CS2223, Project 4
Divide and Conquer[*]

Due Monday, 3 Dec 2012

There are two separate problem areas to work on. The first comes from Kleinberg-Tardös, *Algorithm Design*, pp. 242–4. The second comes from *CLRS*, Section 4.1, pp. 68–74. Read this section in *CLRS* carefully. You may look at Kleinberg-Tardös if you want, you won't need to.

Turn in a lua file with working code, and also a file called README.txt (in plain text format, not Word or rtf, etc.) with the answers to the questions in the text below. Use Turn-in at `https://turnin.cs.wpi.edu/` to turn in a zip file of the lua code and README.txt.

Be sure to include in your files your commands to do testing and measurement, and include in comments the output that you observed.

I. The Peak of a Unimodal Array. An array a is *unimodal* if there is a single index p which has the maximum value, and a is strictly increasing up to index p, and then strictly decreasing after the index p:

$$\begin{aligned} 1 \le i < j \le p &\quad \text{implies} \quad a[i] < a[j] \\ p \le i < j \le \#a &\quad \text{implies} \quad a[i] > a[j] \end{aligned}$$

So the array has a *peak* value at index p, and values rise steadily to $a[p]$ from the left. They descend steadily from $a[p]$ to the right. Naturally, $p = 1$ or $p = \#a$ are both permitted; in this case p is at one end of the array and the whole rest of the array slopes in one direction.

The file `proj4.lua` contains functions to generate unimodal arrays, and to check whether a given array a is unimodal.

[*]Joshua Guttman, FL 137, `mailto:guttman@wpi.edu`. Include [cs2223] in the Subject: header of email messages. Due midnight at the end of 3 Dec.

Your job is to use the divide-and-conquer idea to find the peak of a unimodal array. The idea is simple: If you look at the midpoint of a and the next entry, and the next entry is larger, then the peak must be in the second half of a. If the next entry is smaller, then the peak most have been in the first half. Namely, let $m = floor(\#a/2)$; if

$$a[m] < a[m+1]$$

then we can ignore the half of the array from indices 1 to m and look only at the part from $m+1$ to $\#a$. Otherwise, we only need to look at the part from 1 to m.

So, the logic here is a little like mergesort, except that there's never any need to come back and merge. When we're done with the recursive calls, and we've found the peak, we just need to return the answer.

1. Implement this idea in the function `uni_max_rec` in proj4.lua. Test your code sufficiently to be sure that it always gives the right answer: even if the peak is at one end, even if the array length is odd, etc. (30 points.)

2. Use the comparison-counting code to determine how many comparisons your code needs to make to find the peak in arrays of lengths up to 100,000 at least. This should be fast. What is the rate of growth in the number of comparisons, using Θ notation? (10 points.)

3. A *local maximum* is an index m such that $a[m] > a[m+1]$ and $a[m] > a[m-1]$, or when m is at one end of the array or the other, then $a[m]$ is greater than its one neighbor.

   The peak of a unimodal array is certainly a local maximum.

   Can you sometimes also use the same code to find a local maximum even if the array is not unimodal?

   Define which arrays the procedure will work correctly for. Test it on several arrays of that kind. Also test it on at least one array that it will give an incorrect answer for. Include small examples of each kind in README.txt. (10 points.)

Extra credit. For 10 points extra credit, write a function that will find some local maximum in *every* array that has a local maximum, and will report the absence of a local maximum when an array does not have one.

For full points, your new procedure should have the same asymptotic efficiency if the array it is given is actually unimodal.

II. Maximum Subarray. Read *CLRS*, Section 4.1 to learn about the maximum subarray problem.[1] In this project, you will first program the simple, inefficient algorithm described at the top of p. 70. Then you will program the divide-and-conquer version described on pp. 70–73. You will use the first, brute-force version to test your divide-and-conquer algorithm. You will also compare their runtimes.

1. To give a brute-force solution to the maximum subarray problem, you will compute the sums of all of the $\Theta(n^2)$ subarrays of an array of length n. Write a function to do this, testing it on hand-crafted examples of small lengths. (10 points.)

   Of course, your test cases will need to include negative values, since if all the entries in an array are non-negative, the full array will always have the maximum sum.

2. Now use the divide-and-conquer idea suggested by *CLRS*. You will compute a maximal subarray entirely in the left half of an array; a maximal subarry entirely in the right half; and a maximal subarray that crosses the midpoint of the array. Whichever has the largest sum will be returned as the result.

   In fact, you will return three values, namely the lower endpoint, the upper endpoint, and the sum. Lua conveniently allows you to return multiple values from a function. The starter file illustrates the syntax to use.

   The procedure for computing a maximal crossing subarray is worth 15 points, and it must be $\Theta(n)$ in complexity. The recursive, divide-and-conquer procedure is also worth 15 points.

   Test your efficient procedures; use your brute force method to determine the expected answers.

3. For 10 points, measure the complexity of the divide-and-conquer version. In README.txt, report its measured runtime as a function of array size. Comment whether your measurements are compatible with the analysis in *CLRS*.

Extra credit. For 10 points extra credit, add a modified version of your divide-and-conquer maximal subarray finder to instead find a subarray with

[1]Like *CLRS*, we use only non-empty subarrays. If an array has only negative entries, then we're looking for a one-element subarray containing a largest (least-negative) entry.

the *minimal* sum. In README.txt, identify the (small number of) points in the code that you had to change in order to get the minimal subarray.

You should also explain whether you can use a similar idea to compute a subarray with maximal *product*, rather than sum.

For 3 separate points of extra credit, create a version that allows empty subarrays. Add a modified version of your divide-and-conquer maximal subarray finder that will return an empty subarray when all the subarrays have negative sums. You can represent an empty subarray by the endpoints s and $s - 1$, meaning that it ends one slot before it begins. Its sum is 0.