SYSC 2100, Fall 2005 Final Exam Solutions
Instructor: T. Kunz

Question 1: Big-O Notation (10 marks)

For each of the following functions f, where n = 0, 1, 2, 3, ..., estimate f using **Big-O notation and plain English**. Trivially, $O(2^n)$ is a valid upper bound for all of these functions, but I am looking for tight upper bounds (i.e., you won't get any marks for trivial upper bounds such as "*$O(2^n)$ or f(n) is exponential*", unless that is indeed a tight upper bound for the specific function).

1. f(n) = (2 + n) * (3 + log(n))

   f(n) is O(n log n). f(n) is linear-logarithmic in n. **(1 mark)**

2. f(n) = 11 * log(n) + n/2 - 3452

   f(n) is O(n). f(n) is linear in n. **(1 mark)**

3. f(n) = 1 + 2 + 3 + ... + n

   f(n) is O(n2). f(n) is quadratic in n. **(1 mark)**

4. f(n) = n * (3 + n) - 7 * n

   f(n) is $O(n^2)$. f(n) is quadratic in n. **(1 mark)**

5. f(n) = 7 * n + (n - 1) * log (n - 4)

   f(n) is O(n log n). f(n) is linear-logarithmic in n. **(1 mark)**

6. f(n) = log (n^2) + n

   f(n) is O(n). f(n) is linear in n. **(1 mark)**

7. $f(n) = \dfrac{(n + 1) * \log (n + 1) - (n + 1) + 1}{n}$

   f(n) is O(log n). f(n) is logarithmic in n. **(2 marks)**

8. f(n) = n + n/2 + n/4 + n/8 + n/16 + ...

   f(n) is O(n). f(n) is linear in n. **(2 marks)**

Question 2: Recursion (10 marks)

1. Given two positive integers *i* and *j*, the greatest common divisor of *I* and *j*, written *gcd (i, j)* is the largest integer *k* such that (i % k = 0) and (j % k = 0). For example, gcd (35, 21) = 7 and gcd (8, 15) = 1. Develop a recursive method that returns the greatest common divisor of *i* and *j*. Here is the method specification:

```
/**
* Finds the greatest common divisor of two given positive integers
*
* @param i – one of the given positive integers.
* @param j – the other given positive integer.
*
* @return the greatest common divisor of iand j.
*
* @throws IllegalArgumentException – if either i or j is not a positive integer.
*
*/
public static int gcd (int i, int j)
```

Hint: According to Euclid's algorithm, the greatest common divisor of *i* and *j* is *j* if *i % j* = 0. Otherwise, the greatest common divisor of *i* and *j* is the greatest common divisor of *j* and *(i % j)*.

Answer (4 marks):

```
public static int gcd (int i, int j)
{
        if (i <= 0 || j <= 0)
                throw new IllegalArgumentException();
        if (i % j == 0)
                return j;
        return gcd (j, i % j);
} // method gcd
```

2. A ***palindrome*** is a string that is the same from right-to-left as from left-to-right. For example, the following are palindromes: ABADABA, RADAR, OTTO, MADAMIMADAM, EVE
For the time being, we restrict each string to upper-case letters only. Develop a method that uses recursion to test for palindromes. The only input is a string that is to be tested for palindromehood. The method specification is

```
/**
* Determines whether a given string of upper-case letters is a
* palindrome. A palindrome is a string that is the same from right-to-
* left as from left-to-right.
*
* @param s – the given string
*
* @return true – if the string s is a palindrome. Otherwise, return false.
*
*/
public static boolean isPalindrome (String s)
```

Answer (6 marks):

You can do this best by implementing a helper method that takes a string and two indices (a starting index i and an ending index j). If i >= j, the substring of s at indexes i through j is a (trivial) palindrome. Otherwise, that substring of s is a palindrome if and only if s.charAt (i) = s.charAt (j) and the substring of s at indexes i + 1 through j – 1 is a palindrome.

```
public boolean isPalindrome (String s)
{
    return isPalindrome (s, 0, s.length() - 1);
} // method isPalindrome

/**
* Determines whether the substring of given string of upper-case
* letters is a palindrome. A palindrome is a string that is the same
* from right-to-left as from left-to-right.
*
* @param s – the given string
* @param i – the starting index of the substring of s.
* @param j – the ending index of the substring of s.
*
* @return true – if the substring of s between indexes I and j
* inclusive is a palindrome. Otherwise, return false.
*
* @throws IllegalArgumentException – if either i or j is a negative
* integer.
*
*/
public boolean isPalindrome (String s, int i, int j)
{
    if (i >= j)
        return true;
    if (s.charAt (i) != s.charAt (j))
        return false;
    return isPalindrome (s, i + 1, j - 1);
} // method isPalindrome
```

Question 3: Linked Lists (10 marks)

One of the possibilities for fields in the LinkedList class is to have head and tail fields (pointing to the first and last element in the linked list, respectively), both of type Entry, where the Entry class had element and next fields, but no previous field. Then we would have a *singly-linked* list.

1. Define the addLast method for this design. Here is the method specification:

```
/**
* Appends a specified element to (the back of) this LinkedList object.
*
* @param element – the element to be appended.
*
* @return true.
*
*/
public boolean addLast (Object element)
```

Answer (3 marks):

```
public boolean addLast (Object element)
{
        Entry newEntry = new Entry( );
        newEntry.element = element;
        newEntry.next = null;
        if (head == null)
                head = newEntry;
        else
                tail.next = newEntry;
        tail = newEntry;
        return true;
} // method addLast
```

2. The definition of the removeLast() method would need to make **null** the next field in the Entry object before the Entry object tail. Could we avoid a loop in the definition of removeLast() if, in the LinkedList class, we added a beforeTail field that pointed to the Entry object before the Entry object tail? Explain.

Answer (3 marks):

A loop would still be needed because the beforeTail field would have to be updated to point to its predecessor entry.

3. Hypothesize the output from the following method segment:

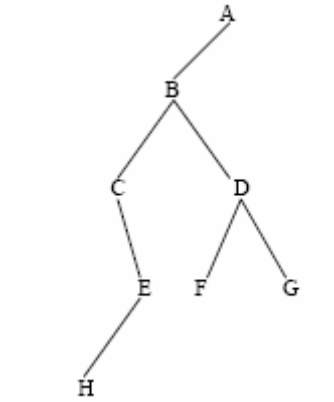
```
LinkedList letters = new LinkedList();
ListIterator itr = letters.listIterator();
itr.add ('f'); itr.add ('t');
itr.previous();
itr.previous();
itr.add ('e');
itr.add ('r');
itr.next();
itr.add ('e');
itr.add ('c');
itr = letters.listIterator();
itr.add ('p');
System.out.println (letters);
```

Answer (4 marks):

The output is [p, e, r, f, e, c, t]

Question 4: Binary Trees (10 marks)

Answer the questions below about the following binary tree:



1. What is the root element?
   Answer (0.5 marks): A

2. How many leaves are in the tree?
   Answer (0.5 marks): 3

3. What is the height of the tree?
   Answer (0.5 marks): 4

4. What is the height of the left subtree?
   Answer (0.5 marks): 3

5. What are the descendants of B?
   Answer (1 mark): C, D, E, F, G, and H

6. What are the ancestors of F?
   Answer (1 mark): A, B, and D

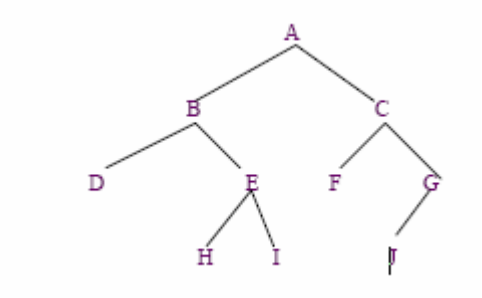
7. What would the output be if the elements were written out during an inOrder traversal?
   Answer (1 mark): C, H, E, B, F, D, G, A

8. What would the output be if the elements were written out during a postOrder traversal?
   Answer (1 mark): H, E, C, F, G, D, B, A

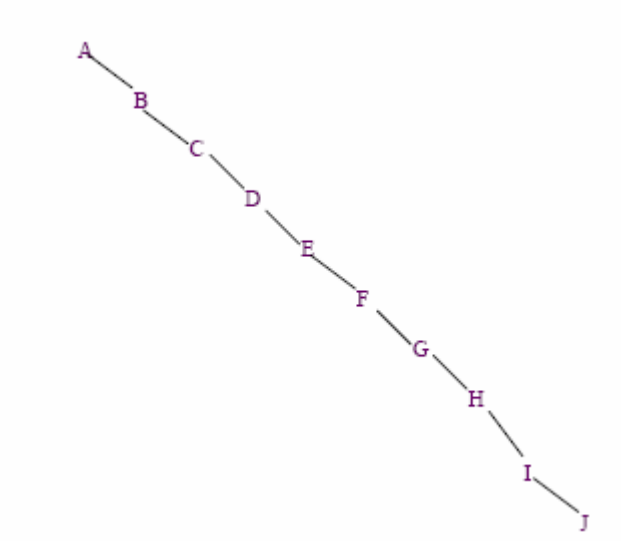
9. What would the output be if the elements were written out during a preOrder traversal?
   Answer (1 mark): A, B, C, E, H, D, F, G

10. What would the output be if the elements were written out during a breadth-first traversal?
    Answer (1 mark): A, B, C, D, E, F, G, H

11. What is the maximum number of leaves possible in a binary tree with 10 elements? Construct such a tree.
Answer (1 mark): 5



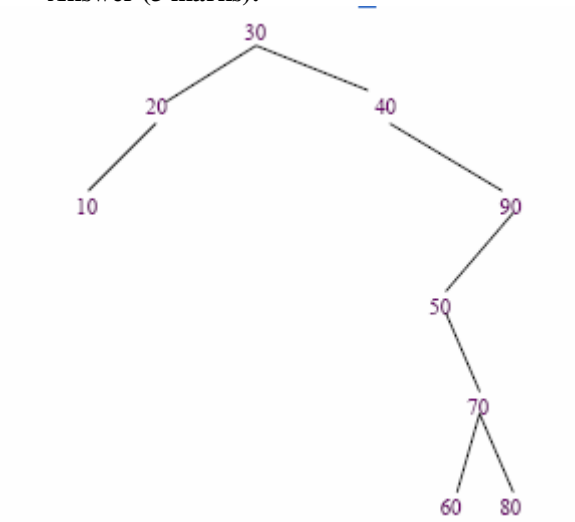
12. What is the minimum number of leaves possible in a binary tree with 10 elements? Construct such a tree.
Answer (1 mark): 1 (in case of a degenerate tree that is in essence a linked list)



Question 5: Binary Search Trees/Huffman Trees (10 marks)

1. Show the effect of making the following insertions into an initially empty binary search tree: 30, 40, 20, 90, 10, 50, 70, 60, 80

Answer (3 marks):



2. Suppose we decide to implement the PurePriorityQueue interface with a TreeSet object:

```
public class TreeSetPriorityQueue
implements PurePriorityQueue
{
        TreeSet pq;
        …
```

Then the definitions of the getMin, removeMin and add methods are oneliners. For example, here is the definition of the getMin method:

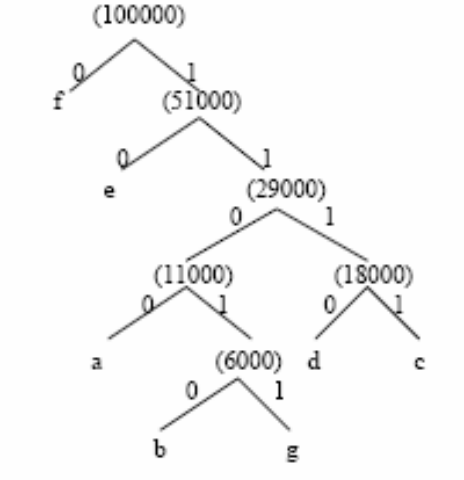
```
        public Object getMin()
        {
                return pq.first();
        } // method getMin
```

Estimate worstTime(n) for those three methods. What is the major drawback to implementing the PurePriorityQueue interface with a class that implements the Set interface? Explain how to overcome this drawback by implementing the PurePriorityQueue interface with the TreeMap class instead.

Answer (4 marks):

For the getMin, removeMin and add methods, worstTime(n) is logarithmic in n. The major drawback is that duplicate elements are not allowed. So if an element has already been inserted, any element equal to that inserted element cannot be inserted. With a TreeMap, the value would contain the number of occurrences of the key. Then duplicate elements would be handled, but maintaining this count would slow down the methods slightly.

3. Use the following Huffman tree to translate the bit sequence 11101011111100111010 back into letters 'a' ... 'g':



Answer (3 marks): decade

Question 6: Hashing (10 marks)

1. Why does the HashMap class use singly-linked lists instead of the LinkedList class?

   Answer (2 marks):
   The singly-linked (instead of doubly-linked) lists are used to save space. Each Entry object in a (doubly) LinkedList object has a previous field. The space for that field would be wasted because that field would not be used in any HashMap (or Entry or HashIterator) method.

2. Suppose you have a HashMap object, and you want to insert an element unless it is already there. How could you accomplish this? **Hint:** The put method will insert the element even if it is already there (in which case, the new value will replace the old value).

   Answer (2 marks):

   ```
   if (!myMap.containsKey (myKey))
           myMap.put (myKey, myValue);
   ```

3. For each of the following methods, estimate averageTime(n) and worstTime(n):
   - making a successful call – that is, the element was found – to the *contains* method in the LinkedList class;

     Answer (1 mark):
     Both averageTime(n) and worstTime(n) are linear in n.

   - making a successful call to the *contains* method in the ArrayList class;

     Answer (1 mark):
     Both averageTime(n) and worstTime(n) are linear in n.

   - making a successful call to the generic algorithm *binarySearch* in the Arrays class; assume the elements in the array are in order;

     Answer (1 mark):
     Both averageTime(n) and worstTime(n) are logarithmic in n.

   - making a successful call to the *contains* method in the BinarySearchTree class;

     Answer (1 mark):
     The averageTime(n) is logarithmic in n, and worstTime(n) is linear in n.

   - making a successful call to the *contains* method in the TreeSet class;

     Answer (1 mark):
     Both averageTime(n) and worstTime(n) are logarithmic in n.

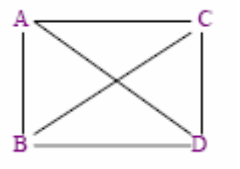
   - making a successful call to the *contains* method in the HashSet class. You should make the Uniform Hashing Assumption. Estimate averageTime(n, m) and worstTime(n, m).

     Answer (1 mark):
     The averageTime(n, m) is constant, and worstTime(n, m) is linear in n.

Question 7: Graphs, Trees, and Networks (20 marks)

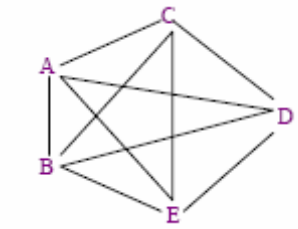
1. Draw an undirected graph that has four vertices and as many edges as possible. How many edges does the graph have?

   Answer (1 mark):
   The graph has six edges



2. Draw an undirected graph that has five vertices and as many edges as possible. How many edges does the graph have?

   Answer (1 mark):
   The graph has 10 edges



3. What is the maximum number of edges for an undirected graph with *V* vertices, where *V* is any non-negative integer?

   Answer (2 marks): V * (V-1) / 2

4. What is the maximum number of edges for a directed graph with V vertices?

   Answer (2 marks): V * (V-1)

5. Prim's algorithm (getMinimumSpanningTree), Dijkstra's algorithm (getShortestPath), and Huffman codes are ***greedy***: the locally optimal choice has the highest priority. In these cases, greed succeeds in the sense that the locally optimal choice led to the globally optimal solution. Do all greedy algorithms succeed for all inputs? In this question we look at coin-changing algorithms. In one situation, the greedy algorithm succeeds for all inputs. In the other situation, the greedy algorithm succeeds for some inputs and fails for some inputs. Suppose you want to provide change for any amount under a dollar using as few coins as possible. Since "fewest" is best, the greedy (that is, locally optimal) choice at each step is the coin with largest value whose addition will not surpass the original amount. Here is a method to solve this problem:

```
/**
* Prints the change for a given amount, with as few coins (quarters,
* dimes, nickels and pennies) as possible.
*
* @param amount – the amount to be given in change.
*
*/
public static void printFewest (int amount)
{
        int coin[ ] = {25, 10, 5, 1};
        final String RESULT = "With as few coins as possible, here is the change for ";
```

```
        System.out.println (RESULT + amount + ":");
        for (int i = 0; i< 4; i++)
                while (coin [i] <= amount)
                {
                        System.out.println (coin [i]);
                        amount – = coin [i];
                } // while
} // printFewest
```

For example, suppose that amount has the value 62. Then the output will be 25, 25, 10, 1, 1. Five is the minimum number of coins needed to make 62 cents from quarters, nickels, dimes and pennies. Give an example to show that a greedy algorithm is not optimal for all inputs if nickels are not available. That is, if we have

```
int coins[ ] = {25, 10, 1};
…
for (int i = 0; i < 3; i++)
...
```

then the algorithm will not be optimal for some inputs.

Answer (4 marks):
If amount = 30, the algorithm will return 25, 1, 1, 1, 1, 1 instead of the minimum of 10, 10, 10.

6. In the Network class, develop a method to produce a topological order. Here is the specification:

```
/**
* Sorts this acyclic Network object into topological order.
* The averageTime(V, E) is linear in V + E.
*
* @return an ArrayList object of the vertices in topological order. Note: if
* the size of the ArrayList object is less than the size of this
* Network object, the Network object must contain a cycle, and
* the ArrayList will not be in topological order.
*
*/
public ArrayList sort()
```

Hint: First, construct a HashMap object, inCount, which maps each vertex w to the number of vertices to which w is adjacent. For example, if the Network object has three edges of the form <?, w>, then inCount will map w to 3. After inCount has been filled in, push onto a stack (or enqueue onto a queue) each vertex that inCount maps to 0. Then loop until the stack is empty. During each loop iteration,

1. pop the stack;
2. append the popped vertex v to an ArrayList object, orderedVertices;
3. decrease the value, in inCount's mapping, of any key w such that <v, w> forms an edge;
4. if inCount now maps w to 0, push w onto the stack.

After the execution of the loop, the ArrayList object, containing the vertices in a topological order, is returned.

Answer (10 marks):

```
/**
* Sorts this acyclic Network object into topological order.
* The averageTime(V, E) is linear in V + E.
*
* @return an ArrayList object of the vertices in topological order. Note: if
* the size of the ArrayList object is less than the size of this
* Network object, the Network object must contain a cycle, and
* the ArrayList will not be in topological order.
*
*/
```

```
public ArrayList sort()
{
    HashMap intCount = new HashMap();
    int inDegree, count;
    for (Vertex w : this)
    {
            inDegree = 0;
            LinkedList list;
            for (Vertex vertex : this)
            {
                    list = adjacencyMap.get (vertex);
                    for (VertexWeightPair pair : list)
                            if (w.equals (pair.getVertex()))
                                    inDegree++;
            } // checking to see vertices that are adjacent to w
            intCount.put (w, inDegree);
    } // for all vertices w

    ArrayList orderedVertices = new ArrayList (size());
    PureStack stack = new LinkedListPureStack ();
    for (Vertex vertex : this)
            if (intCount.get (vertex) == 0)
                    stack.push (vertex);
            Vertex v;
            while (!stack.isEmpty())
            {
                    v = stack.pop();
                    orderedVertices.add (v);
                    for (Vertex w : this)
                            if (containsEdge (v, w))
                            {
                                    count = intCount.get (w);
                                    intCount.put (w, --count);
                                    if (count == 0)
                                            stack.push (w);
                            } // if v,w is an edge
            } // while
    return orderedVertices;
    } // method sort
```