Problem Set 2 Solutions

This problem set is due **at 11:59pm** on **Wednesday, February 22, 2017.**

EXERCISES (NOT TO BE TURNED IN)

We suggest taking a look at these exercises *before* you complete the PSET.

- Do exercise 30.1-1.
- Do exercise 30.2-1.

We suggest you take a look at the following exercise *after* you complete this PSET.

How would you use FFT to multiply polynomials on two variables? Let n be a power of 2. Assume that your input is given as two n by n matrices $P = (p_{ab})_{a,b=0}^{n-1}$ and $Q = (q_{ab})_{a,b=0}^{n-1}$ of coefficients of polynomials $p(x, y)$ and $q(x, y)$ respectively, so that

$$p(x, y) = \sum_{a=0}^{n-1} \sum_{b=0}^{n-1} p_{ab} x^a y^b$$

$$q(x, y) = \sum_{a=0}^{n-1} \sum_{b=0}^{n-1} q_{ab} x^a y^b$$

The size of the input is therefore $2n^2$ and the output will be a $2n$ by $2n$ matrix of coefficients of $p(x, y)q(x, y)$.

1. Show that a bivariate polynomial $p(x, y)$ whose degree is at most $n - 1$ in each variable is uniquely determined by its values on the set $S_n = \{(\omega^i, \omega^j)\}_{i,j=0}^{n-1}$, where ω is the n-th complex root of unity.

   Solution: We will use a similar formula to the univariate case that we analyzed in class. Let us define the matrix $(M)_{ab} = (\omega^{ab})$ and the matrix $V = (p(\omega^i, \omega^j))_{i,j=0}^{n-1}$.

   Now, we will prove that the following formula holds:

   $$MPM = V$$

   $$MPM = M(PM) = M(\sum_{b=0}^{n-1} p_{ab}\omega^{bj})_{a,j=0}^{n-1} = (\sum_{a=0}^{n-1}(\sum_{b=0}^{n-1} p_{ab}\omega^{bj})\omega^{ia})_{i,j=0}^{n-1}$$
   $$= (\sum_{a=0}^{n-1}\sum_{b=0}^{n-1} p_{ab}(\omega^i)^a(\omega^j)^b)_{i,j=0}^{n-1} = V$$

   So, since M is invertible (see lecture notes), P is uniquely defined by:

   $$P = M^{-1}VM^{-1}$$

An alternative way to show this is by defining an n^2-vector p' with p_{ab} as the $an+b$-th element and an n^2-vector v' with $p(\omega^a, \omega^b)$ as $an+b$-th element. Then, there exists a $n^2 \times n^2$ matrix M' such that:

$$M'p' = v'$$

and it is enough to prove that M' is invertible.

Observe that $M' = M \otimes M$, where $\otimes$ denotes the tensor product. Then, by the properties of tensor product M' is invertible.

2. Give an $O(n^2 \log n)$ algorithm for evaluating $p(x, y)$ on the points in S_n and an $O(n^2 \log n)$ algorithm for recovering the coefficients of $p(x, y)$ from its values on the points in S_n.

Solution: Evaluate $p(x, y)$ on the points in S_n: As in the FFT, we will use divide and conquer. The polynomial $p(x, y)$ can be written as:

$$p(x,y) = p_{even,1}(x^2, y^2) + xyp_{even,2}(x^2, y^2) + x \cdot p_{odd,1}(x^2, y^2) + y \cdot p_{odd,2}(x^2, y^2)$$

where

$$p_{even,1}(x,y) = \sum_{k=0}^{(n-2)/2} \sum_{l=0}^{(n-2)/2} p_{2k,2l} x^{2k} y^{2l}$$

$$p_{even,2}(x,y) = \sum_{k=0}^{(n-2)/2} \sum_{l=0}^{(n-2)/2} p_{2k+1,2l+1} x^{2k} y^{2l}$$

$$p_{odd,1}(x,y) = \sum_{k=0}^{(n-2)/2} \sum_{l=0}^{(n-2)/2} p_{2k+1,2l} x^{2k} y^{2l}$$

$$p_{odd,2}(x,y) = \sum_{k=0}^{(n-2)/2} \sum_{l=0}^{(n-2)/2} p_{2k,2l+1} x^{2k} y^{2l}$$

which are 4 degree $n/2$ bivariate polynomials. We need to evaluate them on the set $\{((\omega^i)^2, (\omega^j)^2)\}_{i,j=0}^{n-1}$, which is equal to $S_{\frac{n}{2}}$ since n is even.

So, $T(n) = 4T(n/2) + O(n^2) \Rightarrow T(n) = \Theta(n^2 \log n)$ from Case 2 of Master Theorem.

An alternative solution:

From part 1, we have that :

$$V = MPM$$

From the lecture, we know how to compute the product of an n-vector with M in $O(n \log n)$. So, we can compute $R = PM$ in time $O(n^2 \log n)$. Also, since M is symmetric, we can use the same algorithm in order to compute MR in $O(n^2 \log n)$.

So, we can compute V in $O(n^2 \log n)$.

Recover coefficients of $p(x, y)$ from its values on the points in S_n: From part 1, $P = M^{-1}VM^{-1}$. Also, from the lecture we know that we can multiply a n-vector with M^{-1} in time $O(n \log n)$. So, we can multiply a $n \times n$ matrix with M^{-1} in time $O(n^2 \log n)$.

So, the total time for recovering the coefficients of $p(x, y)q(x, y)$, using the divide and conquer technique from the lecture, is $O(n^2 \log n)$.

3. Use the algorithms in part 2 to design an $O(n^2 \log n)$ algorithm for computing the coefficients of $p(x, y)q(x, y)$.

   Solution: Use the ideas from FFT:

   - Evaluate $p(x, y)$ and $q(x, y)$ on the points in S_{2n} using the $O(n^2 \log n)$ algorithm of part 2.
   - Find the values of $p(x, y)q(x, y)$ on the points in S_{2n} in another $O(n^2)$ operations.
   - Interpolate to get the coefficients of $p(x, y)q(x, y)$ using the $O(n^2 \log n)$ algorithm of part 2.

   In total, this needs $T(n) = O(n^2 \log n) + O(n^2 \log n) + O(n^2) = O(n^2 \log n)$ time.

Problem 2-1. 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: Approaching the Solution: We can approach this problem by studying how we split the polynomial in a similar fashion in the 2-Way FFT. This time, 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.

Solution: 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.

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

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 $0 \le j \le \frac{N}{3} - 1$. Similarly, this holds for $\frac{N}{3} \le j \le \frac{2N}{3} - 1$. 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 that 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 2-2. [50 points]

Let S be a set of n integers.

(a) [10 points] Given an $O(n \log n)$ time algorithm that determines whether S contains *two* elements that sum to zero.

Solution:

Approaching the Solution: Since we are asked to give a solution that runs in $O(n \log n)$ time, this suggests that we cannot directly iterate over each distinct pair of elements in S, which would take $O(n^2)$ time. Next, observe that as soon as we select a number $s \in S$, we know that if $-s \in S$ and $s \neq -s$ then $s + -s = 0$ and s and $-s$ are distinct. Thus we can approach the problem as follows: for each element in S, check whether the negation of that number is also in S. This involves $O(n)$ lookups of whether a number is in S. Since we want an algorithm runs in $O(n \log n)$ time, this means that each lookup should run in $O(\log n)$ time. This can easily be accomplished by storing S in a list, sorting it and then using binary search to check for presence in the S.

Solution: Here is an example of such an algorithm:

1. Store the elements of S in an array X and sort X.
2. For each element of $s \in X$, check if $-s$ is a present in X via a binary search. If $-s \in X$ (implying $-s \in S$) and $s \neq -s$ (implying that the two elements of X are distinct) then return *TRUE*.
3. Return *FALSE*.

Correctness: The algorithm determines whether any pair of distinct elements in X sums to 0 and returns true if and only if this is the case.

Runtime Analysis:

1. Step (1) takes at most $O(n \log n)$ time to populate and sort X.
2. Step (2) involves a up to $O(n)$ executions of a binary search operation (on X), each of which runs in $O(\log n)$ time. Thus this step run in $O(n \log n)$ time.
3. Step (3) takes at most $O(1)$ time.

Thus the algorithm has a runtime of $O(n \log n)$.

(b) [15 points] Give an $O(n^2)$ time algorithm that determines whether S contains *three* elements that sum to zero.

Clarification: You may assume that hashing takes $O(1)$ time.

Solution: **Approaching the Solution:** Consider how this problem is different from part (a): now the desired runtime is $O(n^2)$, so we could loop over all distinct pairs of elements in S, but we are now looking for *three* numbers that sum to 0. Again we observe that if we have already selected two distinct elements from S, we can check whether the negation of their sum is also in S. To maintain the desired runtime, since we are checking $O(n^2)$ distinct pairs drawn from S, for each distinct pair we must perform a lookup in S in $O(1)$ time. Thus, we store S in a hash table so that we can check for membership in $O(1)$ time.

Solution: Here is an example of such an algorithm:

1. Initialize a hash table H that is populated with the elements of S.
2. For each pair, (a, b), of two distinct elements in S, check if H contains $-(a+b)$. If H contains $-(a+b)$ and the three numbers $-(a+b)$, a and b are distinct, then return *TRUE*.
3. Return *FALSE*.

Correctness: The algorithm determines whether there exists a pair of distinct elements in S such that there exists a third element in S that is distinct from the two and the three of them sum to 0.

Runtime Analysis: Step 1 takes on average $O(n)$ time (n insertions into H, each of which take $O(1)$ time on average). Step 2 consists of a loop with $O(n^2)$ iterations, each of which takes on average $O(1)$ time on average. Thus Step 2 runs in $O(n^2)$ time. Thus the algorithm has an average runtime of $O(n^2)$.

(c) [25 points] Now suppose that S only contains integers between 1 and 2017n. Give an $O(n \log n)$ time algorithm to determine whether S contains *three* elements that sum to n.
Correction: An integer may be used more than once. I.e. if $n = 4$ and $S = \{1, 2, 3, 4\}$ then one may use $(1, 1, 2)$ as three numbers that sum to n.

Solution:
Approaching the solution: First, note that the desired runtime of $O(n \log n)$ informs us that we can't use a strategy that involves directly iterating over the pairs of numbers in S as we did in part (b), which would take $O(n^2)$ time. Second, note that we are trying to find *any* way in which three elements (drawn possibly with replacement from S) can sum to n - this suggests that we may want try and determine whether we can sum to n by computing a convolution of the elements of S. The final piece of the puzzle then comes in when we note that S is drawn from a bounded range of integers, so it is possible to represent S as in terms of the coefficients fo a polynomial.

Solution: The following algorithm runs in $O(n \log n)$:

1. Construct the degree $2017n$ polynomial $A(x) = \sum_{i=0}^{2017n} a_i x^i$, where a_i is 1 if $i \in S$ and 0 otherwise.
2. Compute the polynomial $A^3(x)$ by using the FFT to multiply the polynomials.

3. Return *TRUE* if the degree n term in the polynomial $A^3(x)$ is non-zero; otherwise return *FALSE*.

Correctness: The key observation to make is that the coefficient of the degree k term in the polynomial $A^3(x)$ will be non-zero if and only if there are at least three non-zero coefficients, a_x, a_y, a_z, of $A(x)$ such that $x+y+z=n$. This can be shown as follows:

$$A^3(x) = \left(\sum_{i=0}^{2017n} a_i x^i\right)^3 = \sum_{j \in \{0..2017n\}} \left(\left(\sum_{\substack{i_1,i_2,i_3 \in \{0..2017n\} \\ i_1+i_2+i_3=j}} a_{i_1} a_{i_2} a_{i_3}\right) x^j\right)$$

and thus the coefficient of the degree n term in $A^3(x)$ is:

$$\sum_{\substack{i_1,i_2,i_3 \in \{0..2017n\} \\ i_1+i_2+i_3=n}} a_{i_1} a_{i_2} a_{i_3}$$

and this sum is non-zero if and only if there exists three values that may be drawn from S that sum to n.

Runtime Analysis:

1. In this step, the degree $2017n$ polynomial $A(x)$ may be constructed in $O(n)$ time.
2. This step consists of using the FFT to multiply the three polynomials, each of degree $O(n)$, in $O(n \log n)$ time.
3. This step runs in $O(1)$ time.

Thus the algorithm has a runtime of $O(n \log n)$.