Assignment 4: Recursion

October 7, 1999

Due date: October 21, 1999

This assignment asks you to write a number of recursive Java programs. You must write, debug, and run these programs on the computer. Scribbles on paper will not be accepted.

Problem 1: Perfect Packing

As the logistics assistant to the chief bogus executive of NASA's Mars Colonization Program you find yourself in charge of shipping the disassembled parts of the new Mars station from Earth to the red planet. At your disposal is a limited number of brand new class-7 space rockets that can hurl a mind-boggling amount of cargo into space. Of course, class-7 space rockets are very, very expensive, so you really want to make sure that no rocket that goes up to Mars is only partially filled.

Each class-7 rocket can carry a load of T tons, and the space station is disassembled into n parts weighing $v_0, v_1, \ldots, v_{n-1}$ tons, respectively. To aid your planning, write a Java program that will determine whether a given rocket can be packed without waste, i.e. given an integer T and an integer array $v[\,]$, check if there exists a subset of $\{v_0, \ldots, v_{n-1}\}$ that adds up exactly to T. Note that the program need not output the actual subset; a correct YES/NO answer is sufficient.

```
public static boolean perfectPack (int T, int[] v) {
   ...
}
```

Submit your implementation of the `perfectPack` method; neither program output nor sample runs are required. To help us evaluate your program, give an explanation in English of how your algorithm works. What is the base case? What are the subproblems generated recursively by your program? Note that a recursive solution need not be longer than a few lines.

Problem 2: Three-way Merge Sort

The Merge Sort routine described in class cut an array into two arrays of roughly equal size, sorted the two arrays recursively and then merged them to produce the result.

Write a Java routine that sorts an integer array by dividing it into *three* arrays of roughly equal size, sorting the three arrays recursively and then merging them. Give careful thought to the merging routine.

Determine the asymptotic complexity of your algorithm and compare it to the complexity of the standard two-way merge sort.

Problem 3: Recursive Descent Parsing

A *boolean expression* (BE) is defined recursively as follows:

- The constants `T` and `F` are BE's.
- If E is a BE's, then `!`E is a BE.
- If E_1 and E_2 are BE's, then `(`E_1`&`E_2`)` is a BE.
- If E_1 and E_2 are BE's, then `(`E_1`|`E_2`)` is a BE.
- If E_1 and E_2 are BE's, then `(`E_1`>`E_2`)` is a BE.
- If E_1 and E_2 are BE's, then `(`E_1`=`E_2`)` is a BE.

For example, `(!F & (T > !!T))` is a BE, but `T & F` is not.

Write a recursive Java program that reads a BE from a file and returns the result of evaluating it according to the following rules. If the expression is not a valid BE, your program should say so.

		not	and	or	implies	equiv
p	q	! p	p & q	p | q	p > q	p = q
F	F	T	F	F	T	T
F	T	T	F	T	T	F
T	F	F	F	T	F	F
T	T	F	T	T	T	T

Download the `Parser.java` program skeleton and the `CS211In` class file provided on the course home page. Submit your completed parser and show us the result produced by your program for each of the following inputs:

1. `!(T = F)`
2. `(!F)`
3. `((F > !F) > F)`
4. `(T > F & T)`
5. `(F | !(T & F))`
6. `(T = F) > (F = T)`

Beware of one potential pitfall in your implementation! The Boolean operators `&&` and `||` are *lazy*, i.e. in a statement like `boolean a = true || boolFunc();` the function `boolFunc()` will *not* be called, since the value of `a` does not depend on its result. The logical operators `&` and `|` are not lazy.

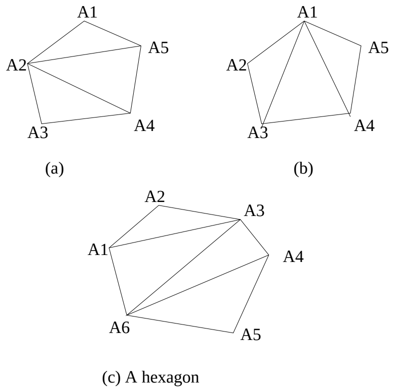

Figure 1: Triangulations

Problem 4: Triangulation

A key step in many geometric algorithms is the *triangulation* of a polygon. Think of a polygon as a cake. To triangulate it, we chop it up into triangular pieces by making cuts which run from one corner of the polygon to another. The cuts are not allowed to intersect each other.

Assume that the corners of the polygon are labeled A_1, A_2 etc. A cut is specified by a pair of vertices (so (A_1, A_5) is a cut between corners A_1 and A_5), and a triangulation is specified by its set of cuts. Two triangulations of a polygon are considered to be the same iff both triangulations use the same set of cuts.

Figure 1(a),(b) show two ways to triangulate a polygon with 5 nodes. Triangulation (a) on the left can be specified by the set of cuts $\{(A_2, A_5), (A_2, A_4)\}$. Figure 1(c) shows a triangulation of a hexagon.

Find a recursive formula for the total number of distinct triangulations of a polygon with n corners (you do not have to find a closed expression). This problem does not involve any coding—the goal is to get you to think recursively. Submit your formula and the derivation.

Hint: Note that making a single cut chops the polygon into two polygons with fewer corners.

Problem 5: $O(\cdot)$-Notation

Rank the functions

$$(\log 2n)^2 \qquad n \log\log n \qquad \log n^2$$

by order of growth, i.e. find an arrangement f_1, f_2, f_3 such that $f_1 = O(f_2)$ and $f_2 = O(f_3)$. Give proofs for the correctness of your ranking.

Problem 6: Computational Complexity

Determine the asymptotic time complexity of the following algorithms. Think about how you can describe the size of a matrix or a vector, and express the running time in these terms. Justify your answers.

(a) Matrix-Matrix Multiplication

```
static int[][] matrixMatrixMult(int[][] A, int[][] B) {
    int[][] r = new int[A.length][B[0].length];
    for (int i=0; i<A.length; i++)
        for (int j=0; j<B[0].length; j++) {
            r[i][j] = 0;
            for (int k=0; k<B.length; k++)
                r[i][j] += A[i][k] * B[k][j];
        }
    return r;
}
```

(b) Matrix-Vector Multiplication

```
static int[] matrixVectorMult(int[][] A, int[] b) {
    int[] r = new int[A.length];
    for (int i=0; i<A.length; i++) {
        r[i] = 0;
        for (int k=0; k<b.length; k++)
                r[i] += A[i][k] * b[k];
        }
    return r;
}
```

(c) Dot-Product

```
static int dotProduct(int[] a, int[] b) {
    int r = 0;
    for (int i=0; i<a.length; i++)
        r += a[i] * b[i];
    return r;
}
```

Problem 7: Running Time

The goal of this exercise is to get a feel for the real running time behavior of programs in different complexity classes. Download the `RunTime` and `RandomInts` classes from the course home page. The `RunTime` class defines five methods with different running time behavior:

Algorithm	Name	Complexity
Selection Sort	`sel`	$O(n^2)$
Merge Sort	`merge`	$O(n \log n)$
Linear Search	`lin`	$O(n)$
Binary Search	`bin`	$O(\log n)$
Fibonacci Numbers	`fib`	$O(2^n)$

First, complete the `RandomInts` class to take an integer n and generate a file of n positive random integers between 1 and n with repetitions allowed. Run the Linear Search algorithm on your file, e.g.

```
java RunTime lin randints.txt
```

if the file you generated is called `randints.txt`. The program will search your file for the integer -1 and output 'false', since -1 is not contained in your file by assumption. Record the running time required for the search as reported by the program.

Next, run the Selection Sort and the Merge Sort algorithms on your file. The programs will generate the file `sorted`, which will contain your numbers in increasing order. Again, record the running time reported by the program (this excludes the time for input and output operations, since we are not interested in those).

Now run the Binary Search algorithm on the file `sorted` (remember that binary search requires a sorted array to work) and record the running time.

Rinse and repeat with at least five different values for n such that your largest value is one or two orders of magnitude larger than your smallest one. Execution time depends on your hardware and the compiler you are using, so you will have to experiment a little to find an interesting interval.

Finally, run the Fibonacci Numbers with values 25, 30, ..., 50, e.g.

```
java RunTime fib 25
```

Plot one graph that shows the running time of the different algorithms over the size n of the input. Useful programs for this task include Matlab, gnuplot, and Excel. You might want to use a different x-axis for the Fibonacci data; if your plot program does not support this, draw it as a separate graph. Hand in your code and a printout of your graph(s), preferably including the numerical values of your data points. Do not submit a transcript of the program runs or the files generated.