Homework 2
Due: Sept. 22, 2015 at 6:00 PM (early)
Sept. 25, 2015 at 6:00 PM (on time)
Sept. 27, 2015 at 6:00 PM (late)

This is a partner homework as usual (reread the preamble of homework 1 to remind yourself of the basic rules): work on all problems together; you are responsible for everything you and your partner submit and it is an academic code violation to submit something that is not yours. Check for your partner in the Google doc that was sent out. If you don’t have a partner, please email the head TAs immediately.

Problems 1, 2, and 3 of this homework involve writing Matlab functions, which must be handed in electronically using the hand-in script. Run `cs157_handin hw2-p1` to copy everything from your current directory to our grading system as a submission for problem 1. (Change the p1 to p2 or p3 for the other problems!) Starter code for these problems is in the directory `/course/cs157/pub/stencils/` Please do not rename the files when you hand them in, or else your TA’s will be sad. Make sure your functions return their results by assigning to the designated output variable and not by printing to the screen; your code should not print anything to the screen (ending a line with a semicolon “;” in Matlab keeps the result from getting printed to the screen). Code will be graded automatically so there will be no style grades, and partial credit may not be given for code that does not work. Other parts of these problems involve writing explanations in complete sentences that must be handed in separately on paper to the hand-in bin as usual.

For Matlab help: talk to the TAs and check the class readings on the course webpage. Also, please make sure you have read the preamble of homework 1 before you continue.

Problem 1
(15 points total)

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.

1. (7 points) Devise an \( O(\log n) \) algorithm that receives such an array as an input and finds the maximum element in the array.
   For example, if the input array is:

   \[ 1 \ 4 \ 7 \ 8 \ 6 \ 3 \ 0 \]

   then the output should be 8.

   Implement your algorithm in Matlab by filling out the provided stencil `findmax.m`

2. (6 points) Prove the correctness of your code, introducing and explaining your algorithm as needed.

3. (2 points) Prove the “big-\( O \)” run-time of your code.
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

(15 points total)

Erek King, the pacifistic android, is intrigued by a certain class of matrices and wants to know how to multiply these matrices with vectors as fast as possible. Erek’s matrices are of sizes that are powers of 2 and are defined recursively as follows:

\[ E_1 = \begin{pmatrix} 1 \end{pmatrix} \]

\[ E_2 = \begin{pmatrix} 1 & -1 \\ 3 & 5 \end{pmatrix} \]

\[ E_4 = \begin{pmatrix} 1 & -1 & -1 & 1 \\ 3 & 5 & -3 & -5 \\ 3 & -3 & 5 & -5 \\ 9 & 15 & 15 & 25 \end{pmatrix} \]

\[ E_{2^k} = \begin{pmatrix} 1 \cdot E_{2^{k-1}} \\ 3 \cdot E_{2^{k-1}} \\ -1 \cdot E_{2^{k-1}} \\ 5 \cdot E_{2^{k-1}} \end{pmatrix} \]

Find an algorithm that, when \( n \) is a power of 2, lets Erek multiply \( E_n \) by length \( n \) vectors in \( O(n \log n) \) time. This is much faster than the \( O(n^2) \) time it would take to compute the product using standard linear algebra. (Note: Do not construct the matrices \( E \) in your code, as even constructing these matrices will take \( O(n^2) \) time, which is too much!)

1. (6 points) Fill out the Matlab stencil code \texttt{erek.m}

2. (5 points) Prove correctness of your algorithm (explaining your algorithm as needed).

3. (4 points) Prove the run-time of your algorithm.

(Hint: The structure of this algorithm is rather similar to the algorithm for computing the Fast Fourier Transform. We hope that solving this problem will help you understand the \texttt{fft} algorithm a bit better.)

Problem 3

(25 points total)

Given two vectors \((a_0, a_1, a_2, \ldots, a_i)\) and \((b_0, b_1, b_2, \ldots, b_j)\) that we consider as describing two polynomials, of degrees \( i \) and \( j \) respectively, \( a_0 + a_1 x + a_2 x^2 + \ldots + a_i x^i \) and \( b_0 + b_1 x + b_2 x^2 + \ldots + b_j x^j \), then the vector of coefficients of the product of these two polynomials, \( a_0 b_0 + (a_1 b_0 + a_0 b_1) x + \)
\[(a_2b_0 + a_1b_1 + a_0b_2)x^2 + \ldots + a_ib_jx^{i+j}\] is known as the convolution of the two vectors. Convolution is often denoted "\(*\)". Namely, \[(a_0, a_1, a_2, \ldots, a_i) \ast (b_0, b_1, b_2, \ldots, b_j) = (a_0b_0, (a_1b_0 + a_0b_1), (a_2b_0 + a_1b_1 + a_0b_2), \ldots, a_ib_j).\]

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. (4 points) 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 Matlab’s `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, and return an answer that always has exactly the same length as the standard `conv`!

2. (3 points) Matlab implements convolution this way because of concerns about accuracy. While the fast Fourier transform is mathematically precise, computers use floating-point representation for real numbers. Because these representations have a fixed, \(n\) bit length (\(n\) is 64 in Matlab, by default), they can only represent \(2^n\) numbers, which results in some implicit aliasing and rounding of numbers. Taking the Fourier transform of a vector with disproportionate elements exaggerates this effect and can cause some of the elements to get swallowed by this limited numeric precision. To see this in action, take the FFT of the vector \((1, 10^{-50})\) by hand, then take the inverse FFT. Now do the same in Matlab (you may want to type `format shortg` before you start, so that Matlab displays small numbers in a more useful manner—"short" asks Matlab to display roughly 5 digits; "g" corresponds to the "\%g" format in the C/C++ `printf` command). Explain any differences between the results. There is no template for this part: write and explain your answer using complete sentences.

3. (6 points) 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 \(O(\log^3 n)\). Be sure to test your code. For example, `roundup235(60)` should return 60, since 60 is already a product of powers of 2, 3, and 5; `roundup235(76)` should return 80 because none of the numbers 76 through 79 are products of powers of 2, 3, and 5.

4. (3 points) 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. (Be sure your output
has the correct length—the output here should always be identical to the output produced by \texttt{conv}, except for tiny rounding errors.)

5. (2 points) 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. Some large primes are 999,983 and 10,000,019. Matlab’s \texttt{factor} function can help you find more primes if you are curious.) You can time things in Matlab with the \texttt{tic} and \texttt{toc} functions, which start and stop a timer, as in: \texttt{tic, myconv(x,y); toc}. Report your results in your write-up, in complete sentences as usual.

6. (7 points) Multiplying large integers is no more than convolving their vectors of digits, and then “carrying” as appropriate. Using convolution (your code or Matlab’s \texttt{conv}), write a function \texttt{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 \texttt{fft}, may have tiny numerical errors (though your code must work despite this!). For example, \texttt{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 “carrying” adds a digit to the product. \textbf{Make sure you test your code on very large numbers.} For example, the thousand digit number consisting of all 1’s, when multiplied by itself:

\[
\text{bigmult(ones(1,1000),ones(1,1000))}
\]

should produce a 1999 digit number consisting of 111 repetitions of the digits 123456790 followed by 111 repetitions of the digits 098765432 followed by a single 1. Make sure your code can reproduce this. (Note: Despite all the above warnings, half of the class last year got all of our harder test cases wrong. Can you do better? “Carrying” is something that you maybe have not paid attention to since you were 6 years old, but now is the time! Make sure you design a carrying algorithm that is efficient and correct even when carrying intermediate results that are millions big, across millions of digits.)

\textbf{Problem 4}

(15 points total)

This problem is a chance for you to prove for yourself some of the basics about Fourier transforms that we covered in class.

1. (7 points) For a certain \(n\), letting \(\omega = (1 - \frac{2\pi}{n}) = e^{-2\pi i/n}\), we define the Fourier transform of an \(n\)-element vector as its product with the \(n \times n\) matrix

\[
M = \begin{pmatrix}
1 & 1 & 1 & \cdots \\
1 & \omega & \omega^2 & \omega^3 & \cdots \\
1 & \omega^2 & \omega^4 & \omega^6 & \cdots \\
1 & \omega^3 & \omega^6 & \omega^9 & \cdots \\
\vdots & \vdots & \vdots & \vdots & \ddots 
\end{pmatrix}
\]
Show for any \( n \), that \( M \) is “almost its own inverse”, by showing that the product of \( M \) with itself, \( M \cdot M \), equals \( n \) times the matrix with 1 in the top left corner, and 1s along the opposite diagonal—for \( n=5 \), the matrix
\[
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0
\end{pmatrix}
\]

(Hint: You will need to consider the dot product of a row of \( M \) with a column of \( M \) and show that it is either 0 or \( n \), depending on the combination of row and column. The formula for the sum of a geometric series will be useful here. You might show that the element-by-element product of a row and a column of \( M \) is itself a row of \( M \), so that all you need to do is analyze the sum of each row of \( M \). Can you explain intuitively why the row sums are what they are?)

2. (8 points) Explain how, if you consider the entries of a vector as coefficients of a degree \( n-1 \) polynomial, then the Fourier transform corresponds to evaluating the polynomial at each of the points \( 1, \omega, \omega^2, \omega^3, \ldots, \omega^{n-1} \). Prove, from this, that you can compute the product of two polynomials \( a \) and \( b \) of degrees \( i \) and \( j \) respectively, via the Matlab code
\[
\text{ifft( fft(a) .* fft(b) )}
\]
provided the polynomials are padded with zeros so that the vectors have length at least \( i+j+1 \). (Remember that \( ".*" \) is Matlab’s notation for element-by-element multiplication.)

(Hint: Use the fact that the coefficients of a polynomial of degree up to \( n-1 \) are uniquely determined by the values of the polynomial at \( n \) points—even complex points such as \( \omega^i \) (this is often called “the uniqueness of polynomial interpolation”). You may assume that Matlab’s \texttt{fft} function correctly computes the Fourier transform, and that \texttt{ifft} correctly inverts this process.)

Problem 5

(30 points total)

Divide and conquer algorithms are often useful when processing trees. In this problem we will work through an example that combines tools from divide+conquer and dynamic programming.

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 \( k \) might be big, so this is not what we are aiming for.) For each of the following parts that asks for an algorithm, a proof of correctness and runtime is also needed.

1. (5 points) 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 \textit{starts at the root}, and goes down from there. (Hint: dynamic programming.)

2. (5 points) 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 \textit{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. (5 points) Now solve the previous part \textit{without} restricting the number of children of the root. The algorithm should still take \(O(n)\) running time.

\textbf{Warning or hint:} Make sure your path does \textit{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. (5 points) 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. (5 points) 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. (5 points) 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.

\textbf{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.