CSC172 LAB 21

DIVIDE AND CONQUER

1 Introduction

The labs in CSC172 will follow a pair programming paradigm. Every student is encouraged (but not strictly required) to have a lab partner. Labs will typically have an even number of components. The two partners in a pair programming environment take turns at the keyboard. This paradigm facilitates code improvement through collaborative efforts, and exercises the programmers cognitive ability to understand and discuss concepts fundamental to computer programming. The use of pair programming is optional in CSC172. It is not a requirement. You can learn more about the pair programming paradigm, its history, methods, practical benefits, philosophical underpinnings, and scientific validation at http://en.wikipedia.org/wiki/Pair_programming.

Every student must hand in their own work, but every student must list the name of their lab partner if any on all labs.

This lab has six parts. You and your partner(s) should switch off typing each part, as explained by your lab TA. As one person types the lab, the other should be watching over the code and offering suggestions. Each part should be in addition to the previous parts, so do not erase any previous work when you switch.

The textbook should present examples of the code necessary to complete this lab. However, collaboration is allowed. You and your lab partner may discuss the lab with other pairs in the lab. It is acceptable to write code on the white board for the benefit of other lab pairs, but you are not allowed to electronically copy and/or transfer files between groups.

2 Maximum Subarray Problem

This problem comes from the area of computer science known as pattern-matching and was originally faced in its two dimensional form by Ulf Grenander at in 1977. While the one dimensional case has a "fast" solution, a "fast" solution to the two dimensional analog still remains an open problem. For the purposes of comparison of algorithms, we will look at the one dimensional case. An excellent discussion of this problem can be found in article number eight of *Programming Pearls 2nd ed*, by Jon Bently [2000].

The problem is stated as follows. The input is a vector x of n floating-point numbers. The output is the maximum sum found in any contiguous sub-vector of the input. For instance, if the input vector contains the following ten elements [31,-41,59,26,-53,58,97,-93,-23,84] then the program returns the sum of x[2..6], or 187. The problem is easy when all the numbers are positive; the maximum sub-

vector is the entire input vector. It's also easy when all the inputs are negative; the maximum sub-vector is the empty vector, which has a sum of zero. The difficulty comes when some of the numbers are positive and some are negative: should we include a negative number in hopes that the positive numbers on either side will compensate for it?

1. Let us begin this lab by defining a simple program that will generate a double array filled with random values from -100 to +100. Your program should take in a single command line argument for the length of this array. Use the main method skeleton below in your class to display usage information if the program is not run correctly, then implement a method to generate the random array. Print the array to the console after it has been generated.

```
public static void main(String[] args) {
    if (args.length != 1) {
        System.out.println("Usage: java MSP <arraySize>");
        System.exit(1);
    }
    // Your testing code here
}
```

2. In this lab we will compare the run times of two alternative algorithms. The first will be the simplistic $O(n^3)$ approach. The second will be O(n log n). So, lets start by quickly coding the obvious algorithm with three for loops that returns the sum of the maximum subarray. The outside loop selects a starting point in the array. The second loop selects an ending point and initializes a variable for the sum. The third loop sums up the values in the array from the start location to the end location. In order to keep time we we can use the system clock. This methodology can be inaccurate because of other processes loading the system, but it provides reasonable and easy to program approximation.

```
long startTime = System.currentTimeMillis();
double maxSubValue = // call your method here
long endTime = System.currentTimeMillis();
long elapsedTime = endTime – startTime;
```

3. Run your code for some different sizes of the array until you can see a pattern in the growth of the function. Write one or two sentences about the comparison between what you measure and what you would theoretically predict from a Big-Oh analysis.
4. Let's now look at the divide and conquer approach. Implement a recursive method according to the pseudocode description below

```
func divconq(arr, lower, upper)
    // Zero elements to consider
    return 0 if lower > upper
```

```
// One element to consider
return max(arr[lower], 0) if lower == upper

median = (lower + upper) / 2

// find the max crossing to the left
max_lower = 0
sum = 0

for i = median, i >= lower, i--
    sum += arr[i]
    max_lower = max(max_lower, sum)

// find the max crossing to the right
max_upper = 0
sum = 0

for i = median + 1, i <= upper, i++
    sum += arr[i]
    max_upper = max(max_upper, sum)



cur_max = max_lower + max_upper

// evaluate each half recursively
rec_max_lower = divconq(arr, lower, median)
rec_max_upper = divconq(arr, median + 1, upper)

// return the maximum subarray sum found
return max(cur_max, rec_max_lower, rec_max_upper)
```

5. Verify that you two solutions give the same results. This is easy since neither changes the array. Use the timing code in your main method to time both solutions for various length arrays until you can find a pattern. Give a brief explanation of the Big-Oh run time of the divide and conquer approach. Write two sentences about the comparison between what you measure and what you would theoretically predict from a Big-Oh analysis for the divide and conquer approach. Write one sentence about the comparison of the two solutions.

3 Hand In

Hand in the source code from this lab at the appropriate location on the BlackBoard system at my.rochester.edu. You should hand in a single zip file (compressed archive) containing your source code, README, and OUTPUT files, as described below.

1. A plain text file named README that includes your contact information, your partner's name, a brief explanation of the lab (A one paragraph synopsis. Include information identifying what class and lab number your files represent.), and one sentence explaining the contents of all the other files you hand in.
2. Several Java source code files representing the work accomplished for this lab. All source code files should contain author and partner identification in the comments at the top of the file.
3. A plain text file named OUTPUT that includes author information at the beginning and shows the compile and run steps of your code. The best way to generate this file is to cut and paste from the command line.

4 Grading

Each section (1-5) accounts for 18% of the lab grade (total 90%)

README file counts for the remaining 10%