CS 241, Homework 2

For full credit, you should turn in this homework on paper, with problems written in numerical order, with all pages stapled together.

1. **Big-oh ranking**

   Order the following big-oh complexities in order from slowest-growing to fastest-growing (this is not the same as slowest to fastest running time!) It is possible some of them are actually in the same big-oh category. If that is the case, make it clear which ones have the same complexity.

   n^2, 3^n, $\sqrt{n}$, 1, n*log(n), 2^n, n!, $2^{\log(n)}$, n^3, n, $n^2\log(n)$, log(n), 2^{n+1}

2. **Big-oh complexity**

   Assume each formula below represents the running time T(n) of some algorithm. For each formula, give the lowest big-oh complexity possible (the tightest bound).

   Here, log represents the base-2 logarithm.

   (a) $5 + n^2 + 25n$
   (b) $50n + 10n^{1.5} + 5n*\log(n)$
   (c) $3n + 5n^{1.5} + 2n^{1.75}$
   (d) $n^2\log(n) + n*\log(n) + n*(\log(n))^2$
   (e) $2^n + n^{10}$
   (f) $n*\log(n) + 8n + n*(\log(n))^2$

3. **Big-oh complexity proof**

   Assume we have analyzed an algorithm, and its run time is determined to be

   $T(n) = n^3 + 2n + 3$

   (a) What is the big-oh running time of this algorithm? (This should be easy.) Call this function *f*(n).

   (b) Now, prove your answer.
   In other words, find constants c > 0 and $n_0 > 0$ such that for all numbers $n >= n_0$, $T(n) <= c * f(n)$. You may prove that your c and n_0 work by drawing a graph showing T(n) and $c * f(n)$. Label your axes and where n_0 is.

4. **Big-oh complexity analysis of code**

Determine the big-oh running time for the following algorithms in terms of n. (No justification needed.)

a. Matrix addition:

```
for (int i = 0; i < n; i++)
{
  for (int j = 0; j < n; j++)
  {
    c[i][j] = a[i][j] + b[i][j];
  }
}
```

b. Matrix multiplication:

```
for (int i = 0; i < n; i++)
{
  for (int j = 0; j < n; j++)
  {
    c[i][j] = 0;
    for (int k = 0; k < n; k++)
    {
      c[i][j] += a[i][k] * b[k][j];
    }
  }
}
```

c.

```
counter = 0;
while (n >= 1)
{
  n = n/2;
  counter++;
}
```

d.

```
counter = 0;
x = 1;
for (int i = 0; i < n; i++)
{
  for (int j = 0; j < x; j++)
  {
    counter++;
  }
  x = x * 2;
 }
```

5. Recall that in class we wrote an RArrayList class that works similarly to the built-in Java ArrayList class. Suppose we want to add a method to our RArrayList class called `duplicate()` that will copy all the items in the list a given number of times, and append them to the end of the list. The function will take a parameter, `howmany`, that specifies the number of times the items should be copied and appended.

   Example of use:

```
RArrayList mylist = new RArrayList();
mylist.append(1);
mylist.append(2);
mylist.append(3);
mylist.duplicate(3);
// mylist is now [1 2 3 1 2 3 1 2 3]
```

   In this problem, you will fill in the starter code below. You may assume that there is enough room in the RArrayList to fit "howmany" total copies of the existing data. The code that you add should do the copying and appending part. Do not call any other functions; you should write all of your code inside this function. In other words, don't call append or anything like that. You may assume that howmany is an integer > 1.

```
public void duplicate(int howmany) {
  // Remember, this code lives inside RArrayList, so you have access to
  // the data[] array, data.length, and the size variable.

  // Assume that the data[] array has enough room for the necessary copies.

  // YOUR CODE HERE: Copy the existing data in the array the correct number of
  // times to the end of the data[] array.  Don't forget to update the size
  // variable when you're done.
}
```

   You do not need to recopy the entire function on your answer sheet; just provide the code that would go in the YOUR CODE HERE section.

6. Suppose we have a singly-linked list class with just a head pointer (no tail pointer) like this:

```
public class Node {
  public int data;
  public Node next;
};

public class SList {
  private Node head;
  // Assume there are more methods defined to add items to the list, etc.
}
```

Write a member function for `SList` called `getSize()` that calculates and returns the size (number of elements) in the linked list.

Here is the skeleton method:

```
public int size()
{
  // Remember, this method lives inside SList, so you have access to the
  // head variable defined above.  You should assume if head is null, the
  // list is empty.

  // YOUR CODE HERE: Traverse the linked list and count the number of elements.
  // Return your answer.
}
```

You do not need to recopy the entire function on your answer sheet; just provide the code that would go in the YOUR CODE HERE section.

7. This question uses the same SList singly-linked list class from the previous problem.

Assume we add the following method to the SList class:

```
public void strange() {
  Node curr = head;
  while (curr != null) {
    curr.next = curr.next.next;
    curr = curr.next
  }
```

(a) Suppose we make an `SList mylist` and add the numbers 1, 2, 3, 4, 5, and 6 to `mylist` (so `mylist` consists of those six numbers in that order). What does `mylist` look like after calling `mylist.strange()`? (You can just write down the items in the list from left to right.)

(b) Suppose we make an `SList mylist2` and add the numbers 1, 2, 3, 4, and 5 to `mylist2` (so `mylist2` consists of those five numbers in that order). What happens when calling `mylist2.strange()`?

(c) In general, describe the difference in behavior when running this function on a list with an even number of items versus a list with an odd number of items.

8. Many linked list functions can be written *recursively* as well as iteratively. For instance, here's a recursive function to print out the items in a linked list (using the same SList class from above).

```
public void print(Node curr) {
  if (curr != null) {
    System.out.println(curr.data);      // (1)
    print(curr.next);                   // (2)
  }
}
```

The code above would be called as `print(head)` where head would be a reference to the first node in the linked list.

(a) What would the `print()` function above do if you switched the order of the lines marked (1) and (2) in the code?

(b) Write a recursive function, similar to the one above, that returns the sum of all the numbers in the linked list. Hint: use this skeleton code:

```
public int sum(Node curr) {
  if (curr == null) {
    return ______;
  }
  else {
    // call sum recursively, add something to that value, and return it
  }
}
```

Your code, like `print()`, should not use any loops.

(c) Compute the big-oh time of the `print()` function (or the sum function; their big-oh times are the same). Use the strategy we learned in class to compute big-oh times of recursive functions, and *show all your work*. See the class website for the handout if you forget how to do this. "*n*" here should represent the number of nodes/items in the list.