CSE 21 Fall 2018
Homework 4
Due: Sunday, October 28, 2018 at 10:00pm

Instructions

Homework should be done in groups of **one to three** people. You are free to change group members at any time throughout the quarter. Problems should be solved together, not divided up between partners. Homework must be submitted through **Gradescope** by **a single representative**. Submissions must be received by **10:00pm** on the due date, and there are no exceptions to this rule.

You will be able to look at your scanned work before submitting it. Please ensure that your submission is **legible** (neatly written and not too faint) or your homework may not be graded.

Students should consult their textbook, class notes, lecture slides, instructors, TAs, and tutors when they need help with homework. Students should not look for answers to homework problems in other texts or sources, including the Internet. You may ask questions about the homework in office hours, but **not on Piazza**.

Your assignments in this class will be evaluated not only on the correctness of your answers, but on your ability to present your ideas clearly and logically. You should **always explain** how you arrived at your conclusions and **justify your answers** with mathematically sound reasoning. Whether you use formal proof techniques or write a more informal argument for why something is true, your answers should always be well-supported. Your goal should be to **convince the reader** that your results and methods are sound.

For questions that require pseudocode, you can follow the same format as the textbook, or you can write pseudocode in your own style, as long as you specify what your notation means. For example, are you using "=" to mean assignment or to check equality? You are welcome to use any algorithm from class as a subroutine in your pseudocode. For example, if you want to sort list A using InsertionSort, you can call InsertionSort(A) instead of writing out the pseudocode for InsertionSort.

Required Reading Rosen Chapter 5 and Sections 8.1-8.3.

Key Concepts: Recursive algorithms (definitions, proofs of correctness, runtime calculations), Karatsuba's method, Master Theorem, Ordinary Generating Functions.

---

1. In a tennis tournament, there are n players, where $n = 2^k$ for some positive integer k. In the first round, each player competes against another player. Only the winners advance to the second round. In the second round, each remaining player competes against another player, and only the winners advance to the third round. This process continues until a single winner is determined. Let $f(n)$ represent the total number of matches played in the tournament.

   (a) (5 points) Give a recurrence for $f(n)$ of the form

   $$f(n) = Af(n/B) + C, \qquad f(1) = D$$

   where A, B, C, and D are constants. Briefly explain in words why this recurrence describes the number of matches in the tennis tournament. Use the Master Theorem on this recurrence to find a Big O description of the number of matches.

   Solution: We can think of a tournament with n players as two separate mini-tournaments, each with $n/2$ players. A winner from each mini-tournament is determined, then these two winners face off in one additional match (the finals) to determine the overall winner of the tournament. In addition, a tournament with only one player will not have any matches. Therefore, a recurrence is

   $$f(n) = 2f(n/2) + 1, \text{ with } f(1) = 0$$

   .

   We can apply the Master Theorem with $a = 2, b = 2, d = 0$. Since $a > b^d$, we have that $f(n)$ is $O(n^{\log_b a}) = O(n^{\log_2 2}) = O(n)$.

   (b) (5 points) Give a recurrence for $f(n)$ of the form

   $$f(n) = Ef(n/F) + Gn, \qquad f(1) = H$$

   where E, F, G, and H are constants. Briefly explain in words why this recurrence describes the number of matches in the tennis tournament. Use the Master Theorem on this recurrence to show that you get the same Big O description as in part (a).

   Solution: Alternatively, we can think of a tournament with n players as having a first round, where everyone pairs up and plays their opponent. Only the winners of this first round will advance, and so after that first round, we essentially have another tournament, but with only half as many players. Since $n/2$ matches happen in the first round, a recurrence is

   $$f(n) = f(n/2) + \frac{1}{2}n, \text{ with } f(1) = 0$$

   .

   We can apply the Master Theorem with $a = 1, b = 2, d = 1$. Since $a < b^d$, we have that $f(n)$ is $O(n^d) = O(n)$.

   (c) (5 points) Find a closed-form formula for $f(n)$ using any method covered in this class.

   Solution: We can use either the recurrence from (a) or the recurrence from (b) to generate a table of values for $f(n)$. This leads to the guess that a formula for $f(n)$ may be $f(n) = n - 1$.

n	1	2	3	4	5
f(n)	0	1	2	3	4

We can prove this guess by strong induction on n. We'll use the recurrence from part (a). For the base case, when $n = 1$, we have $f(1) = 0$ as given by the recurrence and $1-1 = 0$, so the formula holds. Assume as our inductive hypothesis that $f(k) = k-1$ for any value of k between 1 and $n-1$. Then $f(n) = 2f(n/2)+1$ by the recursive formula, and we know from applying the inductive hypothesis with a value of $k = n/2$ that $f(n/2) = n/2 - 1$, so $f(n) = 2(n/2 - 1) + 1 = n - 1$ Thus, our formula is correct.

2. Given two polynomials of degree $n-1$: $A(x) = \sum_{i=0}^{n-1} a_i x^i$ and $B(x) = \sum_{i=0}^{n-1} b_i x^i$. The product of these two polynomials is another polynomial $C(x)$ of degree *up to* $2n-2$ given by

$$C(x) = \sum_{i=0}^{2n-2} c_i x^i \quad \text{where} \quad c_i = \sum_{k=0}^{i} a_k b_{i-k}.$$

For this problem, we are given the coefficients for $A(x) = \sum_{i=0}^{n-1} a_i x^i$ and $B(x) = \sum_{i=0}^{n-1} b_i x^i$ as two arrays $A[0, \ldots, n-1] = [a_0, \ldots, a_{n-1}]$ and $B[0, \ldots, n-1] = [b_0, \ldots, b_{n-1}]$, respectively. We want to compute the product $C(x) := A(x)B(x)$ and return the coefficient lists $C[0, \ldots, 2n-2] = [c_0, \ldots, c_{2n-2}]$.

For all parts, you are not required to prove correctness of your algorithms, but are required to give a time analysis in O form.

(a) (5 points) Give the straight-forward, iterative algorithm for this problem.

Solution. In the iterative algorithm, we multiply each term of the first polynomial with each term of the second. In terms of arrays, the product might have degree $2n-2$, so we'll use $c_0, \ldots, c_{2n-2}$ to store is coefficients of the product.

In pseudocode, this looks like:

procedure *IterativeProduct* ($A[0, \ldots, n-1]$, $B[0, \ldots, n-1]$)

1. **for** $i = 0$ **to** $2n-2$ **do**: initialize c_i to 0.
2. **for** $i = 0$ **to** $n-1$
3. **for** $j = 0$ **to** $n-1$
4. $c_{i+j} := c_{i+j} + a_i b_j$;
5. **return** $c_0, \ldots, c_{2n-2}$

Since the two nested loops each iterate exactly n times, the total time is $\Theta(n^2)$ (the runtime for lines 1 and 5 are negligible).

(b) (10 points) Give a simple divide-and-conquer algorithm for this problem.

Solution. Let $A(x) = \sum_{i=0}^{n-1} a_i x^i$ and $B(x) = \sum_{i=0}^{n-1} b_i x^i$ be two polynomials. The base case is when $n = 1$ where $A(x)$ and $B(x)$ are degree zero polynomials, i.e. constants. So in this case, we simply multiply two constant a_0 and b_0. Thus, the output is a single constant $c_0 = a_0 b_0$.

If $n > 1$, then the idea of divide and conquer suggests that we break each $A(x)$ and $B(x)$ up into two polynomials of degree m where $m = n/2 - 1$. However, in order to achieve this, we require n to be an even number. If n is odd, then we first need to increment n by one and let $a_0 = b_0 = 0$. Specifically, for $A(x)$, we set

$$A_L(x) = \sum_{i=0}^{m} a_i x^i \text{ and } A_H(x) = \sum_{i=m+1}^{n} a_i x^{m+1-i} \text{ so that } A(x) = A_L(x) + x^{m+1} A_H(x).$$

Similarly for $B(x)$,

$$B_L(x) = \sum_{i=0}^{m} b_i x^i \text{ and } B_H(x) = \sum_{i=m+1}^{n} b_i x^{m+1-i} \text{ so that } B(x) = B_L(x) + x^{m+1} B_H(x).$$

In terms of coefficient lists, we have

$$A_L[0,\ldots,n/2-1] = [a_0,\ldots a_{n/2-1}] A_H[0,\ldots,n/2-1] = [a_{n/2},\ldots,a_n]$$
$$B_L[0,\ldots,n/2-1] = [b_0,\ldots b_{n/2-1}] B_H[0,\ldots,n/2-1] = [b_{n/2},\ldots,b_n]$$

Then

$$\begin{aligned} A(x)B(x) &= \left(A_L(x) + x^{m+1}A_H(x)\right)\left(B_L(x) + x^{m+1}B_H(x)\right) \\ &= A_L(x)B_L(x) + x^{m+1}\left[A_L(x)B_H(x) + A_H(x)B_L(x)\right] + x^{2m+2}A_H(x)B_H(x) \\ &= A_L(x)B_L(x) + x^{n/2}\left[A_L(x)B_H(x) + A_H(x)B_L(x)\right] + x^n A_H(x)B_H(x) \end{aligned}$$

Next, we need several subroutine (applied on arrays of coefficients):

- *Add* adds the values of corresponding elements in the two arrays (if the arrays have different dimensions, *Add* will treat the smaller dimension array as having 0's in the larger dimensions)
- $MultiplyPower(A, m)$ shifts an array A over by m positions by adding m 0's at the beginning.

Notice that both of the above subroutines have **linear** runtime.

Finally, we can now describe the pseudocode as follows.

procedure *RecProduct* $(A[0,\ldots,n-1]\ ,\ B[0,\ldots,n-1])$

1. **if** $n = 0$ **then return** $A[0]B[0]$
2. **if** n is odd **then** define $A[n] = B[n] = 0$ and increment n
3. $AL[0,\ldots,n/2-1] := A[0,\ldots,n/2-1]; AH[0,\ldots,n/2-1] := A[n/2,\ldots,n-1];$
4. $BL[0,\ldots,n/2-1] := B[0,\ldots,n/2-1]; BH[0,\ldots,n/2-1] := B[n/2,\ldots,n-1];$
5. $H[0,\ldots,n-2] := RecProduct(AH, BH);$
6. $H[0,\ldots,2n-2] := MultiplyPower(H[0,\ldots,n-2], n);$
7. $L[0,\ldots,n-2] := RecProduct(AL, BL);$
8. $M[0,\ldots,n-2] := Add(RecProduct(AH, BL), RecProduct(AL, BH))$
9. $M[0,\ldots,3n/2-2] := MultiplyPower(M[0,\ldots,n-2], n/2);$
10. **return** $Add(\ Add(H[0,\ldots,2n-2],\ M[0,\ldots,3n/2-2]),\ L[0,\ldots,n-2])$

Since both *Add* and *MultiplyPower* are linear time, as is copying over the subarrays, and we make 4 recursive calls in lines 5, 7, and 8 (twice) to multiply arrays of dimension $n/2$, this gives us the recurrence

$$T(n) = 4T(n/2) + O(n).$$

Using the Master Theorem with $a = 4, b = 2, d = 1$, we are in the bottom-heavy case, and $T(n) = O(n^{log_b a}) = O(n^{log_2 4}) = O(n^2)$, just like the iterative algorithm.

(c) (10 points) Use the Karatsuba method to give an improved divide-and-conquer algorithm for this problem.

Solution. Karatsuba's method is based on the polynomial identity:

$$\begin{aligned} &(A_L(x) + A_H(x))(B_L(x) + B_H(x)) \\ &\qquad = A_L(x)B_L(x) + A_H(x)B_L(x) + A_L(x)B_H(x) + A_H B_H(x). \end{aligned}$$

This implies

$$A_H(x)B_L(x) + A_L(x)B_H(x)$$
$$= (A_L(x) + A_H(x))(B_L(x) + B_H(x)) - A_L(x)B_L(x) - A_H(x)B_H(x).$$

So to get the middle two terms, of

$$A(x)B(x) = A_L(x)B_L(x) + x^{n/2}\left[A_L(x)B_H(x) + A_H(x)B_L(x)\right] + x^n A_H(x)B_H(x),$$

we can compute the product $(A_L(x) + A_H(x))(B_L(x) + B_H(x))$, and subtract off the $A_L(x)B_L(x)$ and $A_H(x)B_H(x)$ terms that we computed earlier.

Here, we shall use another subroutine, *Subtract* to subtracts two arrays coefficient by coefficient, similar to *Add* from the previous part. This subroutine also has linear runtime.

procedure $KaraProduct$ $(A[0,\ldots,n-1]$, $B[0,\ldots,n-1])$

1. **if** $n = 0$ **then return** $A[0]B[0]$
2. **if** n is odd **then** define $A[n] = B[n] = 0$ and increment n
3. $AL[0,\ldots,n/2-1] := A[0,\ldots,n/2-1]; AH[0,\ldots,n/2-1] := A[n/2,\ldots,n-1];$
4. $BL[0,\ldots,n/2-1] := B[0,\ldots,n/2-1]; BH[0,\ldots,n/2-1] := B[n/2,\ldots,n-1];$
5. $H[0,\ldots,n-2] := KaraProduct(AH, BH);$
6. $L[0,\ldots,n-2] := KaraProduct(AL, BL);$
7. $S[0,\ldots,n-2] := KaraProduct(Add(AH, AL), Add(BH, BL))$
8. $M[0,\ldots,n-2] = Subtract(S, Add(L, H));$
9. $M[0,\ldots,3n/2-2] := MultiplyPower(M[0,\ldots,n-2], n/2);$
10. $H[0,\ldots,2n-2] := MultiplyPower(H[0,\ldots,n-2], n);$
11. **return** $Add(\ Add(H[0,\ldots,2n-2],\ M[0,\ldots,3n/2-2]),\ L[0,\ldots,n-2])$

Here, we make 3 recursive calls to arrays of size $n/2$, and add, subtract, and shift a fixed number of times. So the recurrence is

$$T(n) = 3T(n/2) + O(n).$$

Since $3 > 2^1$, we are still in the bottom-heavy case and the time is $T(n) \in \Theta(n^{\log_2 3}) \subseteq O(n^{1.6})$. So we've improved the running time substantially.

3. (Optional) Let $a_0 = 1$ and $a_{n+1} = 3a_n + 2^n$ for $n \geq 0$.

   (a) Find a closed-form formula for the generating function $A(x) = \sum_{n\geq 0} a_n x^n$.

   (b) Use your generating function in part (a) to find the closed-form formula for a_n. Here, you may need to use partial fraction decomposition.

Solution:

$$\begin{aligned} A(x) &= \sum_{n\geq 0} a_n x^n = 1 + \sum_{n\geq 1} a_n x^n = 1 + \sum_{n\geq 1} \left(3a_{n-1} + 2^{n-1}\right) x^n \\ &= 1 + 3x \sum_{n\geq 1} a_{n-1} x^{n-1} + x \sum_{n\geq 1} 2^{n-1} x^{n-1} \\ &= 1 + 3xA(x) + \frac{x}{1-2x} = \frac{1-x}{1-2x} + 3xA(x). \end{aligned}$$

Solving the equation $A(x) = \dfrac{1-x}{1-2x} + 3xA(x)$ for $A(x)$ to obtain

$$A(x) = \frac{1-x}{(1-2x)(1-3x)}.$$

Now to obtain the coefficients a_n of $A(x)$, we use partial fraction to write $A(x)$ into

$$A(x) = \frac{1-x}{(1-2x)(1-3x)} = \frac{2}{1-3x} - \frac{1}{1-2x}.$$

Thus,

$$A(x) = \frac{2}{1-3x} - \frac{1}{1-2x} = \sum_{n\geq 0} 2(3^n)x^n - \sum_{n\geq 0} (2^n)x^n = \sum_{n\geq 0} \left(2 \cdot 3^n - 2^n\right) x^n.$$

Therefore, $a_n = 2 \cdot 3^n - 2^n$ for all $n \geq 0$.