 - 2 - . . . n - 1) and we have a valid solution. This requires linear time.

(c) Suppose that some of the people are not willing to give a gift to everyone else, but only to a strict subset of the people in the group. In this case, is there always a Santa assignment? Justify your answer.

Solution: No, not all graphs have a Hamilton cycle. As a trivial example, suppose you have 2 people, and 1 is willing to give a gift to 2 but 2 does not want to give to 1, then there is no solution.

9. [20 points] **Search in graphs**

Consider the graph in the figure below.



(a) Suppose that you are using breadth-first search to find a path from vertex A to vertex G. Show the state of the queue after each node expansion, assuming nodes are enqueued in alphabetical order. What path do you find?

Solution: We enqueue A, then dequeue it, mark it and get: B C D
Now dequeue B, mark it, and enqueue its successors, so the queue becomes: C D E
Dequeue C, mark it, enqueue its successors, so we get: D E F
Dequeue D, mark it, enqueue its successors, so: E F(C) F(D) G(D) (where we annotated for clarity where the two Fs came from)
Dequeue E: F(C) F(D) G(D)
Dequeue F, mark it, enqueue its successors, so: F(D) G(D) G(F C)
Dequeue F, but because it's marked we do nothing.
Finally, dequeue G. We obtain the path A - D - G (which is the shortest path to G).

(b) Suppose that you are using depth-first search for the same task. Show the node expansion, assuming that successors of the same node are investigated in alphabetical order. What path do you find?

Solution: We go A - B - E, but because this is a dead end, we then investigate A - C - F - D - G (and this is the path that gets returned).

10. [40 points] **Using data structures**

Suppose that you are hired by a big music streaming company to build a recommendation engine. The company has about 10 million songs. Each song has a title, artist, year, album and genre (there are 50 genres in total). There are roughly 2 million users, and each user has a unique identifier, gender, age group, zip code (to encode geographic location) and country. The company wants you to build "user profiles" which can be used to recommend songs to users. The idea is that if we want to recommend a song to user u, we have to find similar users, and choose songs to which the similar users listened.

(a) Your boss suggests that you should maintain the user profile at the level of the songs: for each user, for each song in the data base, record how many times the user has listened to this particular song. Propose a data structure that you could use to maintain this information.

Solution: A 2D array, indexed by users and songs, containing the counts. Note that most elements in the array will be 0 (so it is very sparse).

(b) You secretly think that it would be better to maintain user profiles by genre rather than song (for each user, how many times they played songs from this particular genre. How would you record this information, and what data structure would you use? How could you use such a profile to find songs that you can recommend?

Solution: For each user, you will have an array of 50 elements, corresponding to the genres, and holding the listening count for each genre. The other data structure you need is an array of lists (or some other collection) which holds the songs of each genre. When you want to recommend a song, you can choose the genre with the highest count and pick a song at random from that collection, or you can choose the genre with probability proportional to the count then choose a song of that genre at random.

(c) A different group in the company is developing a method that compares users. For users u and v, u.distance(v) will return a number which tells how similar u and v are. Suppose that this method only uses the user's personal information. Describe a way of efficiently organizing the data, so that for a new user, you can quickly find similar users.

Solution: You can use a tree, in which users get grouped by age. At each leaf, you would have another tree grouped by e.g. geographic location, etc.

(d) You think that a better way of comparing users is by looking at their profile information. In this case, the profiles change all the time. Can you still organize the data efficiently so that for a user u, you can quickly find similar users? Justify your answer

Solution: In this case, you could still use a tree, e.g. grouped by genre, but the tree would need to be re-built periodically as more information about users becomes available.