Homework 3

Due: Feb 16, 2013 2:00 PM (early)
Feb 20, 2013 11:59 PM (on time)

The written portion of the homework must be handed in to the CS157 hand-in bin located on the CIT 2nd floor between the Fishbowl and the locker. The programming portion of homeworks (when applicable) must be handed in electronically via the hand-in script.

Each problem must be handed in separately with your name on the top and the time of hand-in. You may turn in different problems for different deadlines. For more information, please review the assignment and due date policy in the course missive on the course webpage.

Please ensure that your solutions are complete and communicated clearly: use full sentences and plan your presentation before you write. Except in the rare cases where it is indicated otherwise, consider every problem as asking you to prove your result.

You are allowed, and often encouraged, to use Wikipedia to help with your homework. See the collaboration policy on the course webpage for more details.

Important: Please pay attention to the deadlines of this homework, indicated above. Since there is no late deadline, all homework must be submitted prior to the on-time deadline. Late hand-ins will *not* be accepted.

Discussion with peers is strongly encouraged; this homework will be more challenging than the previous ones. However, you are expected to independently write up your own answer. Refer to the collaboration policy for more information.

Problem 1

You are given an array of n distinct numbers with an unusual property: the numbers are strictly increasing from the first element to the k-th element, for some unknown integer k, and the numbers are strictly decreasing from the k-th element to the last element.

Devise a $O(\log n)$ algorithm that receives such array as an input and finds the maximum element in the array. Explain the algorithm in high level as well as provide the pseudocode, then prove its correctness and run time.

For example, if the input array is:

1	4	7	8	6	3	0

then the output should be 8.

Note: Remember, the key to communicating your solution effectively and painlessly is to emphasize the structure of the argument, to "signpost" the main points. If you tell us all the ingredients and how they relate to each other, then we will often be able to trust your solution even if you omit some details.

Problem 2

Given a tree with n vertices that has weighted edges, and a positive integer k, we would like to find a path (that does *not* use any vertex more than once) of length k in the tree, whose sum of edge weights is maximal. Recall that the length of a path is the number of edges in that path.

Over the course of the following steps, we will devise an $O(n \log n)$ algorithm for this problem, that uses a mix of divide-and-conquer and dynamic programming. (There is a simpler algorithm that takes time $O(nk)$, but that is not what we are aiming for.) For each of the following parts that asks for an algorithm, a proof of correctness and running time complexity is also needed.

1. Suppose you have a rooted tree with n vertices. Design an $O(n)$ algorithm that finds the best (that is, the maximal weight) path of length k that *starts at the root*, and goes down from there. (Hint: dynamic programming.)

2. Suppose **for this part only** that the root of the tree has *exactly* two children. Modify your algorithm above so that it now finds the best path of length k that passes *through* the root. (It can start or end at the root, or pass through it as an intermediate node.) This algorithm should also take $O(n)$ time.

3. Now solve the previous part *without* restricting the number of children of the root. The algorithm should still take $O(n)$ running time.

   Warning or hint: Make sure your path does *not* use any vertices more than once; check that the starting vertex and the ending vertex of the path do not belong to the same subtree of the root.

4. Prove that for any given tree with n vertices, there is a vertex that, if we remove it, splits the tree into several smaller trees where each of these remaining trees has at most $\frac{n}{2}$ vertices.

5. Find a $O(n)$ algorithm that finds such a vertex. (Hint: dynamic programming. Note that one way to solve the previous part is to come up with an algorithm for this part and then prove that the output satisfies the conditions of the previous part.)

6. Use the previous two algorithms you have constructed to design an $O(n \log n)$ *divide-and-conquer* algorithm that solves our problem, finding the maximal-weight path of length k in the tree.

   Note: You do not have to solve parts 3 and 5 to get credit for part 6: in part 6 you may assume that algorithms for parts 3 and 5 exist and they work correctly as indicated.

Problem 3

Parts of this problem involve writing Matlab functions, which must be handed in electronically using the hand-in script `cs157_handin hw3` from the directory containing your Matlab source files; other parts involve writing explanations in complete sentences that must be handed in separately on paper to the hand-in bin as usual. Stencils for each of the functions are in `/course/cs157/pub/stencils/`. The four functions that you will write for this problem are `myconv.m` (for part 1), `roundup235.m` (for part 3), `myconv235.m` (for part 4), and `bigmult.m` (for part 6). Please **do not** rename the files when you hand them in, or else your TA's will be sad.

Matlab has a function `conv` that will convolve two vectors (arrays) with each other. If `x,y` are vectors of length m, n respectively, then `conv(x,y)` will take time $O(mn)$ because Matlab uses the straightforward approach of multiplying all elements of `x` with all elements of `y`, and adding up these products appropriately.

1. Implement convolution faster by creating a function `myconv`, using Matlab's built-in `fft` implementation of the Fast Fourier Transform (and if you wish, `ifft`, which is the inverse). Do not worry about error checking: you may assume that both of your inputs are row vectors (that is, they have 1 row and many columns). For convolving two vectors of length n, your algorithm should take $O(n \log n)$ time – where only $O(n)$ time is spent in your code, and $O(n \log n)$ time is spent in Matlabs `fft` and `ifft` code. (This is much better than Matlab's built-in convolution, which takes time $O(n^2)$.) Remember, the lengths of the two input vectors may be different, and your code should work in this case too!

2. Matlab implements convolution this way because of concerns about accuracy. The fast Fourier transform is about as accurate as you might expect, however because in some sense it "smears" the inputs over its domain, the errors can be "smeared" in an unsatisfactory way. Figure out what this means by trying to find the most "embarrassing" example of your code's accuracy (as compared to `conv`'s accuracy) as possible. There is no template for this part: write and explain your answer using complete sentences.

3. The fast Fourier transform algorithm we saw in class works *only* for transforms whose size is a power of 2. There are a variety of related divide-and-conquer tricks to perform transforms of arbitrary sizes, but perhaps not surprisingly, these algorithms perform best when the transform size is a product of small primes (2,3,5). Think about how you can redesign your convolution code so that the size of each Fourier transform can be rounded up to a bigger more convenient size. Implement a function `roundup235` that rounds an integer up to the next product of powers of 2, 3, and 5. Running your code on a number n should take time *sublinear* in n, actually (hint!) something like $O(\log^3 n)$.

4. Write a new version of your convolution code, `myconv235`, that leverages `roundup235` so that each of your original `fft`'s and `ifft`'s is now a more efficient size.

5. Find the best speedup you can of your new code over your old code. (Try a configuration which would have produced Fourier Transforms whose size is a large prime to get the most embarrassing performance.) You can time things in Matlab with the `tic` and `toc` functions, which start and stop a timer, as in: `tic, myconv(x,y); toc`. Report your results in your write-up, in complete sentences as usual.

6. Using convolution (your code or Matlab's `conv`), write a function `bigmult` to multiply two large integers, assuming that each integer is represented as an array of digits. Keep in mind that convolution, as implemented via fft, may have numerical errors (though your code should return the exact answer). For example, `bigmult([5 0],[1 4])` should return `[7 0 0]`, indicating that $50 \cdot 14 = 700$. Your code should *not* return `[6.999999 0 0]`.

   As a hint, you might have to enlarge the array by 1, if carries add a digit to the product.

7. Assuming no limitations on memory, roughly what is the longest length of numbers your integer multiplication code should be able to handle? As usual, write this up in complete sentences.