DUKE UNIVERSITY
Department of Computer Science

Test 1: CompSci 100

Name (print): _____

Community Standard acknowledgment (signature): _____

	value	grade
Problem 1	30 pts.	
Problem 2	16 pts.	
Problem 3	14 pts.	
Problem 4	15 pts.	
TOTAL:	75 pts.	

This test has 10 pages, be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts only 75 minutes and there are 75 points on the exam. That means you should spend no more than *1 minute per point*.

Don't panic. Just read all the questions carefully to begin with, and first try to answer those parts about which you feel most confident. Do not be alarmed if some of the answers are obvious.

If you think there is a syntax error or an ambiguity in a problem's specification, then please ask.

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.

PROBLEM 1 : (*Short Ones (30 points)*)

Briefly answer the following questions.

A. What is the difference between an *abstract class* and an *interface*? If I have implemented generic methods that will be used in subclasses, should I put them in an abstract class or interface?

B. Given the following line of code

```
Object o = s.substring(0);
```

Give **two** (2) examples of errors that could occur. Specify whether the error is a run-time or compile-time error. (Note: there are more than two possibilities. You only need to give two. The errors must occur at this line of code, not inside the substring method.)

C. Translate the following *postfix* expressions into BOTH prefix and infix. Use parentheses where necessary.

I. `a b + c d - / e +`

prefix:

infix:

II. `a b c + d e - * -`

prefix:

infix:

D. What is the output of the following code fragment? What are the contents of the stack at the end?

```
Stack s = new Stack();

s.push("Maryland");
s.push("UNC");
System.out.println(s.pop());
s.push("Duke");
s.push("FSU");
System.out.println(s.pop());
System.out.println(s.pop());
s.push("Clemson");
s.push("Wake");
System.out.println(s.pop());
s.push("UVa");
s.push("GaTech");
System.out.println(s.pop());
System.out.println(s.pop());
s.push("NCSU");
System.out.println(s.pop());
System.out.println(s.pop());
```

E. DNA can be represented as long strings of the nucleotides a, t,c , and g. If we wanted to determine whether DNA strings of length 10^6 were anagrams of each other, which normalizer representation would be more efficient, the fingerprint/histogram or the sort method? *Briefly* justify your answer.

F. State whether the following statement is true or false. If it is true, justify using the definition of big-Oh. If it is false, give a counterexample.

If $f(n) \in O(g(n))$ and $d(n) \in O(h(n))$, then $f(n) + d(n) \in O(g(n) + h(n))$

G. Bill shows you an algorithm to optimally choose classes for all students at Duke. You, being a good Computer Science student, notice that the big-Oh complexity of the algorithm is $O(2^n)$ where n is the number of students. However, Bill demonstrates the program for a sample of 100 students and it returns the schedules almost immediately.

Bill says that his algorithm is good enough for Duke. He mentions something about Moore's Law and states that "Computers are getting faster at an exponential rate. That is, every 18 months, they double in speed. Even if the program is not fast enough now, it will be soon."

Bill is off a little bit on what Moore's law means. However, given that computers continue doubling in speed every 18 months, will the $O(2^n)$ algorithm ever be practical? Explain why or why not?

PROBLEM 2 : (*Complex (16 points)*)

Give big-Oh runtime complexities with *brief* justifications.

A. Give the big-Oh complexity in terms of n, the length array A, of the running time of the following code:

```
/**
 * returns the sum of the odd elements in an Array
 */
int sumOddElems(int A[])
{
    int sum = 0;
    for (int i = 1; i < A.length; i += 2)
        sum += A[i];
    return sum;
}
```

B. Give the big-Oh complexity in terms of n, the length array A, of the running time of the following code:

```
/**
 * Given 2 arrays, A & B  each storing n integers where n >= 1
 * return the number of elements in B equal to the sum of prefix
 * sums in A
 */
int prefixSums(int A[], int B[])
{
    if (A.length != B.length)
        return 0;

    int c = 0;
    for (int i=0; i < A.length; i++)
    {
        int sum = 0;
        for (int j=0; j < A.length; j++)
        {
            sum += A[0];
            for (int k = 1; k <= j; k++)
            {
                sum += A[k];

            }
        }
        if (B[i] == sum)
            c += 1;
    }
    return c;
}
```

C. Give the big-Oh complexity in terms of n of the running time of the following code:

```
/**
 * returns the integer binary logarithm of n
 */
int lg(int n)
{
    if (n==1) return 0;

    return 1 + lg(n/2);

}
```

D. Below are two methods, *sumPoly1* and *sumPoly2* that both correctly calculate the sum of a polynomial with coefficients $a0, \ldots, a_{n-1}$, that is:

$$\text{sumPoly}(x) = \sum_{i=0}^{n-1} a_i x^i = a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$$

```
double sumPoly1(double a[], double x, int index)
{
    if (index >= a.length) return 0;
    double prod = 1.0;
    for (int i=0; i < index; i++)
    {
        prod *= x;
    }
    return a[index] * prod + sumPoly1(a, x, index+1);
}

double sumPoly2(double a[], double x, int index)
{
    if (index >= a.length) return 0;
    return a[index] + x*sumPoly2(a, x, index+1);
}
```

I. Give the call to *sumPoly1* that returns $5x^4 + 3x^2 - 17x + 1$

II. What is the runtime complexity of *sumPoly1* in terms of the length of `a`, the coefficients in the polynomial? Use big-Oh and justify your answer briefly, using recurrences where possible.

III. What is the runtime complexity of *sumPoly2* in terms of the length of `a`, the coefficients in the polynomial? Use big-Oh and justify your answer briefly, using recurrences where possible.

PROBLEM 3 : (*Find your perfect match (14 points)*)

The *match* method is given an array of prefixes and suffixes and attempts to match each prefix to form a valid word. Each prefix and each suffix is used exactly once in the match. You can determine what strings are words by searching in the `TreeSet dict`. If a valid match is found after a given `index`, *match* returns true and the entries in the suffix array have been rearranged such that that the suffix matched to `prefix[0]` is at index 0, `prefix[1]`'s match is at index 1, and so on. If no match is found, the function returns false and the entries in the suffix array are unchanged.

If attempting to match these two arrays:

`prefixes`	show	great	cook	read	top

`suffixes`	est	s	ier	ing	ed

`match(prefixes, suffixes, dict, 0)` would return `true` and the suffixes array would be rearranged as shown below.

`suffixes`	ier	est	ed	ing	s

Hints:

- Use the `contains` method in the `TreeSet` class to determine if a prefix + suffix combination is in `dict`
- Recursion works well here. The base case is where all prefixes and suffixes have matched, that is when where there are no prefixes past `index` to consider. The recursive case should consider all unused suffixes.
- You may use the helper `swap` function below that exchanges strings in an array.

```
void swap(String[] a, int i, int j)
{
  String temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}
```

Complete *match* below.

```
/**
 * @return true if entries in pref and suff can be combined
 * to create words that match strings in dict
 * Assume pref and suff are of the same length
 */
boolean match(String[] pref, String[] suff, TreeSet dict, int index)
{
```

PROBLEM 4 : (*Love is not always rational (15 points)*)

The class `RationalNumber.java` is appended to the back of this exam. The class represents *rational numbers*. All rational numbers can be represented as n/d where n and d are integers.

A. Fill in the `compareTo` method for *RationalNumber*.

```
/**Compares this object with the specified object for order.  Returns a
  * negative integer, zero, or a positive integer as this object is less
  * than, equal to, or greater than the specified object.
  */
public int compareTo(Object o)
{
```

B. Complete method the method *add* that returns the result of adding this rational number to another passed as an argument. Note that

$$\frac{i}{j} + \frac{k}{l} = \frac{il + kj}{jl}.$$

]

```
/**
 *  returns the result of adding this rational number to one
 *  passed as a parameter. A
 */
public RationalNumber add(RationalNumber op2) {
```

C. We would like to write an algorithm to find two rational numbers in A that sum to x. The method below, *findPair*, takes a *sorted* ArrayList of RationalNumbers, index bounds i and j, and a RationalNumber x. *findPair* returns true if there exists a pair of RationalNumbers in the ArrayList between i and j (inclusive) that sum to x.

The code for *findPair* is started below. Fill in the recursive calls below.

```
/**
 * Given sorted sublist A[i..j] , determines whether there is any
 * pair of elements that sums to x
 */
public boolean findPair(ArrayList A, int i, int j, RationalNumber x)
{
    if (i == j) return false;
    RationalNumber Ai = (RationalNumber)A.get(i);
    RationalNumber Aj = (RationalNumber)A.get(j);

    RationalNumber pairSum = Ai.add(Aj);
    int comp = pairSum.compareTo(x);
    if (comp == 0)
        return true;
    else if (comp < 0)

        return ____________________________________________________;
    else if (comp > 0)
    {

        return ____________________________________________________;
    }
}
```

D. Given that `Collections.sort` runs in $O(n \log n)$ time, what is the worst-case big-Oh time complexity of *addsToX* in terms of n, the size of the ArrayList. Briefly justify your answer.

```
public boolean addsToX(ArrayList A, RationalNumber x)
{
    Collections.sort(A);     // nlogn

    return findPair(A, 0, A,size()-1, x);
}
```