Problem Set 2 Solutions

This problem set is due **at 11:59pm** on **Thursday, September 22, 2016.**

EXERCISES (NOT TO BE TURNED IN)

Divide and Conquer Algorithms

- Do Exercise 9.3-2 in CLRS on page 223.
- Do Exercise 9.3-7 in CLRS on page 223.
- Do Exercise 9.3-9 in CLRS on page 223.

The Fast Fourier Transform

- Do Exercise 30.1-1 in CLRS on page 905.
- Do Exercise 30.1-4 in CLRS on page 905.
- Do Exercise 30.2-4 in CLRS on page 914.

Problem 2-1. Sorting Heights [50 points] As part of an ongoing effort to collect essential statistics at the Xavier Institute for Higher Learning, Professor Xavier has been tasked with creating a sorted list of all the students at the school by their height. The school has n teachers and each teacher has m students. By order of Professor Xavier, each teacher has individually sorted their students by height and sent their results to the Professor. Professor Xavier now has n sorted lists $\{L_1, L_2, .. L_n\}$, each with m elements. Help Professor X combine the lists into one sorted list of heights.

(a) [15 points] Recall the merge step from Mergesort. Suppose Professor X merged the sorted lists in the following way. Use the merge step from Mergesort to merge lists L_1 and L_2. Then, continue to merge the resulting list with L_3, and so on until all lists are merged into one sorted list. What is the runtime analysis for his procedure?

Solution: It first takes $2m$ steps to merge L_1 and L_2, where the resulting list will have size $2m$. Subsequently, each iteration i of merging costs $s_i + m$ where s_i is the size of the list before iteration i. Hence the total running time is given by $\sum_{i=1}^{n} i \cdot m = O(n^2 m)$.

(b) [35 points] Show how to use divide and conquer to improve on this runtime.

Solution: We use the following divide and conquer algorithm.

1. If $n = 1$, return the list, otherwise continue.
2. Arbitrarily split the set of n lists into two sets of size $n/2$ each.
3. Recursively merge both sets to obtain two sorted lists of size $mn/2$ each.
4. Use the Mergesort merging algorithm to combine the two lists into one final sorted list, which takes $O(mn)$ steps.

The total runtime for this algorithm can be given by the recurrence

$$T(n, m) = 2T(n/2, m) + O(nm).$$

Notice that even though this is a recurrence in two variables, m is always constant. The recurrence tree has $O(\log n)$ levels with $O(mn)$ cost per level. Therefore the total runtime of the algorithm is $O(mn \log n)$.

Problem 2-2. Custom Polynomial [50 points] You are one of Professor Xavier's students at the Xavier Institute for Higher Learning. Earlier in the year, Professor X had given you a degree n polynomial in coefficient form, with leading coefficient 1, to guard with your life. However, sometime during the semester, you lost the piece of paper that he gave to you with the coefficients on it. Luckily, you remember all n zeros of the polynomial $x_1, x_2, .. x_n$ which are all distinct.

Recall that a value x_j is a zero for polynomial $P(x)$ if $P(x_j) = 0$. Using this information, you have decided to reconstruct the polynomial in coefficient form.

(a) [15 points] Give a polynomial time procedure to reconstruct the original polynomial, in coefficient form, from $x_1, \ldots x_n$. Give an argument of correctness for why your method works as well as for your runtime.

Hint: A polynomial $P(x)$ has a zero at x_j if and only if the polynomial $Q(x) = (x - x_j)$ is a factor of $P(x)$.

Solution: Let $P(x)$ be the polynomial we are trying to reconstruct. Since we know that each polynomial of the form $Q_i(x) = (x - x_i)$ for $i \in [1, n]$ is a factor of $P(x)$, we know that the product polynomial $P'(x) = \prod_{i=1}^{n} Q_i(x)$ is also a factor of $P(x)$. Furthermore, since we know that $P(x)$ is a degree n polynomial with leading coefficient 1, and it has exactly n distinct zeros, then by the *Fundamental Theorem of Algebra* we have that $P(x) = P'(x)$. Therefore, to compute $P(x)$ in coefficient form, all we have to do is compute the product $P(x) = \prod_{i=1}^{n}(x - x_i)$.

Naively, computing this requires n multiplications of polynomials with degree bounded by n. Each multiplication takes $O(n)$ time to multiply a degree 1 polynomial by a degree bound n polynomial. Therefore the total runtime is $O(n^2)$.

(b) [35 points] Show how to use divide and conquer to get an $O(n \log^2 n)$ time algorithm. You may assume arithmetic operations take constant time.

Solution: We use the following divide and conquer algorithm.

1. If $n = 1$, return the degree 1 polynomial itself, otherwise continue.
2. Split the product into two equal sized products $P(x) = \prod_{i=1}^{n/2}(x - x_i) \cdot \prod_{i=n/2+1}^{n}(x - x_i)$.
3. Recursively compute the two half size product polynomials to obtain two polynomials of degree $n/2$ in coefficient form.
4. Use FFT to multiple the two polynomials in $O(n \log n)$ time and return the resulting polynomial in coefficient form.

The total runtime for this algorithm has the recurrence

$$T(n) = 2T(n/2) + O(n \log n)$$

By case 2 of the master theorem, this solves to $O(n \log^2 n)$.

Problem 2-3. 3-Way Fast Fourier Transform [50 points] This week in lecture we saw how to multiply two polynomials using the divide and conquer algorithm called the Fast Fourier Transform. Recall that the FFT evaluates a degree n polynomial $A(x)$ at the n-th roots of unity in $O(n \log n)$ time by dividing the problem into $A_{even}(x)$ and $A_{odd}(x)$, which are of degree $n/2$. In this problem, we want to see if we can do better by dividing it into 3 recursive subproblems. We assume that n is a power of 3, and arithmetic operations take constant time.

(a) [15 points] Show how to divide the polynomial $A(x) = a_0+a_1x+a_2x^2+..+a_{n-1}x^{n-1}$ into 3 polynomials of degree $(n-3)/3$. And give an equation to reconstruct A from these 3 smaller polynomials.

Solution: We split the polynomial in a similar fashion as we did in the 2-Way FFT. Instead of using even and odd powers, we use the powers modulo 3. Therefore, the three polynomials will consist of terms with degree 0 mod 3, 1 mod 3 and 2 mod 3 each.
Let $A(x) = a_0 + a_1x + a_2x^2 + .. + a_{n-1}x^{n-1}$, define the sub-polynomials $A^{(0)}(x)$, $A^{(1)}(x)$ and $A^{(2)}(x)$ as follows:

$$A^{(0)}(x) = a_0 + a_3x + a_6x^2 + .. + a_{n-3}x^{(n-3)/3}$$
$$A^{(1)}(x) = a_1 + a_4x + a_7x^2 + .. + a_{n-2}x^{(n-3)/3}$$
$$A^{(2)}(x) = a_2 + a_5x + a_8x^2 + .. + a_{n-1}x^{(n-3)/3}$$

Using these sub-polynomials, we can reconstruct $A(x)$ as follows,

$$A(x) = A^{(0)}(x^3) + xA^{(1)}(x^3) + x^2(A^{(2)}(x^3))$$

It is trivial to substitute in and see that this does in fact reconstruct $A(x)$.

(b) [30 points] Prove that we can use the complex roots of unity to recursively evaluate this polynomial in $O(n \log n)$ time. Give a recurrence for the algorithm and solve it!
Hint: Show that raising the set of the n-th roots of unity to the power of 3 shrinks the size of the set by a factor of 3.

Solution: In order to multiply the two degree $n-1$ polynomials A and B, we need to evaluate them on at least $2n-2$ points. This is because the degree of the product polynomial $A \cdot B$ will be $2n-2$. Since n is a power of three, let us choose $N = 3n$, which is also a power of 3, and evaluate the polynomial at the N-th complex roots of unity.
We first show that taking the set of the N-th roots of unity to the 3rd power collapses the size by 3. Recall that the $w_N = e^{2\pi i/N}$ is the primary N-th root of unity. Therefore the set of the N-th roots of unity are the successive powers of w_N as follows.

$$\{w_N^0, w_N^1, w_N^2, .., w_N^{N-1}\}$$

Now consider the set where we raise each element to the power of 3.

$$\begin{aligned}
&\{(w_N^0)^3, (w_N^1)^3, (w_N^2)^3, .., (w_N^{N-1})^3\} \\
=&\{w_N^0, w_N^3, w_N^6, .., w_N^{3N-3}\} \\
=&\{e^{\frac{2\pi i}{N}\cdot 0}, e^{\frac{2\pi i}{N}\cdot 3}, e^{\frac{2\pi i}{N}\cdot 6}, .., e^{\frac{2\pi i}{N}\cdot (3N-3)}\} \\
=&\{e^{\frac{2\pi i}{N/3}\cdot 0}, e^{\frac{2\pi i}{N/3}\cdot 1}, e^{\frac{2\pi i}{N/3}\cdot 2}, .., e^{\frac{2\pi i}{N/3}\cdot (N-1)}\}
\end{aligned}$$

Notice that the last set is exactly the $N/3$ complex roots of unity, repeated 3 times. This is because after $e^{\frac{2\pi i}{N/3}\cdot(\frac{N}{3}-1)}$, the numbers repeat cyclically because $e^{\frac{2\pi i\cdot(N/3+j)}{N/3}} = e^{\frac{2\pi i\cdot(N/3)}{N/3}} e^{\frac{2\pi i\cdot j}{N/3}} = e^{\frac{2\pi i\cdot j}{N/3}}$ for $0 \leq j \leq \frac{N}{3} - 1$. Similarly, this holds for $\frac{N}{3} \leq j \leq \frac{2N}{3} - 1$. Hence, we have a collapsing set of complex numbers.

Therefore, the recurrence for the algorithm to evaluate a degree-bound n polynomial on $N = 3n$ -th complex roots of unity is

$$T(n, N) = 3T\left(\frac{n}{3}, \frac{N}{3}\right) + O(n).$$

The $O(n)$ comes from the additions and multiplications in the merge step of the algorithm. Since $N = O(n)$ and it is also cut in 3 with each step, this can be simplified to,

$$T(n) = 3T\left(\frac{n}{3}\right) + O(n).$$

Hence by case 2 of the master theorem, this algorithm also runs in $O(n \log n)$ time.

(c) [5 points] Putting it all together, give the full algorithm for multiplying the two degree $n - 1$ polynomials $A(x)$ and $B(x)$ using our new 3-Way FFT algorithm. (You may assume the we already have the 3-Way Inverse-FFT). Give the runtime analysis for your overall algorithm.

Solution: We use the following algorithm to multiply A(x) and B(x).

1. Use our 3-Way FFT to evaluate A and B on the $N = 3n$-th roots of unity.
2. Perform a point-wise multiplication of each point to obtain $3n$ evaluations for the product polynomial $C = A(x) \cdot B(x)$.
3. Use the 3-Way Inverse FFT to interpolate the points and obtain C in coefficient form.

Step 1 takes $O(n \log n)$ time, step 2 takes $O(n)$ time, and step 3 takes $O(n \log n)$ time as well. Therefore the overall runtime of our algorithm is $O(n \log n)$.

Problem 2-4. Making Change in Polyland[50 points] In Polyland, they have n different denominations in their currency. Each denomination has value in the range $[1, 1000n]$. Alice is a software engineer at a major bank in Polyland. Her first task is to design an algorithm that can solve the following problem. *Given a value k, is it possible to make change for k units of money using **at most 3** notes of the Polyland currency, where a denomination may be used more than once?* Help Alice solve this problem.

For example, given the set of denominations $D = \{1, 3, 4, 6\}$ and $k = 7$, the answer is **YES** because we can get 7 as $(3 + 4)$ or $(1 + 6)$ or even $(1 + 3 + 3)$.

(a) [15 points] Give an algorithm to solve this problem in $O(n^2)$ time.

Solution: Our algorithm for this problem proceeds in 3 phases. In each phase j we check whether it is possible to make change for k units of money using exactly j notes of the currency.

1. In the first phase, we simply iterate through the set of denominations, D and return **YES** if $k \in D$.
2. In the second phase, we iterate though D and for each denomination d, we return **YES** if $k - d \in D$.
3. In the last phase, we construct a list L of length $O(n^2)$ which contains the sum of every pair of denominations in D. We also construct a bit array A of length $1000n$ containing a 1 at index i if $i \in D$ and 0 otherwise. We then iterate through each element s in L and return **YES** if $k - s \geq 0$ and $A[k - s] = 1$.

If all three phases do not return, simply return **NO**. Step 1 takes $O(n)$ time, steps 2 and 3 take $O(n^2)$ time, therefore the total runtime for this algorithm is $O(n^2)$.

(b) [35 points] Give an algorithm which uses the Fast Fourier Transform to solve this problem in $O(n \log n)$ time.
Hint: Can you use the set of denominations to construct a useful polynomial?

Solution: We solve this problem by constructing a polynomial $P(x)$ of degree $1000n$ as follows:

$$P(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{1000n} x^{1000n}$$

Where $a_0 = 1$, and each a_i for $i > 0$ is assigned according to,

$$a_i = \begin{cases} 1 & \text{if } i \in D \\ 0 & \text{Otherwise.} \end{cases}$$

We then use the FFT to compute $P^3(x) = P(x) \cdot P(x) \cdot P(x)$. Let b_k be the coefficient of x^k in $P^3(x)$. We return **YES** if $b_k > 0$ else **NO**.

Runtime: In the first step, it takes $O(n)$ time to construct $P(x)$. Since $P(x)$ has degree $O(n)$, it then takes $O(n \log n)$ time to compute $P^3(x)$ using two calls to FFT (notice that $P^2(x)$ will also have degree $O(n)$). Then finally, depending on the representation of the polynomial, it takes $O(n)$ time to read b_k. Therefore the total runtime is $O(n \log n)$.

Correctness: To see why this works, consider first what happens when we compute

$P^2(x) = P(x) \cdot P(x)$. Let c_j be the the coefficient of x^j in $P^2(x)$. Recall that c_j is given by the convolution formula as follows:

$$c_j = \sum_{i=0}^{j} a_i a_{j-i}$$

From here we conclude that $c_j \neq 0$ if and only if there exist both denominations $i, (j-i) \in D$ implying that both $a_i, a_{j-i} \neq 0$, notice that since we fix $a_0 = 1$, even though $0 \notin D$, it is useful to allow for the case where $j \in D$ itself. All other coefficients in $P^2(x)$ will be 0. Similarly, consider what happens when we multiply $P^2(x)$ by $P(x)$ to get $P^3(x) = P^2(x) * P(x)$. Let b_j be the coefficient of x^j in $P^3(x)$. We have that,

$$b_j = \sum_{i=0}^{j} a_i c_{j-i}$$

Therefore we see that $b_j \neq 0$ if and only if there exists a $a_i, c_{j-i} \neq 0$. Therefore there needs to exist an element $i \in D$ and, $j - i$ can be made using at most 2 elements from D. Particularly, $b_k \neq 0$ if and only if there exits a set of at most 3 denominations that sum to k.