**1.** In class, we looked at Karatsuba's algorithm for fast multiplication of large integers. Java's *BigInteger* class can multiply very large integers. Does it use the standard "grade-school" algorithm, Karatsuba's algorithm, or maybe something else. I wrote the following short program to test the run time for *BigInteger* multiplication:

```
import java.math.BigInteger;
import java.util.Random;

public class BigTest {
  public static void main(String[] args) {
    Random random = new Random();
    for (int bits = 1000; bits <= 10000000; bits *= 10) {
      BigInteger x = new BigInteger(bits,random);
      BigInteger y = new BigInteger(bits,random);
      long start = System.nanoTime();
      BigInteger z = x.multiply(y);
      long time = System.nanoTime() - start;
      System.out.printf("%,10d bits:  %,d nanoseconds%n", bits, time);
    }
  }
}
```

Here is the output from one run of the program. (Note that the program was run with the command `java -Xint BigTest` to disable the just-in-time compiler.)

```
     1,000 bits: 84,509 nanoseconds
    10,000 bits: 3,661,251 nanoseconds
   100,000 bits: 108,247,437 nanoseconds
 1,000,000 bits: 3,114,291,488 nanoseconds
10,000,000 bits: 96,868,687,259 nanoseconds
```

**Your Assignment:** Analyze this data and say what you can about the algorithm used by multiplication in *BigInteger*. Is it likely to be the grade-school algorithm? Karatsuba's algorithm? Something else? Can you tell? *Explain your reasoning carefully!* If you would like to experiment further, you can find a copy of the program in */classes/cs327*. (Hint: Consider the ratio $T(10 * n)/T(n)$.)

**2.** **a)** Write a recursive Java function, `void max( int[] A, int lo, int hi )`, that finds the maximum value among the array elements `A[lo]`, `A[lo+1]`, ..., `A[hi]`, using the tournament method. (That is, find the maximums in two halves of the array and then compare them.)

**b)** This function can be used to find the max of an $N$-element array by calling `max(A,0,N-1)`. Write a recurrence relation for the run time, $T(N)$, of the function.

**c)** Use the Master Theorem to find $T(N)$. (Explain your reasoning. You shouldn't be surprised by the answer!)

**3.** Suppose that a recursive algorithm divides a problem of size $n$ into 4 problems of size $n/2$. The amount of extra work that is done to split the problem into parts and to combine the results from processing the parts is $\Theta(n^2)$. Write a recurrence relation for the run time of the algorithm

and use the Master Theorem to find the run time. How does the answer change if the extra work has run time $\Theta(n^{3/2})$?

**4.** This is a small programming assignment using Java's *HashSet* data type. You might already have done something similar in CPSC 225. You can work on this assignment with a partner if you want; if you do, be sure to list both names in the file. Your program for this assignment must be named **Books.java**. You can submit your program to any location inside your homework folder in */classes/cs327/homework*. My script will look for files named Books.java in that directory. My version of the program is 42 lines long, without comments. **You do not need to include comments in this program.**

The program will read words from two text files. (You can ask the user for the file names, or you can get the file names from command line arguments, but please do not hard code them into the file!) Read the words from one file, convert them to lower case, and put them into a *HashSet*<*String*>. Read the words from the other file into another *HashSet*<*String*>. A word is defined to be a sequence of ASCII letters, possibly with embedded apostrophes as in the words *o'clock* and *shouldn't've*. You can read words easily using a *Scanner*, provided you change the delimiter that is used by the *Scanner* to separate tokens. Here is a command that you can use to set a *Scanner*, *scanr*, to read words from the file and discard everything else:

```
scanr.useDelimiter("('*[^a-zA-Z']'*|''+|^'|'$)+");
```

(It took me significantly longer to write and debug the regular expression in this command than it did to write the rest of the program. You can copy-and-paste it from the PDF of this assignment on line.)

Now that you have the two hashsets, your goal is to answer the following questions: How many unique words were found in each file? How many words were present in the first file that were not in the second file? How many words were present in the second file that were not in the first file? Putting all the words from the two files together, how many unique words were there in both files combined? (You can answer all of these questions using the *HashSet* API, without writing any loops.)

To make things a little more interesting, you might want to try your program on some of the files in */classes/cs327/books*, which contain the full text of several classic (out-of-copyright) books. For example, here is the output from my program when it compared Jane Austin's *Pride and Prejudice* to Mark Twain's *Huckleberry Finn*:

```
File 'books/austin.txt' contains 6346 unique words.
File 'books/twain.txt' contains 6089 unique words.
'books/austin.txt' contains 4213 words that are not in 'books/twain.txt'.
'books/twain.txt' contains 3956 words that are not in 'books/austin.txt'.
Together, 'books/austin.txt' and 'books/twain.txt'
                    contain a total of 10302 different words.
```

**5. This problem is not required. You can do it for a little extra credit.** The program in the preceding problem could have used a *TreeSet* instead of a hash set, but you would usually do that only if you are interested in processing the contents of the set in alphabetical order, and you would expect the *TreeSet* version of the program to require more time. How much more? Write a program to find out, using the *System.nanoTime* method that was used in problem **1** above. (Your program will have to do enough processing to take a significant amount of time.) Turn in a printout of your program and report on your results.