Test 1: Compsci 100

Owen Astrachan

February 13, 2008

Name: _____ (3 points)

Login: _____

Honor code acknowledgment (signature) _____

	value	grade
Problem 1	12 pts.	
Problem 2	20 pts.	
Problem 3	22 pts.	
Problem 4	28 pts.	
Problem 5	15 pts.	
TOTAL:	97 pts.	

This test has 14 pages, be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts 75 minutes.

In writing code you do not need to worry about specifying the proper `import statements`. Assume that all libraries and packages we've discussed are imported in any code you write.

Unless indicated otherwise, here's the `Node` class for this test.

```
public class Node{
   public String info;
   public Node next;
   public Node(String s, Node link){
       info = s;
       next = link;
   }
}
```

PROBLEM 1 : (*It Depends (12 points)*)

Part A (4 points)

What value is returned by the call `calculate(2043)`? What is the complexity (big-Oh, in terms of N) of the call `calculate(N)`? Briefly justify your answers.

```
public int calculate(int n){
    int prod = 1;
    while (prod < n){
        prod *= 2;
    }
    return prod;
}
```

Part B (8 points)

Consider method `zcal` below, the call `zcal(5)` evaluates to 15.

```
public int zcal(int n){
    if (n == 0) return 0;
    return n + zcal(n-1);
}
```

Using big-Oh what is the running time of the call `zcal(n)`? Justify your answer.

Using big-Oh what is the *value returned* by the call `zcal(n)` (note: complexity of value returned, not running time: use big-Oh)

Using big-Oh what is the running time of the call `zcal(zcal(n))` based on your answers to the previous two questions. Justify.

Using big-Oh what is the *value returned* by the call `zcal(zcal(n))` (again, based on previous answers, justify).

PROBLEM 2 : (*Listing (20 points)*)

All methods are in a class in which the `Node` class defined on the front-page of this test is accessible.

Part A (4 points)

The method below returns the sum of the lengths of all strings in a linked-list:

```
public int totalLength(Node list) {
    if (list == null) return 0;
    return list.info.length() + totalLength(list.next);
}
```

Write method `listLength` to return the number of nodes in a linked list. Model your code on the method above, your code must be recursive.

```
public int listLength(Node list) {
    if (list == null) return 0;

    return

}
```

Part B (6 points)

The expression below correctly calculates the average length of all the strings in a list:

```
double avg = totalLength(list)*1.0/listLength(list);
```

A classmate argues that the method below calculates the value more efficiently:

```
public double average(Node list){
    int total = 0;
    int count = 0;
    while (list != null){
        total += list.info.length();
        count++;
        list = list.next;
    }
    return total*1.0/count;
}
```

Provide justification that your classmate is correct and justification that your classmate is wrong (two justifications, be sure to indicate which is which.)

Part C (4 points)

Write a method `sorted` that returns true if its linked-list parameter is in sorted/alphabetical order, and returns false otherwise. The call `sorted(list)` returns true for the list on the left and false for the list on the right:

```
{"apple", "banana", "melon"}          {"banana", "lemon", "fig"}
```

Recall that for strings `p` and `q` we have `p.compareTo(q) < 0` whenever `p` comes before `q` in alphabetical/lexicographical order (this is how the `compareTo` method works.) Complete `sorted` below (a list with no nodes is sorted). Assume all elements in the list are unique.

```
public boolean sorted(Node list) {









}
```

Part D (6 points)

Write method `first2last` that moves the first node of a singly-linked list to the end of the list and returns a pointer to the first node of the changed list. Effectively the call `list = first2last(list)` changes the list on the left to the list on the right

```
("ape", "bat", "cat", "dog")          ("bat", "cat", "dog", "ape")
```

```
public Node first2last(Node list) {
    if (list == null || list.next == null) return list;









}
```

PROBLEM 3 : (*Reasoning About Lists (22 points)*)

The method `duplicate` in the code shown below changes parameter `list` so that it's *doubled in place* – for example, the list

```
("ape", "bat", "cat", "dog")
```

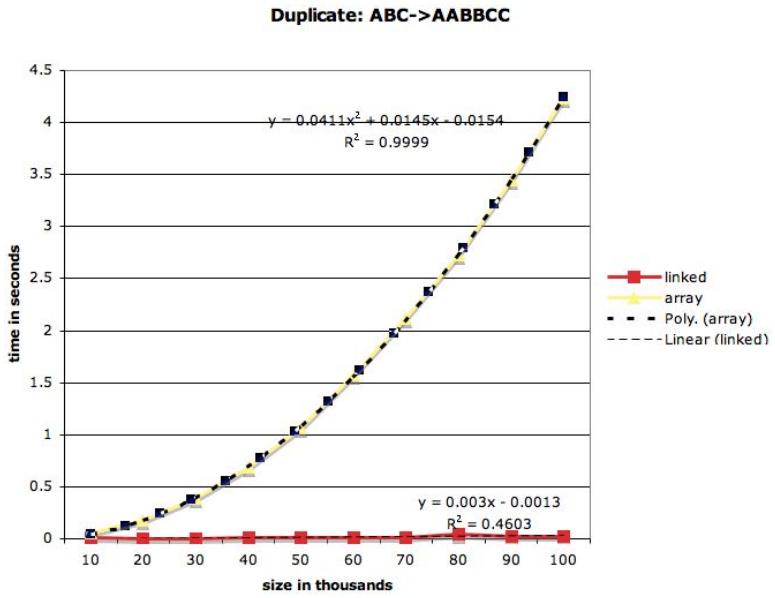
is changed to the list below as a result of the call `duplicate(list)`.

```
("ape", "ape", "bat", "bat", "cat", "cat", "dog", "dog")
```

The method `duplicate` is called with both an `ArrayList` and a `LinkedList` as parameters – the times for duplicating the different lists are shown tabularly and graphically below the code where the size of the list varies from 10,000 to 100,000.

```
public void duplicate(List<String> list){
    ListIterator<String> iter = list.listIterator();
    while (iter.hasNext()){
        String s = iter.next();
        iter.add(s);
    }
}
```

size (10^3)	link	array
10	0.008	0.053
20	0.002	0.176
30	0.005	0.388
40	0.009	0.686
50	0.009	1.069
60	0.014	1.574
70	0.015	2.120
80	0.049	2.729
90	0.019	3.443
100	0.023	4.234



Part A (6 points)

Using the same code on the same computer how much time will it take to duplicate both an `ArrayList` and a `LinkedList` list with 1,000,000 (one million) values. Justify your answers (your answer will be considered approximate, we're looking for close enough.)

(continued)

Part B (4 points)

Using big-Oh, what is the complexity of duplicating both an N-element `ArrayList` and `LinkedList` using the code above. Justify your two answers empirically (based on timings).

Part C (6 points)

Although we haven't discussed `ListIterator` in class, explain the differences in the observed times of duplicating an `ArrayList` and a `LinkedList` based on your understanding of how these classes are implemented. Be specific and account for the observed differences (explain the empirical timings with reasoning).

(continued)

Part E (6 points)

The following suggestion is offered as an alternative to the `duplicate` code above. This code correctly modifies `list` so that it is doubled-in-place.

```
public void duplicate2(List<String> list){
        int originalSize = list.size();
        list.addAll(list);
        for(int k=originalSize-1; k >= 0; k--){
            String current = list.get(k);
            list.set(2*k, current);
            list.set(2*k+1,current);
        }
    }
```

Explain why the `for` loop runs down to zero rather than up from zero (which would not work). Provide a justification for why this code runs very quickly for `ArrayList` parameters and very slowly for `LinkedList` parameters.

PROBLEM 4 : (*Tournament APT (28 points)*)

The code below correctly solves the *TournamentRanker* APT whose description is the last pages of this test. You'll be asked to reason about the code and improvements to it.

```
import java.util.*;
public class TournamentRanker {

    public String[] rankTeams(String[] names, String[] lostTo) {
       final ArrayList<String> nameArray = new ArrayList<String>(Arrays.asList(names));
       final ArrayList<String> lostArray = new ArrayList<String>(Arrays.asList(lostTo));

       Arrays.sort(names, new Comparator<String>(){

            public int compare(String a, String b) {
                int awins = Collections.frequency(lostArray, a);
                int bwins = Collections.frequency(lostArray, b);
                int diff = bwins - awins;
                if (diff != 0) return diff;
                int aindex = nameArray.indexOf(a);
                int bindex = nameArray.indexOf(b);
                return compare(lostArray.get(aindex), lostArray.get(bindex));
            }
       });
       return names;
    }
}
```

Part A (4 points)

The line below calculates the number of wins by the team represented by parameter `a` in the `compare` method. Explain in a sentence or two both why this code is correct and what its big-Oh complexity is when the array parameters to `rankTeams` contain N-elements. Justify your answer.

```
int awins = Collections.frequency(lostArray, a);
```

continue

Part B (4 points)

Using big-Oh, describe the complexity of the method `compare` without the recursive call — so you just account for the six lines before the recursive call to `compare`. Assume the array parameters to `rankTeams` contains N elements and justify your answer. You should account for each of the six lines in justifying your big-Oh answer.

Part C (4 points)

Using big-Oh, what is the maximum number of recursive calls made by method `compare`? Justify your answer, assume the array parameters to `rankTeams` contain N elements (and that N is a power of 2).

Part D (4 points)

Describe how the values in the returned array will change if the line

```
int diff = bwins-awins;
```

is changed to the following. Be brief and precise, not thorough.

```
int diff = awins-bwins;
```

Part E (4 points)

The `compare` method could be made more efficient by replacing calls to `Collections.frequency` and `nameArray.indexOf` with look-ups to appropriate maps as shown below. Describe why this version of `compare` is better than the original in terms of being more efficient.

```
final Map<String,Integer> winMap = new HashMap<String,Integer>();
final Map<String,String> lostMap = new HashMap<String,String>();

// some code skipped that initializes the maps

Arrays.sort(names, new Comparator<String>(){

     public int compare(String a, String b) {
        int awins = winMap.get(a);
        int bwins = winMap.get(b);
        int diff = bwins - awins;
        if (diff != 0) return diff;
        return compare(lostMap.get(a),lostMap.get(b));
       }
  });
```

Part F (4 points)

The code below correctly initialize the maps so that entire APT is solved correctly when the modified, map-using `compare` method is called.

```
for(String s : names){
    winMap.put(s, Collections.frequency(lostArray, s));
    lostMap.put(s,lostArray.get(nameArray.indexOf(s)));
}
```

The complexity of this code is **not** $O(N)$. What is the complexity of this code in terms of N, where the array parameters to `rankTeams` contain N-elements. Justify your answer.

Part G (4 points)

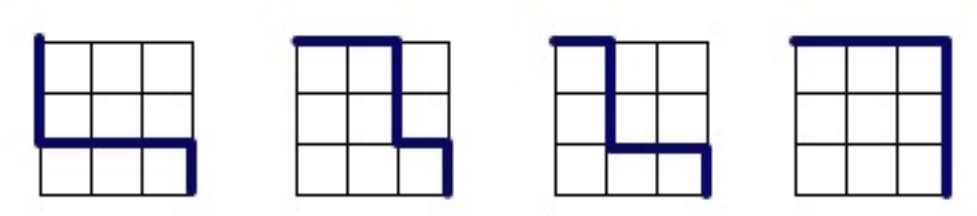
Describe how to initialize the maps in $O(N)$ time when the arrays have N-elements (you don't have to write code, but you can – a description is fine).

Part H (extra credit) (2 points)

Would you like Duke to win either (or both) the NCAA men's and women's basketball tournaments?

PROBLEM 5 : (*Grid Computing (15 points)*)

On a 3×3 grid there are twenty *shortest paths* from the top left to the bottom right where each path consists of six steps. Four of the paths are shown below.



There are 252 shortest paths (of length 10) in a 5×5 grid and in general there are $\frac{(2n)!}{n!^2}$ shortest paths of length $2n$ in an $n \times n$ grid. The code below computes the number of shortest paths by using an $(n+1) \times (n+1)$ grid (because there are four vertices as shown above in each row of of a 3×3 grid). Starting at the top left, or `grid[0][0]`, the code visits every grid point by moving down or right (increasing the row or column), incrementing the number of ways of visiting each grid point. The methods `pathCount` and `visit` together calculate the number of shortest paths so that `pathCount(5)` returns 252 indicating there are 252 shortest paths for a 5×5 grid.

Results of calling `pathCount` for values $3 \leq n \leq 12$ are shown below on the right.

```
public class ShortestPaths {

    private int grid[][];

    public int pathCount(int n){
        grid = new int[n+1][n+1];
        visit(0,0);
        return grid[n][n];
    }

    private void visit(int row, int col){
        if (row < 0 || col < 0) return;
        if (row >= grid.length || col >= grid.length) return;
        grid[row][col]++;
        visit(row+1,col);
        visit(row,col+1);
    }
```

n	pathCount(n)
3	20
4	70
5	252
6	924
7	3,432
8	1,2870
9	48,620
10	184,756
11	705,432
12	2,704,156

Part A (4 points)

To calculate paths that can go up as well as down and right a student suggests adding one line to method `visit` as the last line:

```
    visit(row-1,col);
```

Explain why this results in infinite recursion so that method `visit` doesn't work.

Part B (6 points)

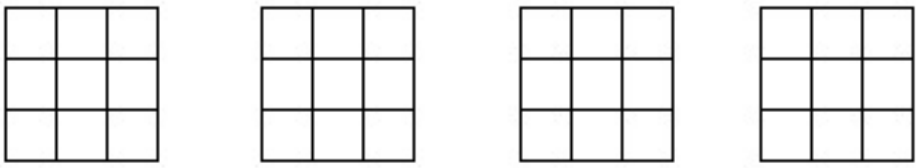
If (row,col) is n-steps away from the bottom right corner when `visit` is called, and T(n) is the time for `visit` to execute when (row,col) is n-steps away from the bottom right corner, then the recurrence relation below correctly describes the complexity of `visit` (note that there are two recursive calls):

```
T(n) = 2T(n-1) + O(1)
```

Justify that this is the correct recurrence relation in a sentence or two. Provide intuition/hand-wavy explantions for why the solution to this recurrence is $O(2^n)$.

Part C (5 points)

Suppose paths in a grid can go up, down, left, or right, but not cross themselves and not touch. The longest such path in the 3×3 grids shown at the beginning of this problem have length 14. Draw *one* such path below.



(nothing on this page)