CSC 225: Assignment 2 Part B, Written Exercises

Due at the beginning of class on Monday Oct. 16, 2017

Instructions for all assignments:

1. Draw boxes for your marks on the top of the first page of your submission. Place a 0 in the corresponding box for any questions you omit. For this assignment:

Question	1	2	3	4	5	6
Marks						

2. Questions should be **in order**.
3. Show your work unless otherwise stated.
4. Put your name (last name underlined) and student number on all submissions.

Learning objectives

This assignment is an introduction to algorithm analysis. It reinforces skills presented on assignment #1 (setting up and solving recurrences, and induction) as tools for algorithm analysis.

1. This question refers to the implementation of a *bigInteger* from Assignment #1. I added three methods to the *LinkedList* class so that *bigInteger* values can be compared (given below). Let x be a *bigInteger* value with n digits and let y be a *bigInteger* with $n + k$ digits for some $k \geq 0$. The *compare* method is called like this:
   int cmp= x.compare(y);

   (a) [2] Give an example of two big integers (leading zeroes are allowed) that represent a worst case example for the running time of the compare method when $n = 8$ and $k = 7$.

   (b) [3] Give a description of a family of worst case examples (expressed in terms of n and k) such that on your examples, the number of calls to *compareDigit* is maximized.

   (c) [4] How much time (in terms of n and k) does the *nonZeroDigit* method take on your worst case examples? Justify your answer.

   (d) [4] Derive a recurrence $T(n)$ which represents the time complexity of the *compareDigit* method on your worst case example. Explain where each part of the recurrence is coming from.

   (e) [4] Solve your recurrence from part (d) by repeated substitution, numbering the steps starting at Step 0.

   (f) [4] Prove that you have the correct solution to the recurrence by induction.

(g) [4] How much time does the *compare* method take in the worst case (in terms of n and k)? Justify your answer. Keep in mind that if k is large, for example $k = n^5$ then the dominant term could depend on k and when k is small, for example, $k = 0$, the dominant term could depend on n. As a result, it is critical that your answer is a function of both n and k.

The methods for Question 1:

```
/*
   This method returns:
   -1 if the bigInteger associated with the method is less than y
    0 if the bigInteger associated with the method is equal to y
   +1 if the bigInteger associated with the method is greater than y
*/
  public int compare(LinkedList y)
  {
     int nv;

//   If one has more digits than the other, check if any
//   of the extra digits are non-zero- if so, the one with
//   more digits is bigger.

     // Block 1

     nv= n; // Change made from first version.
     if (n > y.n)
     {
        if (nonZeroDigit(y.n)) return(1);
        nv= y.n;
     }
     else if (y.n > n)
     {
        if (y.nonZeroDigit(n)) return(-1);
     }

     // Block 2

//   Compare digits starting with most significant ones.

     return(compareDigit(nv, y));
  }
```

```
public boolean nonZeroDigit(int nIgnore)
{
   ListNode current;
   int i;
   current= start;
   for (i=0; i < nIgnore; i++)
      current= current.next;
   while (current != null)
   {
      if (current.data != 0) return(true);
      current= current.next;
   }
   return(false);
}


public int compareDigit(int nv, LinkedList y)
{
   int dx, dy;
   ListNode xcurrent, ycurrent;
   int i;

   if (nv ==0) return(0); // They are equal.

//    Get xcurrent/ycurrent to point to cell nv of x/y.

   xcurrent= start;
   ycurrent= y.start;
   for (i= 1; i < nv; i++)
   {
      xcurrent= xcurrent.next;
      ycurrent= ycurrent.next;
   }
   dx= xcurrent.data;
   dy= ycurrent.data;
   if (dx < dy) return(-1);
   if (dx > dy) return( 1);
   return(compareDigit(nv-1, y));
}
```

The aim of Questions 2-6 is to analyze the time complexities of three divide and conquer methods, *beginMax*, *middleMax*, and *endMax*, for finding a cell with a maximum key value on a linked list. The list can be split at the **beginning** (the first sublist has size 1 and the second one has size $n-1$), or in the **middle** (the first sublist has size $\left\lfloor \frac{n}{2} \right\rfloor$ and the second sublist has size $\left\lceil \frac{n}{2} \right\rceil$), or it can be split at the **end** (the first sublist has size $n-1$ and the second sublist has size 1). The three variants have similar programs. The code given on the last page of this assignment does the split in the middle. To get the other two variants, line B1 which says *last* = *n*/2; is replaced with *last* = 1; (if the split is at the beginning), or *last* = *n* – 1; (if the split is at the end). Also, the conquer steps for *beginMax* and *endMax* in lines C1 and C2 are recursive calls to *beginMax* and *endMax* respectively.

As suggested in the text (pp. 182-184), one approach to estimating algorithm time complexities is to identify an operation (or operations) such that the order of growth of the running time of an algorithm is the same as the order of growth of the function representing the number of times the operation occurs on a given problem size.

To simplify the analysis, instead of choosing a proxy operation, I want you to count the exact number of times that one of the **designated statements** A4, B2, B4, C1, C2, or D1 is executed. This properly accounts for the time complexities of the base case (it takes $O(1)$ time and we count executions of A4), the divide step of the algorithm (the time it takes is proportional to the work done by the loop for which we count executions of B2, plus a constant number of other steps for which we count executions of B4), and the marriage (the work done is in $O(1)$ and we count executions of D1). The recursive calls themselves have $O(1)$ overhead and this is accounted for by counting executions of C1 and C2.

For a list of size n, $B(n)$ will be used to denote the number of times a designated statement is executed for *beginMax*, $M(n)$ will be used to denote number of times a designated statement is executed for *middleMax* and $E(n)$ will be used to denote number of times a designated statement is executed for *endMax*.

2. The program provided is divided into four blocks (A, B, C, and D as indicated on the code provided).

(a) [6] For each block/algorithm variant, fill in this chart with the number of times that a designated statement is executed.

Split at:	Beginning	Middle	End
Block A			
Block B			
Block C			
Block D			

Use floor/ceiling functions in your answer as required. Your answers for block C should be expressed in terms of the functions B, M, and E. Except for block *C*, the chart includes only the number of designated statements executed at the top level of recursion (the others are accounted for in the counts for block C).

(b) [9] Use your table from (a) to set up recurrence relations for $B(n)$, $M(n)$ and $E(n)$.

Note: The correctness of your answers to Question 2-5 depend on getting correct formulas in the chart for Question 2(a). If you are not sure if your formulas are correct, then you can copy my program and run the three algorithm variants with some additional code added to count the number of times each designated statement is executed.

3. $B(n)$ is equal to the number of times *beginMax* executes a designated statement.

(a) [5] Solve your recurrence for $B(n)$ from question 2(b) by repeated substitution to get a closed formula.

(b) [5] Prove by induction that your answer to 3(a) correctly gives the number of times that a designated statement is executed when calling the method with a list of size n for *beginMax*.

(c) [5] Choose a function $f(n)$ that is as simple as possible such that $B(n) \in \Theta(f(n))$ and then prove that $B(n) \in \Theta(f(n))$.

4. $M(n)$ is equal to the number of times *middleMax* executes a designated statement.

(a) [10] Assume that $n = 2^k$ for some integer $k \geq 0$. Solve your recurrence for $M(n)$ from question 2(b) by repeated substitution to get a closed formula.

(b) [10] Assume that $n = 2^k$ for some integer $k \geq 0$. Prove by induction that your answer to 4(a) correctly gives the number of times that a designated statement is executed when calling the method with a list of size n for *middleMax*.

(c) [5] Choose a function $f(n)$ that is as simple as possible such that $M(n) \in \Theta(f(n))$ and then prove that $M(n) \in \Theta(f(n))$.

5. $E(n)$ is equal to the number of times *endMax* executes a designated statement.

(a) [5] Solve your recurrence for $E(n)$ from question 2(b) by repeated substitution to get a closed formula.

(b) [5] Choose a function $f(n)$ that is as simple as possible such that $E(n) \in \Theta(f(n))$ and then prove that $E(n) \in \Theta(f(n))$.

6. Consider the three variants from question 2 for finding a maximum key value on a list.

(a) [2] Which of the three approaches is the fastest?

(b) [2] Which of the three approaches is the slowest?

(c) [6] A student suggests that an alternate proxy operation for this problem is to count accesses to the **data** fields of ListNodes instead. Is this a reasonable alternative for estimating the time complexity of these algorithm variants? Why or why not?

```
    public class LinkedList
    {   int n;
        ListNode start, rear;

        public ListNode middleMax(int level)
        {
            // Block A
A1              LinkedList listA, listB;
A2              ListNode rear1, start2, max1, max2;
A3              int i, last;

A4              if (n==1) return(start);

            // Block B
B1              last= n/2; // beginMax sets last=1, endMax sets last= n-1

B2              rear1= start;
B3              for (i=1; i < last; i++)
B4                  rear1= rear1.next;
B5              start2= rear1.next;
B6              rear1.next= null;
B7              listA= new LinkedList(last, start, rear1);
B8              listB= new LinkedList(n-last, start2, rear);

            // Block C
C1              max1= listA.middleMax(level+1);
C2              max2= listB.middleMax(level+1);

            // Block D
D1              listA.rear.next= listB.start;
D2              if (max1.data > max2.data) return(max1);
D3              else return(max2);
        }
E0      public LinkedList(int size, ListNode first, ListNode last)
        {   // Block E
E1              n= size;
E2              start= first;
E3              rear= last;
        }
```