CS 38 Algorithms	**April 28, 2023**

Problem Set 3

Codename: Jambul	**Late Submission:** No

1 Problem 1

Algorithm 1. (Chirp Z-transform.) We will use the Fast Fourier Transform for the convolution of two polynomials to solve the problem.

1. Create two empty vectors a, b with $|a| = n$, $|b| = 2n-1$, and set $a_i = x_i z^{i^2/2}, b_i = b_{2n-i-1} = z^{-i^2/2}$.
2. Set $a_i = 0$ for $n < i \leq 2n-2$, so $|a| = 2n-1$.
3. Compute the FFT of a and b and multiply the resultant sequences pointwise. Denote the output of these operations by c.
4. Compute the inverse FFT of c, which we denote by $(a * b)$.
5. Return $z^{j^2/2} \cdot (a * b)$.

Proof of correctness. As per the hint, we have that

$$ij - \frac{j^2}{2} = \frac{i^2}{2} - \frac{(j-i)^2}{2} \implies y_j = \sum_{i=0}^{n-1} x_i z^{ij} = z^{j^2/2} \sum_{i=0}^{n-1} x_i z^{i^2/2-(j-i)^2/2}.$$

Now define $a = (a_i)_i, b = (b_i)_i$ to be two new sequences defined by

$$a_i = x_i z^{i^2/2}, b_i = z^{-i^2/2}.$$

Then we can write

$$y_j = z^{j^2/2} \sum_{i=0}^{n-1} a_i b_{j-i}.$$

We now define two new vectors a^*, b^*, with $a_i^* = a_i$ for $0 \leq i \leq n-1$, and $a_i^* = 0$ for $n-1 < i \leq 2n-2$, and with $b_i^* = b_{i+n-1}$. Thus we simply append zeroes to a until its length is $2n-1$ to get a^*, and we shift the index on b by $n-1$ to get b^*.[1] Then $|a| = |b| = 2n-1$, and we have that

$$y_j = z^{j^2/2} \cdot (a * b)_j.$$

We know from lecture that the convolution $(a * b)$ can be obtained by computing the FFT of a and b, multiplying the transforms pointwise, and taking the inverse FFT of the resultant sequence. Thus the algorithm indeed returns the desired chirp z-transform $(y_0, ..., y_{n-1})$ of $(x_0, ..., x_{n-1})$.

Complexity. We will now confirm that the overall runtime of this algorithm is $\mathcal{O}(n \lg(n))$. We must first compute the sequences a, b. Computing $z^{i^2/2}$ will take at most $\mathcal{O}(\lg(n))$ time (and multiplication of x_i by the resultant complex number is assumed to take $\mathcal{O}(1)$ time), so computing both a and b takes $\mathcal{O}(n \lg(n))$. Padding a then takes $\mathcal{O}(n)$ time. Finally, computing the FFT of both a and b takes $\mathcal{O}(n \lg(n))$ time; multiplying the resultant transforms pointwise takes $\mathcal{O}(n)$ time; and finding the inverse FFT of the multiplied sequence takes $\mathcal{O}(n \lg(n))$.

Thus the algorithm indeed has $\mathcal{O}(n \lg(n))$ time complexity.

[1] Note that in Algorithm 1, the vectors we define in step 1 and 2 are simply a^* and b^*.

2 Problem 2

We will use a combination of a divide-and-conquer approach alongside the FFT for polynomial multiplication to solve the problem.

Algorithm 2. (Polynomial Multiplication.) Let our input array of constants be $c = (c_1, ..., c_n)$. Then we proceed as follows.

1. If $|c| = 1$, return $(c_1, 1)$.
2. Set $m = \lfloor n/2 \rfloor$.
3. Set l to be the list of coefficients returned from calling "Polynomial Multiplication" on $(c_1, ..., c_m)$.
4. Set r to be the list of coefficients returned from calling "Polynomial Multiplication" on $(c_{m+1}, ..., c_n)$.
5. Compute the FFTs of l and r and multiply the resultant sequences pointwise.
6. Compute the inverse FFT of the sequence of pointwise products and denote the output o.
7. Return the coefficients on the terms of o in order of decreasing degree.

Proof of correctness. Denote the list of input constants by c, so in any call to Algorithm 2, we are given some partition $c^* = (c_i, ..., c_j), 1 \leq i \leq n, 1 \leq j \leq n$, of c. We will prove the correctness of the algorithm by induction on the length $|c^*|$ of c^*.

For our base case, we consider $|c^*| = 1$, that is, the event in which our input polynomial is of the form $(x - c_i)$ for some real constant c_i. Trivially, then, the coefficients of this polynomial are given by $(c_i, 1)$. Thus the claim holds for the base case.

Now assume the claim holds for some $1 < |c^*| = k < n$. Then assume we are given some list of constants of length $k + 1$. We will notate this new list again by c^*.[2] By assumption, calling the algorithm on the left half $(c^*_1, ..., c^*_{\lfloor (k+1)/2 \rfloor})$ and the right half $(c^*_{\lfloor (k+1)/2 \rfloor + 1}, ..., c^*_{k+1})$ of c^* will yield the coefficients representing the polynomials formed by $(x - c^*_1)...(x - c^*_{\lfloor (k+1)/2 \rfloor})$ and $(x - c^*_{\lfloor (k+1)/2 \rfloor})...(x - c^*_{k+1})$, respectively.

Then, we know from lecture that the product of two polynomials l and r is given by

$$o(x) = \{l * r\}(x) = \mathcal{F}^{-1}\{\mathcal{F}(l) \cdot \mathcal{F}(r)\}, \tag{1}$$

where $\mathcal{F}\{\cdot\}$ is the Fourier transform operator. It follows that computing and returning the coefficients of (1) will yield the list of coefficients of the polynomial

$$(x - c^*_1)...(x - c^*_{\lfloor n/2 \rfloor}) \cdot (x - c^*_{\lfloor n/2 \rfloor})...(x - c^*_n),$$

which is of course c^*. Thus by induction on $|c^*|$, the claim holds for all $1 \leq |c^*| \leq n$, and in particular, it holds for $c^* = c$.

Complexity. As mentioned, the algorithm combines a divide-and-conquer approach with the FFT algorithm for polynomial multiplication to solve the problem. Splitting the polynomial takes $\mathcal{O}(1)$ time. We know we can leverage the FFT to compute the coefficients of the product of two polynomials in $\mathcal{O}(n \log(n))$ time. We then must do this $\mathcal{O}(\log(n))$ times, as we will be "merging" (multiplying polynomials) $\mathcal{O}(\log(n))$ times, since we split the array of constants in half at each call. Thus the total running time of the algorithm is indeed $\mathcal{O}(n \log^2(n))$.

[2] Apologies for the notation abuse.

3 Problem 3

We will use a variation of the mergesort algorithm to solve this problem.

In particular, we note that we seek to populate the output entries $c[i]$ with the number of elements in a that begin on the right of $a[i]$ but ultimately rest on the left side of it in the sorted array. We will first outline a subroutine for the modified mergesort algorithm that will be the driving component of this solution.

Algorithm 3. (Modified Mergesort.) This algorithm requires some input array $m = (m_1, ..., m_n)$, where each entry m_i is a 2-tuple with the first entry equal to the index i of the entry (i.e., $m_i[0] = i$). We also assume there exists an array p that has been created outside the algorithm but can be modified by the algorithm.

1. If $|m| = n = 1$: return m.
2. Let m_l be the left half (rounded down) of the input array m; i.e., $m_{(l)} = (m_1, ..., m_{|m|//2})$. Let $m_{(r)}$ be the remaining right half of m.
3. Recursively call "Modified Mergesort" on $m_{(l)}$; denote the output s_l for the sorted left half of m.
4. Recursively call "Modified Mergesort" on $m_{(r)}$; denote the output s_r for the sorted right half of m.
5. Create an empty, temporary array o. Set $l = r = 0$.
6. While $l < |s_l|$ and $r < |s_r|$:
   (a) If $s_l[l][1] < s_r[r][1]$:
      i. Set the entry at the index $s_l[l][0]$ in the output array p to r.
      ii. Append $s_l[l]$ to the temporary array o.
      iii. Add 1 to l.
   (b) Else:
      i. Append $s_r[r]$ to the temporary array o.
      ii. Add 1 to r.
7. While $l < |s_l|$:
   (a) Set the entry at the index $s_l[l][0]$ in the output array p to r.
   (b) Append $s_l[l]$ to the temporary array o.
   (c) Add 1 to l.
8. While $r < |s_r|$:
   (a) Append $s_r[r]$ to the temporary array o.
   (b) Add 1 to r.
9. Return the temporary array o.

Algorithm 4. (Smaller Elements Count.) We are now ready to outline the full solution to the problem. In line with the problem, we denote the input array a.

1. Create an array t such that $t[i] = (i, a[i])$. Initialize the output array $p = [0, ..., 0]$ with length $|a| = n$.
2. Call "Modified Mergesort" on t. (We assume in "Modified Mergesort" that the subroutine has permission to modify p.)

3. Return the output array p.

Proof of correctness. Denote the input array $a = (a_1, ..., a_n)$. We will show that any call to Algorithm 3 with an input $m = (m_1, ..., m_n), m_i = (i, a_i), a_i \in \mathbb{R}$ will return m sorted by the values stored in the second index of each 2-tuple m_i and will populate an empty array $p = (p_1, ..., p_n)$ such that $p_i = |\{a_j : a_j < a_i, j > i\}|$. We will prove the correctness of the algorithm by induction on the length n of the input array a.

For our base case, consider input arrays of length $n = 1$. Trivially, we will have that $m = ((0, a_1))$ is in fact sorted by the second entry in each tuple in m, as there is only one such entry in the first place. Then we expect Algorithm 3 to return m and Algorithm 4 to return $p = (0)$, which is indeed the case, since Algorithm 4 initializes $p = (0, ..., 0)$ with length $|p| = n = 1$.

Now suppose the claim holds for some $n \geq 1$. Then suppose we input a list $m = ((0, a_1), ..., (0, a_{n+1}))$ to Algorithm 3 through Algorithm 4. Then we know that when we call Algorithm 3 recursively on l the left half of m and r the right half of m, we will be able to retrieve l and r sorted by their entries in the second slot of each 2-tuple in the arrays. Furthermore, also following from the induction hypothesis, we know $p = (p_1, ..., p_n)$ will be populated such that each entry p_i on the left half of p will hold the number of elements a_j in the left half of a that are of index greater than a_i and are also less than a_i. The same will hold for the right half of p.

Then to combine and sort the arrays l and r, we can simply use the mergesort algorithm described in lecture. To populate p, however, we note we must keep track of the number n_r of elements that have been added to the auxiliary output array. Each time we append an element l_i from l to the auxiliary array, we know that l_i must be greater than every element that is now to the left of it. However, we cannot simply add the number of these elements to p_i: We must include only those elements that are smaller *and* from r. Thus we add n_r as defined previously.

By assumption, this will yield p such that $p_i = |\{a_j : a_j < a_i, j > i\}|$, since each time we add l_i to r_i, we know that p_i has already been updated to include the number of elements in l that were originally to the right of l_i and are smaller than l_i. Thus adding that same count against the new set of numbers in r completes the algorithm, thus confirming its correctness.

Complexity. Similar to the classic mergesort algorithm, this solution has two parts: Splitting and merging. Splitting is done in constant time. Merging, however, takes $\mathcal{O}(n)$ time, as at most we will encounter n comparisons in each merge. Furthermore, at each step, writing to the output array p in the "Modified Mergesort" subroutine takes constant time. Since the array is split into two at each step, there will be $\mathcal{O}(\lg(n))$ levels in the mergesort tree, resulting in the usual $\mathcal{O}(n \lg(n))$ runtime found in mergesort.