STUDENT NAME: _____

STUDENT ID: _____

McGill University
Faculty of Science
School of Computer Science

FINAL EXAMINATION

COMP-250: Introduction to Computer Science - Winter 2008

April 29, 2008
2:00-5:00

Examiner: Prof. Doina Precup

Solutions

Write your name at the top of this page. Answer directly on exam paper. Three blank pages are added at the end in case you need extra space. There are 11 questions worth 100 points in total. The value of each question is found in parentheses next to it. Please write your answer on the provided exam. Partial credit will be given for incomplete or partially correct answers.

SUGGESTIONS: READ ALL THE QUESTIONS BEFORE YOU START! THE NUMBER OF POINTS IS NOT ALWAYS PROPORTIONAL TO THE DIFFICULTY OF THE QUESTIONS. SPEND YOUR TIME WISELY!

GOOD LUCK!

1. [15 points, 3 points each] **True or false?**

   Indicate whether the following statements are true or false. Give a two-line justification for each. Credit will be given only if the justification is correct.

   (a) $f(n) = 1000n + 5$ is $O(n^2)$

   Solution: True (for example you can use the limits theorem to prove it).

   (b) All sorting algorithms are $O(n \log n)$.

   Solution: False - selection sort (and several others) are $O(n^2)$.

   (c) Suppose that some algorithm A has $O(n^3)$ and algorithm B has $O(n \log n)$. Then B is always preferred to A.

   Solution: False. There are other considerations in practice, such as the amount of memory used, as well as the size of the constants that are buried in the $O()$ notation. If n is not expected to be very big and the constants are very big, then A may run faster. Additionally, the average case of A may be better than the average case of B.

   (d) There exists a polynomial-time algorithm for deciding if there is an assignment of truth values to the variables in a Boolean expression such that the expression is true.

   Solution: No (this is the most famous NP-complete problem, called satisfiability).

   (e) In Java, if class C *extends* class B, then any method from class B can be called on an object of class C.

   Solution: True (essentially class B is included in class C, with all its methods.

2. [10 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 $j = 1$ **to** i **do**
   print(j)
   $i \leftarrow i + 2$

   Solution: The outer loop executes $n/2$ times. The inner loop executes first 1 time, then 3 times, ... all the way to n times. Hence, the running time is:

   $$1 + 3 + 5 + \ldots + (2k - 1) + 2k + 1$$

   where $2k + 1$ is the largest odd integer $\leq n$. By adding the first and last number, second and second to last etc, we get:

   $$n/2 * (2k + 2) = n * n/2$$

   which yields $O(n^2)$.

(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 executes $n/2$ times. At the exit, $i = n+1$, so the second loop will execute $n+1$ times. Hence, both loops have $O(n)$. Since they are in a sequence (not nested!) the whole algorithm is $O(n)$

(c) **Algorithm** f3(n)
if $n = 0$ **return**
print(n)
f3($n/100$)

Solution: This algorithm is $O(\log n)$. This is a standard pattern, when we have just one recursive call, and the variable gets divided by something. To convince ourselves, we can write the recurrence for the running time of this method, $T(n)$:

$$T(n) = c + T\left(\frac{n}{100}\right) = c + c + T\left(\frac{n}{100^2}\right) = 2c + T\left(\frac{n}{100^2}\right) = \ldots = k * c + T\left(\frac{n}{100^k}\right)$$

The argument of the last term will become 0 (which stops the recursive calls) once $k > \log_{100} n$. At this point, we get:

$$T(n) = c * \log_{100} n + c$$

so the coplexity is $O(\log n)$.
(Recall that we can change the base of the log as follows:

$$\log_b(x) = \log_b(c)\log_c(x) = \frac{\log_c(x)}{\log_c(b)}$$

This means that in the class $O(\log n)$ we have logarithms of all bases, as the conversion between bases just gives a multiplicative constant).

3. [10 points] **Java**

Consider the following piece of Java code. Write next to every printing statement in the main function what its output will be.

```
public class MyParentClass {
   int x;
   public MyParentClass() { x=1; }
   public int get_x() { return x; }
   public void add_to_x(int y) {
      x=x+y;
   }
}

public class MyChildClass extends MyParentClass {
   public MyChildClass() { x=2; }
   public void new_add_to_x(int y) {
      super.add_to_x(y);
      x=x+1;
   }
   public void another_add_to_x(int y) {
      super.add_to_x(y);
      y=10;
   }
}

   public static void main (String[] args) {
      int y=3;

      MyParentClass obj = new MyParentClass();
      System.out.println(obj.get_x()); //prints 1
      obj.add_to_x(y);
      System.out.println(obj.get_x());  //prints 4

      MyChildClass c = new MyChildClass();
      System.out.println(c.get_x()); //prints 2
      c.add_to_x(y);
      System.out.println(c.get_x()); //prints 5
      c.new_add_to_x(y);
      System.out.println(c.get_x());   //prints 9
      c.another_add_to_x(y);
      System.out.println(c.get_x());  //prints 13
      System.out.println(y); // prints 3
   }
```

4. [10 points] **A simple algorithm**

   You are given an array of n positive integers. The numbers do not appear in any particular order. Write an algorithm that will make the array "wiggly", i.e. arrange the elements in such a way that $a[1] \leq a[2] \geq a[3] \leq a[4]$ etc. In other words, a node with an even index must be greater than or equal to its two neighbors, and a node with odd index must be less than or equal to its neighbors. You may call as a subroutine any algorithm we discussed in class. State and justify what $O()$ is for your algorithm.

   Solution: The idea is to sort in increasing order, then swap $a[2k]$ with $a[2k+1]$ until the end of the array. The pseudocode is as follows (ssuming the indexing starts at 1):

   Algorithm Wiggly(a, n)
   Input: An array a of positive integers of size n
   Output: The array elements will be rearrange such that $a[i-1] \leq a[i]$ and $a[i] \geq a[i+1]$, for all even i

   MergeSort(a,n)
   $i \leftarrow 2$
   while $i \leq n$ **do**
   Swap($a[i-1]$, $a[i]$);
   $i \leftarrow i+2$

   The time complexity is $O(n \log n) + O(n)$ (from sorting and from the loop, respectively), which yields $O(n \log n)$.

5. [5 points] **Sorting and Linked Lists**

   Suppose that you are asked to provide a sorting algorithm for doubly linked lists which works *without copying the list into an array*. What sorting algorithm would you use, and what would $O()$ be? Justify your answer (no pseudocode necessary).

   Solution: Since you are not allowed to copy the list into an array, you cannot use algorithms that need to index directly particular elements. Most likely you would use selection sort, because it is easy to find the minimum or maximum and swap it in the appropriate position in the list. This can all be done by changing the references of elements, and the complexity will be $O(n^2)$.

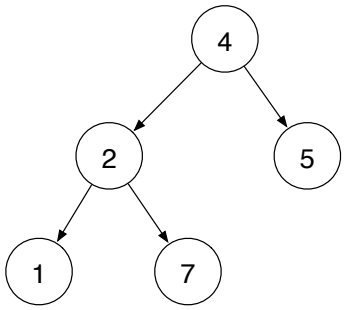
6. [10 points] **Binary trees**

Consider a BinaryTreeNode data structure with three attributes:

- content - an integer
- left - reference to a BinaryTreeNode representing the left child (null if there is no left child)
- right - reference to a BinaryTreeNode representing the right child (null if there is no right child)

Write an algorithm in pseudocode that takes as input a BinaryTreeNode n and an integer s. Your algorithm should return true if there is a path from n to the leaves such that the sum of the node content along the path is equal to s, and false otherwise.

For example, for the tree below, when called with a reference to node 4, your algorithm should return true for s= 9 and false for s= 6.



Solution: We will write a recursive algorithm:

Algorithm hasPathSum (BinaryTreeNode node, int s)
Input: A BinaryTreeNode node and an integer s
Output: true if there is a path from root to leaves in the tree such that the sum of the nodes in the path is s, false otherwise

if (node = **null**) **then return false**;
if ((node.left = **null**) || (node.right = **null**)) **then**
    return ((s =node.content) || hasPathSum(node.left,s-node.content) || hasPathSum(node.right,s-node.content));
return (hasPathSum(node.left,s-node.content) || hasPathSum(node.right,s-node.content));

Let us also discuss $O()$ for this algorithm (though this was not required in the question). Based on the algorithm above, we can write a recurrence for the running time on a tree of n nodes, $T(n)$, as:

$$T(n) = c + T(k) + T(n-k-1)$$

where k is the number of nodes in the left sub-tree. The worst-case scenario here is that at each recursive call, we have a 1:(n-2) split between the number of nodes in each sub-tree (very similar

situation to Quicksort). Expanding the recurrence using the worst-case at each step we get:

$$\begin{aligned} T(n) &= c + T(1) + T(n-2) = 2c + T(n-2) \\ &= 2c + (c + T(1) + T(n-4)) = 2c + T(n-4) = \ldots \\ &= k * c + T(n-k) \end{aligned}$$

The recursion will finish (hitting the base case) when $k = n$, and we will end up with:

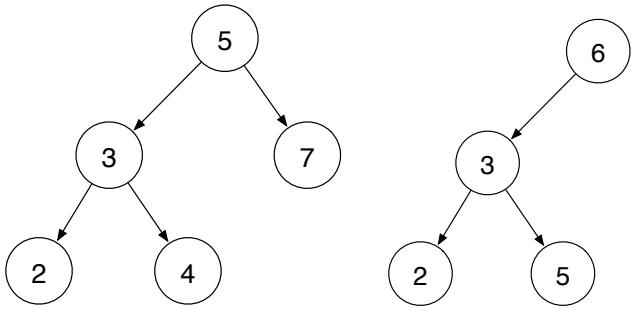
$$T(n) = nc + c$$

which leads to complexity $O(n)$, where n is the number of nodes in the tree.

7. [10 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. Write in pseudocode an algorithm which, given a binary search tree with at least two nodes, will return the element with the node with the *second largest* key in the tree.

   For example, for both of the trees below, the algorithm should return 5.



   Your algorithm should run in $O(h)$, where h is the height of the tree

   Algorithm findSecondMax(BinarySearchTreeNode n)
   Input: A binary search tree node, representing the root of a BST
   Output: A the node in the binary search tree whose key is second largest

   Solution: The main idea is that the biggest element is found by going all the way to the right. Then, if there is a left subtree, go all the way to the right in that subtree. If there is no left subtree, then return the parent. The psedocode is as follows:

   Algorithm: SecondLargest (BSTNode root)
   Input: The root of a binary search tree with at least two nodes
   Output: The second largest element

   BSTNode node$\leftarrow$ root; // we will use this reference to navigate the tree
   while (node.right $\neq$ **null**) **do**

```
        node ← node.right;
if (node.left = null) then return node.parent;
node ← node.left; while (node.right ≠ null) do
        node ← node.right;
return node;
```

8. [5 points] **Heaps**

   In the heaps we studied in class, one could retrieve the minimum element in $O(1)$. Now suppose that instead you want to retrieve the *maximum* element in $O(1)$. What properties should such a heap have?

   Solution: Heap condition should be flipped (the root of any sub-tree has to be bigger than all other nodes), otherwise everything is the same.

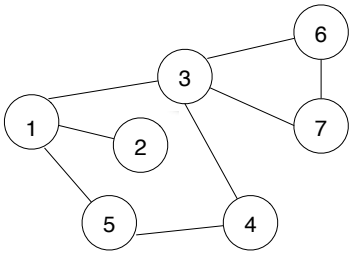
9. [5 points] **Hash tables**

   Suppose you are hired to build an application in which you need to store data about millions of company customers in a hash table. You have no idea what would be a good hashing function, but you look on the internet and find two promising candidates. However, you have no idea which function would work best. Describe an approach which can take advantage of both functions. You do not need to write pseudocode, but your description should be precise.

   Solution: The simplest approach is to use a hash table in which the buckets are also hash tables. You use the first function for the main hash table, the second one for the buckets.

10. [10 points] **Search in graphs**

    Consider the graph in the figure below.



    (a) [4 points] Suppose that you are using breadth-first search to find a path from vertex 1 to vertex 7. Show the state of the queue after each node expansion, assuming nodes are enqueued in increasing order. What path do you find?

    Solution: First we enqueue 1. Next we dequeue it and enqueue its successors, which gives the queue:

    2 3 5

    (the head of the queue is the leftmost element). We dequeue 2 but it has no unvisited successors. We dequeue 3 and enqueue its successors, which yields:

    5 4 6 7

Now we dequeue 5 and enqueue its successor, which is 4:

4 6 7 4

Note that the two copies of 4 in the queue correspond to the two paths (through 3 and through 5). We dequeue 4, but its neighbors have already been visited. The we dequeue 6 and enqueue its unvisited successor, 7:

7 4 7

Finally we dequeue 7, and we have obtained the desired path: 1-3-7.

(b) [4 points] 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 increasing order. What path do you find?

Solution: We go 1-2, then this is a dead end so we go back to 1 and try 1-3-4-5-1. At this point, if we marked the nodes where we passed, we will know that this path went in a circle, so we backtrack to 3 and go from there to 6 and 7. The path obtained is 1-3-6-7

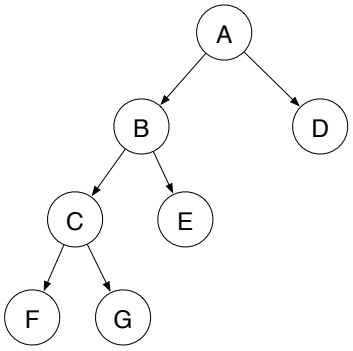
(c) [2 points] Based on this example, what is the main advantage of depth-first search? What is the main advantage of breadth-first search?

Solution: Depth-first search is more memory efficient, as it only keeps in mind one path (and that is stored implicitly through the recursion stack). Breadth-first search, on the other hand, always returns the shortest path (in this case, its solution is better than depth-first search.

11. [10 points] **Proof by induction**

Recall that a binary tree is called *proper* if every node has either 2 or 0 descendents. Suppose that you have a proper binary tree with n internal nodes. Let $PI(n)$ be the sum of the length of the paths between the root and all the internal nodes, and $PL(n)$ be the sum of the length of the paths between the root and all the leafs.

For example, in the tree below, A, B and C are internal nodes, $PI(n) = 0+1+2 = 3$ and $PL(n) = 1+2+3+3 = 9$.



Prove by induction that $PL(n) - PI(n) = 2n$.

Solution:

Base case: one leaf: $n = 0$, and both sums are 0.

One more case: $n = 1$ (one internal node, 2 leaves), $PI = 0, PL = 2$ so the relationship is satisfied

Induction step: Take a tree with $n+1$ nodes, cut out its root. Note that the number of internal nodes in this tree is $k+(n-k)+1$, where k and $n-k$ are the number of nodes on the left and the right respectively. In the left sub-tree, we have, by induction hypothesis:

$$PL(k) - PI(k) = 2k$$

In the right sub-tree:

$$PL(n-k) - PI(n-k) = 2(n-k)$$

Note that all paths in both sub-trees get a 1 added to them in the big tree, plus we have 2 more paths of length 1. The 1s added onto the existing paths cancel out. Putting all together, we get:

$$PL(n+1) - PI(n+1) = 2k+2(n-k)+2 = 2(n+1)$$