187 Programming With Data Structures

Assignment 03 – 65 Points

Overview

This assignment focuses on the topics covered in Chapters 4 in *Object-Oriented Data Structures Using Java 3rd Edition*. The exercises include both written and programming problems. You must provide a solution for each exercise completely to receive credit. The written exercises must be submitted as a PDF document and the programming tasks must be submitted as an exported Eclipse project[1]. You must submit both before the specified due date to receive full credit.

Submission Instructions

You are to submit your solutions to Blackboard before the specified due date. It is your responsibility to ensure that you have submitted your assignment properly. To do this you should upload your solutions to Blackboard and check to make sure that your submission has been received. In addition, if you are not confident that you uploaded your solutions properly it is helpful to download your submission to make sure that what you submitted is what you think you submitted. We will not allow extensions to any assignment, including assignments that were not submitted properly, so it is important that you verify that Blackboard has received your submission correctly before the assigned due date. It is extremely unlikely that Blackboard will be in error so make sure your submission is received.

Written Exercises

Question 1 (5 Points): Use the following three mathematical functions (assume *N >= 0*):

- Sum(N) = 1 + 2 + 3 + … + N
- BiPower(N) = 2^N
- TimesFive(N) = 5N

Write Java-like pseudo code methods that use recursion to implements the above three functions. Describe any input constraints in comments (using '//' as your line comment prefix) and use the 3-Question approach to verify these programs. Describe each step of the 3-Question approach as part of your answer.

Question 2 (5 Points): Describe the difference between direct and indirect recursion.

Question 3 (5 Points): Consider the following code:

```
int puzzle(int base, int limit) {
  if (base > limit) {
    return -1;
  }
  else if (base == limit) {
     return 1;
  }
  else {
    return base * puzzle(base + 1, limit);
```

[1] Please see the video on the course website describing how to export Eclipse projects for submission.

```
  }
}
```

Identify the following:

a. The base case(s) of the puzzle method.
b. The general case(s) of the puzzle method.
c. The constraints on the arguments passed to the puzzle method.

Show what would be written by the following calls to the recursive method puzzle:

```
d. System.out.println(puzzle(14, 10));
e. System.out.println(puzzle(4, 7));
f. System.out.println(puzzle(0, 0));
```

Question 4 (5 Points): Explain what is meant by the following terms:

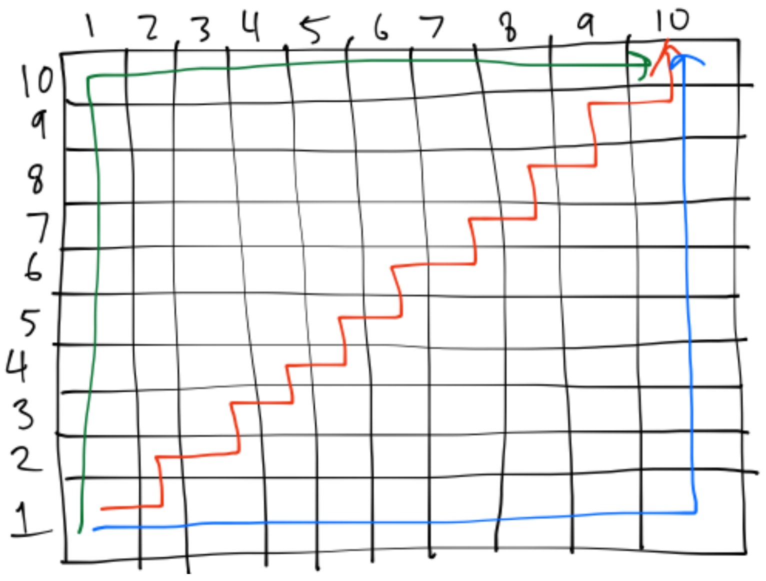
a. Run-time stack
b. Static storage allocation
c. Dynamic storage allocation
d. Activation record
e. Tail recursion

Question 5 (5 Points): Explain the relationship between dynamic storage allocation and recursion.

Programming Exercise

Part 1 (20 Points)

We want to count the number of possible paths to move from row 1, column 1 to row N, column N in a two-dimensional grid. Steps are restricted to going up or to the right, but not diagonally. The illustration shows three of many paths, if N = 10:



a. The following method, `numPaths`, is supposed to count the number of paths, but it has some problems. Create an Java Eclipse project called *asn3* and create a single class called Grid with the following method:

```
int numPaths(int row, int col, int n) {
  if (row == n)
    return 1;
  else if (col == n)
    return (numPaths + 1);
  else
    return (numPaths(row + 1, col) * numPaths(row, col + 1));
}
```

Run the algorithm on the following input values:

numPaths(0, 0, 10);
numPaths(0, 0, 30);
numPaths(0, 0, 50);
numPaths(0, 0, 100);
numPaths(0, 0, 1000);

Report the results generated by your `numPaths` method.

b. After you have corrected the method, trace the execution of `numPaths` with n = 4 by hand. Why is this algorithm inefficient?

c. The efficiency of this operation can be improved by keeping intermediate values of `numPaths` in a two-dimensional array of integer values (the results of computing a recursive call to `numPaths` from a particular point). This approach keeps the method from having to recalculate values that it has already done. Design and code a version of `numPaths` called `numPathsMemoize` that uses this approach.

d. Run the algorithm in part c with the same input values as part a.

e. How do the two versions of `numPaths` compare in terms of time efficiency? Space efficiency?

Part 2 (20 Points)
The Fibonacci sequence is the series of integers

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

See the pattern? Each element in the series is the sum of the preceding two elements. Here is a recursive formula for calculating the *nth* number of the sequence:

$$\mathrm{Fib}(N) = \begin{cases} N, & \text{if } N = 0 \text{ or } 1 \\ \mathrm{Fib}(N-2) + \mathrm{Fib}(N-1), & \text{if } N > 1 \end{cases}$$

a. Write a recursive method `fibonacci` that returns the *nth* Fibonacci number when passed the argument *n*. To do this, create a new Java file in your *asn3* project called `Fib.java` containing the class Fib and includes this static recursive method.

   Run your method on the following inputs:

   fibonacci(10);
   fibonacci(30);
   fibonacci(50);
   fibonacci(100);
   fibonacci(1000);

   Report the results generated by your `fibonacci` method.

b. Write a non-recursive method `fibonacci-loop` that returns the *nth* Fibonacci number using loops. Run your new method on the same inputs from part a and report the results generated.

c. Write a third recursive method `fibonacciMemoize` using the same technique we used in part 1.c above for the Grid problem to keep a two-dimensional array of integer values that hold the results of computing Fibonacci so we do not need to recalculate values that have already been done. The two indexes correspond to the N-2 and N-1 in the recursive call in `fibonacci` and the value is the result from that recursive call. Run your new method on the same inputs from part a and report the results generated.

d. Compare the efficiency of each of these implementations of Fibonacci. (Use words, not Big-O notation)