Problem 1: [DPV] Problem 2.9 part (b)
Do polynomial multiplication by FFT for the pair of polynomials $1 + x + 2x^2$ and $2 + 3x$. (You might try part (a) for practice.)

Solution.

Let $A(x) = 1+x+2x^2$, $B(x) = 2+3x$ and $C(x) = A(x)B(x)$. Since the highest power of x in $C(x)$ is 3, we can choose $n = 4$. The coefficients of $A(x)$ can be written in an array of size n as

$$a = [1, 1, 2, 0].$$

Similarly, the coefficients of $B(x)$ are captured in

$$b = [2, 3, 0, 0].$$

We also want a n-th root of unity. Here $n = 4$, and we let

$$w = \exp(2\pi i/4) = i.$$

The first thing we want to compute is $\mathrm{FFT}(a, w)$. Recursive calls on the sets of even and odd coefficients of $A(x)$ give

$$\begin{bmatrix} A_{\text{even}}(1) \\ A_{\text{even}}(-1) \end{bmatrix} = \mathrm{FFT}(a_{\text{even}}, -1) = \mathrm{FFT}\left(\begin{bmatrix}1\\2\end{bmatrix}, -1\right) = \begin{bmatrix}1+2\\1-2\end{bmatrix} = \begin{bmatrix}3\\-1\end{bmatrix}$$

and

$$\begin{bmatrix} A_{\text{odd}}(1) \\ A_{\text{odd}}(-1) \end{bmatrix} = \mathrm{FFT}(a_{\text{odd}}, -1) = \mathrm{FFT}\left(\begin{bmatrix}1\\0\end{bmatrix}, -1\right) = \begin{bmatrix}1+0\\1-0\end{bmatrix} = \begin{bmatrix}1\\1\end{bmatrix}.$$

Therefore,

$$\begin{aligned} A(1) &= A_{\text{even}}(1) + 1A_{\text{odd}}(1) = 4 \\ A(i) &= A_{\text{even}}(-1) + iA_{\text{odd}}(-1) = -1 + i \\ A(-1) &= A_{\text{even}}(1) - 1A_{\text{odd}}(1) = 2 \\ A(-i) &= A_{\text{even}}(-1) - iA_{\text{odd}}(-1) = -1 - i. \end{aligned}$$

Similarly, we can compute $B(1) = 5$, $B(i) = 2 + 3i$, $B(-1) = -1$ and $B(-i) = 2 - 3i$. Element-wise multiplication gives $C(1) = 20$, $C(i) = -5 - i$, $C(-1) = -2$ and $C(-i) = -5 + i$.

Finally, we need to convert back from the point representation to get the coefficients of $C(x)$. This is done via Inverse FFT. Note that $w^{-1} = -i$. The coefficents of $C(x)$ are given by

$$\frac{1}{4}\mathrm{FFT}\left(\begin{bmatrix}20\\-5-i\\-2\\-5+i\end{bmatrix}, -i\right) = \frac{1}{4}\begin{bmatrix}8\\20\\28\\24\end{bmatrix} = \begin{bmatrix}2\\5\\7\\6\end{bmatrix}.$$

Therefore, $C(x) = 2 + 5x + 7x^2 + 6x^3$.

Problem 2: [DPV] Problem 2.16 (find x in infinite array)
You are given an infinite array $A[.]$ in which the first n cells contain integers in sorted order and the rest of the cells are filled with ∞. You are not given the value of n. Describe an algorithm that takes an integer x as input and finds a position in the array containing x, if such a position exists, in $O(\log n)$ time.

Solution.

The basic idea of this algorithm is to first find a good estimate of the number of integers in the array, then search the array for the input. Then, the algorithm is as follows:

(1) Query $A[1], A[2], A[4], A[8] \ldots$ until $A[2^i] = \infty$.

(2) Binary search for value x in the sorted array from $A[1]$ to $A[2^i]$.

It is clear that the above algorithm finds the position of x in the array A if such position exists. It remains to prove the running time. Let i be the smallest integer such that $A[2^i] = \infty$. Since the first n cells contain integers and $A[2^{i-1}] \neq \infty$, we know that $2^{i-1} \leq n$. Hence, $2^i \leq 2n$. Therefore, we have that $i \leq \log n + 1$ and the first step of the algorithm takes $O(\log n)$. Moreover, since $2^i \leq 2n$ the second step is doing binary search on an array of size $O(n)$ and hence performing binary search in the second step also takes $O(\log n)$ time.

In fact, if you increase i only until $A[2^i] \geq x$ in step (1), then in step (2), you can do binary search in the array A on the interval $[2^{i-1}, 2^i]$ (instead of the entire interval $[1, 2^i]$). This reduces the range from size $\leq 2n$ to $\leq n$, and hence also gives $O(\log n)$ running time.

Problem 3
You are given an array $A = [1, ...n]$ of n positive numbers that represent the price of a particular stock on n days. You want to buy the stock at one day and sell on a later day. Your goal is to determine what would have been the best pair of days for buying/selling. You simply have to output the difference in price. You can assume n is a power of 2. Here is an example: $A = [10, 15, 6, 3, 7, 12, 2, 9]$. Then the optimal decision would be to purchase on day 4 for \$3 and sell on day 6 for \$12, so your algorithm should output \$9.

(a) Suppose you want to buy in the first $n/2$ days and you want to sell in the last $n/2$ days. Give an $O(n)$ time algorithm for finding the best pair of days for buying/selling under this restriction. Explain your algorithm in words and justify/explain its running time.

Solution.

Find the lowest price in the first $n/2$ days and buy on that day. Find the largest price in the last $n/2$ days and sell on that day. This will ensure the maximum profit, and there is no need for anything more complex to ensure that the purchase is before the sale. The algorithm requires one scan of the array and hence it finds the best pair in $O(n)$ time.

(b) Give a divide and conquer algorithm with running time $O(n \log n)$ for finding the best pair of days for buying/selling. (There are no restrictions on the days, except that you need to buy at an earlier date than you sell.) Explain your algorithm in words and analyze your algorithm, including stating and solving the relevant recurrence.
Hint: Your algorithm should use your solution to part (a).

Solution.

Divide the array into the first $n/2$ days and the last $n/2$ days. Observe that an optimal buy/sell pair can be one of the following cases:

- Buy and sell on the first $n/2$ days.
- Buy and sell on the last $n/2$ days.
- Buy on the first $n/2$ days and sell on the last $n/2$ days.

The first two cases can be handled using recursive calls on the first and last $n/2$ days respectively. The last case is exactly part A of this problem, and hence can be done in $O(n)$ time. The algorithm compares the three cases and returns the best solution. Therefore, we have the recurrence

$$T(n) = 2T(n/2) + O(n),$$

which solves to $T = O(n \log n)$.

(Extra Credit) Give a divide and conquer algorithm with running time $O(n)$ for finding the best pair of days for buying/selling.
Hint: Modify the problem to obtain additional info from the subproblems so that you can do the "merge" part in $O(1)$ time.

Solution.

Note that if we can handle the last case in part (b) (buy on the first $n/2$ days and sell on the last n/2 days) in $O(1)$ time, then the algorithm will take $O(n)$. To do it in $O(1)$ time we need to know the minimum in the first $n/2$ days and the maximum in the last $n/2$ days. Therefore, we modify the problem to also return a day with lowest price and a day with highest price. With this additional information, it is clear that the last case can be done in $O(1)$ time. Moreover, a day with lowest price and a day with highest price in an array of n days can also be found in $O(1)$ given the solutions of the subproblems. The recurrence becomes

$$T(n) = 2T(n/2) + O(1),$$

which solves to $T = O(n)$.

Problem 4: Integer multiplication using FFT

(a) Given an n-bit integer number a where $a = a_0a_1...a_{n-1}$, define a polynomial $A(x)$ where $A(2) = a$.

Solution.

The polynomial is

$$A(x) = a_{n-1} + a_{n-2}x + \ldots + a_0x^{n-1}.$$

Note that

$$A(2) = a_{n-1} + a_{n-2}2 + \ldots + a_02^{n-1},$$

which is exactly the value of a.

(b) Given 2 n-bit integers a and b, give an algorithm to multiply them in $O(n \log n)$ time. Use the FFT algorithm from class as a black-box (i.e. dont rewrite the code, just say run FFT on ...). Explain your algorithm in words and its running time.

Solution.

The algorithm is as follows:

1. Find polynomials $A(x)$ and $B(x)$ corresponding to a and b such that $A(2) = a$ and $B(2) = b$ respectively.
2. Compute $C(x) = A(x)B(x)$ using polynomial multiplication.
3. Evaluate $C(2)$ and return the value.

The algorithm returns $C(2) = A(2)B(2) = ab$ as desired. It remains to prove that the running time is $O(n \log n)$. The first step takes $O(n)$ to read off the bits of a and b. Since $A(x)$ and $B(x)$ have degree at most $n - 1$, the FFT algorithm on step 2 takes $O(n \log n)$. Note that $C(x)$ has degree at most $2n - 2$. Thus, evaluating $C(2)$ takes $O(n)$ arithmetic operations. Therefore, the total running time is $O(n \log n)$.

Problem 5: Deterministic Algorithm for finding the Median

For the deterministic $O(n)$ time algorithm for finding the median presented in class on Thursday, February 2, suppose we broke the array into groups of size 3 or 7 (instead of 5). Does groups of 3 or 7 work? Why? Make sure to state the recurrence $T(n)$ for the modified algorithm for each case and explain why it does or does not solve to $O(n)$.

Solution.

Assume that we break the array into groups of size k for some odd number k. Furthermore, assume that the groups $G_1, G_2 \dots G_{n/k}$ are sorted by their medians $m_1 \leq m_2 \leq \dots \leq m_{n/k}$. Let S be the set of medians. We know that the median of S is $p = m_{n/2k}$.

We will analyze the number of elements in A that are at most p. For each $1 \leq i \leq n/2k$, m_i is greater than or equal to at least $(k+1)/2$ elements in G_i. Moreover, $p = m_{n/2k} \geq m_i$ since the groups are sorted by medians. It follows that p is greater than or equal to at least $n(k+1)/4k$ elements. Therefore, the number of element greater than p is at most $n - n(k+1)/4k = n(3k-1)/4k$. The recurrence is

$$T(n) = T(n(3k-1)/4k) + T(n/k) + O(n).$$

For $k = 3$,

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

which does not solve to $O(n)$ since $2/3 + 1/3$ is not smaller than 1.

For $k = 7$,

$$T(n) = T(5n/7) + T(n/7) + O(n),$$

which solves to $O(n)$ since $5/7 + 1/7 = 6/7 < 1$.

Note that any k greater than 3 works since $(3k-1)/4k + 1/k < 1$ is equivalent to $k > 3$.