Problem Set 4

Due dates: Electronic submission of the pdf file of this homework is due on **2/17/2023 before 11.59pm** on canvas.

Name: Huy Quang Lai

Resources. Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford. *Introduction to Algorithms*. The MIT Press.

On my honor, as an Aggie, I have neither given nor received any unauthorized aid on any portion of the academic work included in this assignment. Furthermore, I have disclosed all resources (people, books, web sites, etc.) that have been used to prepare this homework. The solutions given in this homework are my own work.

Signature: _____

Make sure that you describe all solutions in your own words. Typeset your solutions in LaTeX. Read chapters 30 and 15 in our textbook.

Problem 1. (20 points) Let ω be a primitive nth root of unity. The fast Fourier transform implements the multiplication with the matrix

$$F = (\omega^{ij})_{i,j\in[0..n-1]}.$$

Show that the inverse of the matrix F is given by

$$F^{-1} = \frac{1}{n}(\omega^{-jk})_{j,k\in[0..n-1]}$$

[Hint: $x^n - 1 = (x-1)(x^{n-1} + \cdots + x + 1)$, so any power $\omega^\ell \neq 1$ must be a root of $x^{n-1} + \cdots + x + 1$.] Thus, the inverse FFT, called IFFT, is nothing but the FFT using ω^{-1} instead of ω, and multiplying the result with $1/n$.

Solution. Since F is a matrix, by definition, $F \cdot F^{-1} = I$ where I is the identity matrix.
Representing F as a summation, leads to

$$F = \sum_{j=0}^{n-1} \omega^{ij} = \begin{cases} n & \text{if } i \mod n = 0 \\ 0 & \text{if } i \mod n \neq 0 \end{cases}$$

This summation is valid because if i is a multiple of n, $\omega^i = 1$. Since the summation has a lower-bound of 0 and a higher bound of $n-1$, there will be n ones added together resulting in n.
$F = 0$ when i is not a multiple of n using the hint given.
The summation would result in $\omega^{0i} + \omega^{1i} + \cdots + \omega^{(n-1)i}$. From the hint, when this summation is multiplied by $\omega^i - 1$ the result will be $\omega^n - 1$.
Since $\omega^{in} - 1 = (\omega^i - 1)\left(\omega^{0i} + \omega^{1i} + \cdots + \omega^{(n-1)i}\right)$, all ω^i's are roots of this equation since any ω^i plugged into the left-hand side would result in the nth root raised to the nth power. As a result, the left-hand side is $1 - 1 = 0$.
From this result, either $\omega^i - 1$ or $\omega^{0i} + \omega^{1i} + \cdots + \omega^{(n-1)i}$ must be equal to zero.
Since $\omega^i \neq 1$, then $\omega^{0i} + \omega^{1i} + \cdots + \omega^{(n-1)i}$ must be equal to zero.
Next F^{-1} needs a summation representation. By definition, $F \cdot F^{-1} = I$. Using the summation definition, $F \cdot F^{-1} = 1$.
In order to get this result, the imaginary numbers can be multiplied by their complex conjugates to result in 1. This complex conjugate would result in the following summation,

$$F^{-1} = \sum_{j=0}^{n-1} \omega^{-jk}$$

Multiplying F and F^{-1}, the product should be 1.

$$F \cdot F^{-1} = \sum_{i=0}^{n-1} \omega^{(i-k)j}$$

This summation is very similar to the representation of F with i substituted for $i-k$. As a result, $F \cdot F^{-1}$ will equal n if $i-k$ is a multiple of n. However we need $F \cdot F^{-1}$ to equal zero in this condition, therefore the entire summation needs to be multiplied by $\frac{1}{n}$.
Finally, with these results,

$$F^{-1} = \frac{1}{n}\left(\omega^{-jk}\right)_{j,k\in[0..n-1]}$$

Problem 2. (20 points) Describe in your own words how to do a polynomial multiplication using the FFT and IFFT for polynomials $A(x)$ and $B(x)$ of degree $\leq n-1$. Make sure that you describe the length of the FFT and IFFT needed for this task. Be concise and precise.

Solution. Since $A(x), B(x) \in P_{n-1}$,
$A(x) = a_0+a_1x+a_2x^2+\cdots+a_{n-1}x^{n-1}, B(x) = b_0+b_1x+b_2x^2+\cdots+b_{n-1}x^{n-1}$.
Representing the polynomials as vectors results in,

$$A = \langle a_0, a_1, a_2, \cdots a_{n-1}\rangle, B = \langle b_0, b_1, b_2, \cdots b_{n-1}\rangle$$

Setting up Fast Fourier Transform, A^* and B^* need to be split into their respective odd and even degrees.
Let $A_{even} = \langle a_0, a_2, \cdots, a_{n-2}\rangle, A_{odd} = \langle a_1, a_3, \cdots, a_{n-1}\rangle$ and a similar definition for B_{even} and B_{odd}.
With these polynomials, $A(x) = A_{even}(x^2) + xA_{odd}(x^2)$.
Since the even and odd polynomials only have half of the degree of the original polynomial, x^2 is needed to compensate.
By doing this division, the number of operations is reduced to $\frac{n}{2}$ every recursive call.
Converting A into A^*, where A^* is the vector of point values for the polynomial $A(x)$. A similar process is done for $B(x)$.
Then, to multiply the two polynomials, multiply each corresponding point value together. This would result in some vector $C*$ that holds the point values of the product polynomial. After C^* is calculated, a inverse conversion is needed. To do this, an IFFT would need to be applied. The IFFT would have the same things as the FFT except ω^i is replaced with its complex conjugate.
Then, the resulting vector would need to be divided by n and result in C, the coefficient vector representation of the product polynomial, $C(x)$.
Each conversion would take $O(n \log n)$ time, and each root of unity computation would be $O(1)$.

Problem 3. (20 points) How can you modify the polynomial multiplication algorithm based on FFT and IFFT to do multiplication of long integers in base 10? Make sure that you take care of carries in a proper way. Write your algorithm in pseudocode and give a brief explanation.

Solution. The polynomial multiplication can be modified to multiply long integers in base ten by representing each digit of the integers as coefficients of the polynomials.
For example, the polynomial for 123456 would be $a_0 = 6, a_1 = 5, a_2 = 4, a_3 = 3, a_4 = 2, a_5 = 1$ This process is repeated for the second number.
After the conversion, the FFT and IFFT from problem 2 is applied to the polynomials.
Once the polynomial product is calculated, multiply each coefficient by 10^i where i is the coefficient number. Then add all the coefficients after being multiplied together to get the product.

```
def mult(A, B) {
    // Convert A and B into
    A_poly = to_polynomial(A)
    B_poly = to_polynomial(B)

    // Apply FFT on A and B
    A' = FFT(A_poly)
    B' = FFT(B_poly)

    // Multiplly A' and B'
    C' = pointwise_mult(A', B')

    // Apply IFFT
    C = IFFT(C')

    // Calculate product
    sum = 0
    for (int i = 0; i < C.size(); ++i)
        sum += C[i] * 10^i
    return sum
}
```

Problem 4 (20 points)**.** Solve Exercise 15.3-4 on page 389 of our textbook. As stated, in dynamic programming we first solve the subproblems and then choose which of them to use in an optimal solution to the problem. Professor Capulet claims that we do not always need to solve all the subproblems in order to find an optimal solution. She suggests that we can find an optimal solution to the matrix-chain multiplication problem by always choosing the matrix A_k at which to split the subproduct $A_i A_{i+1} \cdots A_j$ (by selecting k to minimize the quantity $p_{i-1} p_k p_j$) before solving the subproblems. Find an instance of the matrix-chain multiplication problem for which this greedy approach yields a suboptimal solution.

Solution. A matrix-chain multiplication problem that yields a sub-optimal solution is the three matrices that follows.
A_0, a 5×6 matrix; A_1, a 6×3 matrix; and A_2, a 3×2 matrix.
Using the greedy approach, the multiplication would be $(A_0 A_1) A_2 = 120$ multiplications. However the optimal solution is $A_0(A_1 A_2) = 96$ multiplications.

Problem 5 (20 points)**.** Solve Exercise 15.4-1 on page 396 of our textbook. Determine an LCS of $\langle 1, 0, 0, 1, 0, 1, 0, 1\rangle$ and $\langle 0, 1, 0, 1, 1, 0, 1, 1, 0\rangle$. Make sure that you explain your answer step-by-step, in detail, rather than just giving an LCS.

Solution. Let $X = \langle 1, 0, 0, 1, 0, 1, 0, 1\rangle$
Let $Y = \langle 0, 1, 0, 1, 1, 0, 1, 1, 0\rangle$
Using the following algorithm, the length of the LCS between the two sequences can be determined.

1. Initialize a table T with 0's of size $(m + 1) \times (n + 1)$, where m and n are the lengths of the sequences X and Y, respectively.
2. Iterate through the sequences X and Y, and for each pair of elements (x, y) do the following:
   (a) If x equals y, set $T[i][j] = T[i - 1][j - 1] + 1$
   (b) If x does not equal y, set $T[i][j] = \max(T[i - 1][j], T[i][j - 1])$
3. The length of the LCS is given by $T[m][n]$.

Filling out the table using this algorithm results in

		X	1	0	0	1	0	1	0	1
		-1	0	1	2	3	4	5	6	7
Y	-1	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	1	1	1
1	1	0	1	1	1	2	2	2	2	2
0	2	0	1	2	2	2	3	3	3	3
1	3	0	1	2	2	3	3	4	4	4
1	4	0	1	2	2	3	3	4	4	5
0	5	0	1	2	3	3	4	4	5	6
1	6	0	1	2	3	4	4	5	5	6
1	7	0	1	2	3	4	4	5	5	6
0	8	0	1	2	3	4	5	5	6	6

To find the actual LCS itself, start in the bottom right corner and traverse backwards through the table. Following the path of the larger values until you reach the top or left edge of the table. The elements in the sequence that correspond to the table cells you visit during this backtracking process will form the LCS.
The traversal algorithm is as follows

1. If the cell value directly to the left and directly above are the same as the current cell, go up and to the left by one.
2. If this condition fails, go directly up.

$\langle 0, 1, 0, 1, 0, 1\rangle$.