Homework 2: OpenSSN

Due: Sept. 18, 2018 at 6:00 PM (early, 5% bonus)
Sept. 21, 2018 at 6:00 PM (on time)
Sept. 23, 2018 at 6:00 PM (late, 20% penalty)

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 TAs 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 via Gradescope as usual (see Piazza for instructions).

For Matlab help: come to the Sunlab 7-9pm this Sunday night (Sept. 17), or talk to the TAs, or check the class readings on the course webpage. *Also, please make sure you have read the preamble of homework 1 before you continue.*

Make sure that you clearly indicate on each problem which track you're in (either **1-credit** or **full**). **1-credit track:** do problems 2; 3.1 and 3.6; all parts of problem 5 except 5.3 (5.1, 5.2, 5.4, and 5.5) *and indicate "***1-credit***" on each problem.*

It's like Venmo, but for Social Security Numbers!

Our previous app, **BokChain** would have worked perfectly, but it turns out that our VC investor is allergic to cabbage. We've decided to pivot...

Stop us if any of these has happened to you. A website asks for your social security number before you read an article, but you can't remember it. That one friend keeps hitting you up for the last four digits of your credit card, but you left it in your other pants. Random people on the internet keep asking you for your bank account information and it's getting annoying to type it out over and over again. Wouldn't it be great if everyone already knew what your number was?

Enter **OpenSSN**[1], our revolutionary crowd-sourced service for sharing personal information on the fly! We believe that our combination of blockchain technology and deep learning architecture will totally disrupt the personal identification industry and make the world a better place.

We'll handle the business side, but we need you to tackle the backend.

[1] Unrelated to `http://openssn.sourceforge.net/`

Problem 1

(15 points total)

> According do our legal counsel, *apparently* it's a felony to share social security numbers over the internet. To bypass this unfortunate obstacle, we've decided to leverage bluetooth technology. All a user has to do is walk close enough to our patented unmarked **OpenSSN** reception vans and voila! Their personal information will become uploaded to our databases. **OpenSSN** vans leverage a novel algorithm for locating passerbys in their general vicinities whose information is available for uploading! Our current algorithm properly detects distances of users from the van, but a bug in our system causes the numbers to appear in a weirdly specific order. We need to you reimplement the algorithm to fix it and show our users where the closest ~~victim~~ customer is.

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 (which may equal 1 or n), 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` (there is no paper handin for this part).

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 the crucial ingredients and how they relate to each other in a concise manner, then your argument will be more convincing and pleasant to read than a writeup with all the low-level details jumbled together.

Problem 2

(15 points total)

> **OpenSSN** implements a *mostly* legal algorithm that leverages dark web services to store and retrieve personal information. One of its main components involves multiplying a matrix of user data with a vector of builtin weights. Unfortunately, the retrieval algorithm currently takes way too long to complete; we've been notified by the San Francisco police department that our vans are spending suspiciously long amounts of time tailing random people waiting for their numbers to upload. We need you to speed up the algorithm.

Consider the set of matrices, all of which are of sizes that are powers of 2 and are defined recursively as follows:

$$B_1 = \begin{pmatrix} 1 \end{pmatrix}$$

$$B_2 = \begin{pmatrix} 1 & -1 \\ 3 & 2 \end{pmatrix}$$

$$B_4 = \begin{pmatrix} 1 & -1 & -1 & 1 \\ 3 & 2 & -3 & -2 \\ 3 & -3 & 2 & -2 \\ 9 & 6 & 6 & 4 \end{pmatrix}$$

$$\vdots$$

$$B_{2^k} = \left(\begin{array}{c|c} 1 \cdot B_{2^{k-1}} & -1 \cdot B_{2^{k-1}} \\ \hline 3 \cdot B_{2^{k-1}} & 2 \cdot B_{2^{k-1}} \end{array}\right)$$

Find an algorithm that, when n is a power of 2, lets our app multiply B_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 B 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 `bigdata.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 `fft` algorithm a bit better. You might check out the textbook's description of the FFT algorithm, section 2.6.4 of `https://people.eecs.berkeley.edu/~vazirani/algorithms/chap2.pdf`)

Problem 3

(25 points total)

Consider two polynomials a, b, of degrees m and n respectively, whose coefficients are described by the vectors $(a_0, a_1, a_2, \ldots, a_m)$ and $(b_0, b_1, b_2, \ldots, b_n)$. In other words, $a = a_0 + a_1x + a_2x^2 + \ldots + a_mx^m$ and $b = b_0 + b_1x + b_2x^2 + \ldots + b_nx^n$. The vector of coefficients of the *product* of these two polynomials, $a_0b_0 + (a_1b_0 + a_0b_1)x + (a_2b_0 + a_1b_1 + a_0b_2)x^2 + \ldots + a_mb_nx^{m+n}$ is known as the *convolution* of the two vectors. Convolution is often denoted "$*$" by mathematicians (but unfortunately, not by computers). Namely,

$$(a_0, a_1, a_2, \ldots, a_m) * (b_0, b_1, b_2, \ldots, b_n) = (a_0b_0\,,\ (a_1b_0 + a_0b_1)\,,\ (a_2b_0 + a_1b_1 + a_0b_2)\,,\ \ldots\,,\ a_mb_n).$$

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 must 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 `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 `factor` function can help you find more primes if you are curious.) 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. (7 points) Multiplying large integers is no more than convolving their vectors of digits, and then "carrying" as appropriate. Using your `myconv` convolution function, 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 tiny numerical errors (though your code must work despite this!). 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 "carrying" adds a digit to the product. **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:

   ```
   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. Do not just rely on test cases—try to prove to yourself that your code will be correct in all cases.)

Problem 4

There are TWO completely different options here. Choose and do ONE of them. OPTION A mathematically derives properties of the Fourier transform we discuss in class and use in problem 3; OPTION B, entirely different, is about finding counterexamples to dynamic programming algorithms - try your hand at breaking incorrect algorithms. Choose the one that best aligns with your interests.

(15 points total)

OPTION A:

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

$$M = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots \\ 1 & \omega & \omega^2 & \omega^3 & \cdots \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

   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 $n = 5$, we have

$$M \cdot M = 5 \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

   (**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 `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 ω^j (this is often called ''the uniqueness of polynomial interpolation"). You may assume that Matlab's `fft` function correctly computes the Fourier transform, and that `ifft` correctly inverts this process.)

OPTION B:

Try to find a counterexample as described in each part below. We strongly suggest you solve each part by hand, to give you insight, as opposed to coding up the algorithms and letting a computer blindly run test cases until something breaks. Finding counterexamples to code might be a new skill for you. See hints below.

For each part, provide 1) a brief intuitive explanation of *why* the algorithm breaks for your counterexample, along with **2)** enough of a dynamic programming table filled out (and explanation as needed) to convince the reader that it does indeed break. (As always in this course, consider drawing tables/diagrams by hand, even if everything else is typed.)

1. (6 points) Recall the dynamic programming algorithm for finding the edit distance between word1 and word2. The algorithm fills out a 2 dimensional table T, where the meaning of entry $T(i, j)$ should be "the minimum cost of editing the first i characters of word1 into the first j characters of word2." The algorithm works by filling out a table initialized as $T(i, 0) = i$ and $T(0, j) = j$, with the internal entries defined by the recurrence

$$T(i,j) = \min \begin{cases} T(i-1, j-1) + [\text{word1}(i) \stackrel{?}{\neq} \text{word2}(j)], \\ T(i-1, j) + 1, \\ T(i, j-1) + 1 \end{cases}$$

   Suppose that there is an error in the initialization, and, mistakenly, $T(0, 2) = 17$ (instead of 2). Find two words of length 5 such that the algorithm will compute the *wrong* edit distance in entry $T(5, 5)$.

   Hint: What is the meaning of the wrong entry $T(0, 2)$? When, broadly, will the recurrence rely on this entry? How can you find two words that lead to this occurring?

2. (3 points) Recall the Bellman-Ford algorithm for finding the distance between two nodes on a graph. Given a starting node i, the algorithm fills out a 2 dimensional table T, where entry $T(j, k)$ represents "the cost of the best path from i to j using up to k intermediate nodes. The table is initialized by setting $T(j, 0) = len(i, j)$ for each j where there is an edge from i to j, and infinity otherwise. The recurrence relation for $k > 0$ is:

$$T(j,k) = \min \begin{cases} T(j, k-1) \\ \min\limits_{s \text{ with an edge } s \to j} T(s, k-1) + len(s \to j) \end{cases} \tag{1}$$

   For an n node graph, this recurrence should be run inside a loop that goes over all vertices j, inside a loop that goes from $k = 1$ to $k = n - 2$ to ensure all possible paths are considered. Find a graph of size $n = 6$ with only positive edge weights, and nodes i, j, such that if the outer loop only goes from $k = 1$ to $k = n - 3 = 3$, then the last entry in the row $T(j, \cdot)$ will be *not* correctly record the distance from i to j.

   Hint: What is the meaning of the entries with $k = n - 2$ that the modified algorithm never computes? For what graph, i, j, are these entries crucial for finding the distance from i to j?

   (**Note:** The Bellman-Ford algorithm actually needs an additional $k = n - 1$ iteration to check for "negative cost cycles", which makes path costs negative infinity, by allowing one to repeatedly go around this cycle an arbitrary number of times. This is not relevant for this problem, as we consider only positive costs.)

3. (6 points) Wikipedia's version of Bellman-Ford is slightly more efficient: it only uses a 1-dimensional table $T(j)$, with the same recurrence as above except ignoring the role of k:

$$T(j) = \min \begin{cases} T(j) \\ \min\limits_{s \text{ with an edge } s \to j} T(s) + len(s \to j) \end{cases} \tag{2}$$

The meaning of $T(j)$ at the k^{th} iteration is more vague, essentially "the length of the best path found so far from i to j, and at least as good as the path found in the k^{th} iteration of the algorithm of the previous part".

Running the recurrence of Equation 2 inside a loop that goes over all nodes from $j = 1$ to $j = n$, inside an outer loop that goes from $k = 1$ to $k = n - 3 = 3$, find a 6-node graph using only positive edge weights that fools even this more efficient algorithm. Namely, find an i and j in the graph, such that $T(j)$ at the end of the algorithm will not be the length of the shortest path from i to j.

Hint: Can you adapt your counterexample from part 2 so that it fails in the same way even for this modified recurrence?

Problem 5

(30 points total)

> Unfortunately, we have been informed that some mobile phone users would rather not be a part of our personal information revolution. Users have begun downloading the app **ClosedSSN**, which provides a firewall for their bluetooth connectivity that keeps our vans from borrowing information. Luckily, we have acquired user data from the new app that shows which neighborhoods have high concentrations of phones with **ClosedSSN** downloaded.
> We need you to implement a new algorithm for planning routes for our vans through these neighborhoods (which are conveniently shaped like search trees) so as to maximize the amount of user data we can upload.

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 (i.e. recording results of computations for future use).

Given a tree with n vertices that has weighted edges, and a positive integer $k \leq n$, 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. (3 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 *starts at the root*, and goes down from there.

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 *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 *without* restricting the number of children of the root. The algorithm should still take $O(n)$ running time. (You cannot just reuse part 2 on all pairs of children of the root, as this could take quadratic time or k memory-per-child-of-the-root, which would be too slow.)

   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. (10 points)

   (a) (4 points) Find an $O(n)$ algorithm that, given a tree with n vertices, returns a vertex v such that, if we remove v, the tree is split into several smaller trees, each of which has at most $\frac{n}{2}$ vertices.

   (b) (4 points) Prove that your algorithm always finds such a vertex, which will also prove that such a vertex always exists. (Reminder: avoid using the word "correct" in your writeup, which is often too vague; instead be explicit about what you are proving.)

   (c) (2 points) Prove that your algorithm has runtime $O(n)$.

5. (7 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.

   Note: You do not have to solve parts 3 and 4 to get credit for part 5: in part 5 you may assume that algorithms for parts 3 and 4 exist and they work correctly as indicated.