Practice Problems 2

1. Prove that a deterministic comparison-based algorithm makes at least $n-1$ comparisons to find the median of n distinct elements.

A. Assume for sake of contradiction that there exists a deterministic comparison-based algorithm A that correctly outputs the median on inputs of n distinct elements using fewer than $n-1$ comparisons.

To demonstrate the contradiction, I will construct a directed graph G using the following rules:

(a) Each of the n numbers is represented as a vertex.

(b) Each directed edge between two vertices is constructed by comparing the two numbers representing each of the vertices; i.e., an edge goes from u to v (tail at u and head at v) if u is smaller than v.

Using the assumption that there can only be fewer than $n-1$ comparisons, G will have fewer than $n-1$ edges. Therefore, G must have at least two connected components (since the minimum number of edges in a connected graph is $n-1$). Suppose now that algorithm A outputs some element x as the median, where x is in some component C_1.

In that case, pick a number y from a different component C_2. The contradiction arises from the fact that we cannot tell if y is greater than or less than x. An adversary can change the value of y, moving it to the opposite side of the median without contradicting the results of any of the comparisons done. It can do this by applying the same function to each of the vertices in C_2. To accurately classify y as less than or greater than x, there must be at least a directed path to or from the median for every other vertex in the graph. In fact, any median finding algorithm finishes its work when there has emerged a vertex (median) that can reach up to $(n-1)/2$ other vertices or to which $(n-1)/2$ other vertices can connect. Once again, at the very least, only one connected component must emerge.

Hence, as shown by the construction and subsequent contradiction above, no such algorithm A can exist. Therefore, any deterministic comparison-based algorithm that correctly outputs the median on inputs of n distinct elements must use at least $n-1$ comparisons.

2. Given a sorted array of distinct integers $A[1, ..., n]$, you want to find out whether there is an index i for which $A[i] = i$. Give a divide-and-conquer algorithm that runs in time $O(\log n)$.

A Since the array A is sorted and contains distinct integers, we can apply a binary search approach to solve this problem efficiently.

The idea is as follows:

(a) Start with two pointers: ‘left‘ pointing to the beginning of the array and ‘right‘ pointing to the end of the array.

(b) Find the midpoint `mid`:

$$\text{mid} = \left\lfloor \frac{\text{left} + \text{right}}{2} \right\rfloor$$

(c) Compare $A[\text{mid}]$ with `mid`:

- If $A[\text{mid}] = \text{mid}$, then we have found the desired index $i = \text{mid}$.
- If $A[\text{mid}] > \text{mid}$, any valid index i must lie to the left of `mid`, so update `right` to $\text{mid} - 1$.
- If $A[\text{mid}] < \text{mid}$, any valid index i must lie to the right of `mid`, so update `left` to $\text{mid} + 1$.

(d) Repeat this process until `left` exceeds `right`.

If no such index is found, return that there is no index i such that $A[i] = i$.

The algorithm divides the search interval into halves at each step, similar to binary search. Therefore, the time complexity is $O(\log n)$.

3. Suppose you have k sorted arrays, each with n elements, and you want to combine them into a single sorted array of kn elements.

   (a) Here's one strategy: Using the `merge` function, a subroutine in `mergesort`, merge the first two arrays, then merge in the third, then merge in the fourth, and so on. What is the time complexity of this algorithm, in terms of k and n?

   (b) Give a more efficient solution to this problem, using divide-and-conquer.

A. (a) Merging two arrays of size n each takes $O(2n)$ time. Merging this array with another of size n takes $O(3n)$ time. Therefore, total time complexity becomes $O(2n + 3n + 4n + ... + kn) = O(k^2 n)$

   (b) Rather than merging arrays in a linear manner, put arrays into groups of two and merge those. Then make groups of two from the result and merge those as well. Continue untill you have one array remaining, This can be visualized as a tree. Each level has $O(kn)$ work being done. the tree's height is $\log n$. So the time complexity is $O(kn \log k)$.

4. Practice with polynomial multiplication by FFT.

   (a) Suppose that you want to multiply the two polynomials $x+1$ and x^2+1 using the FFT. Choose an appropriate power of two, find the FFT of the two sequences, multiply the results component-wise, and compute the inverse FFT to get the final result.

   (b) Repeat for the pair of polynomials $1 + x + 2x^2$ and $2 + 3x$.

A. (a) Multiplication of a polynomial of degree 1 with a polynomial of degree 2 yields a polynomial of degree 3. Obviously, $3 < 4$, and 4 is a power of 2. We are given 4 coefficients, so

$$\omega = \cos\frac{2\pi}{4} + i\sin\frac{2\pi}{4} = i.$$

   The FFT of the first polynomial is

$$\text{FFT}((1,1,0,0),i) = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & i^2 & i^3 \\ 1 & i^2 & i^4 & i^6 \\ 1 & i^3 & i^6 & i^9 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1+1 \\ 1+i \\ 1-1 \\ 1-i \end{pmatrix} = \begin{pmatrix} 2 \\ 1+i \\ 0 \\ 1-i \end{pmatrix}.$$

For the second polynomial, we have

$$\mathrm{FFT}((1,0,1,0),i) = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & i^2 & i^3 \\ 1 & i^2 & i^4 & i^6 \\ 1 & i^3 & i^6 & i^9 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1+1 \\ 1-1 \\ 1+1 \\ 1-1 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \\ 2 \\ 0 \end{pmatrix}.$$

The coefficients of the product polynomial are now given by the inverse FFT of the pointwise product of these value representations. That is,

$$\mathrm{FFT}^{-1} \begin{pmatrix} 4 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \frac{1}{4}\mathrm{FFT}((4,0,0,0),i^{-1})$$

$$= \frac{1}{4} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i^{-1} & i^{-2} & i^{-3} \\ 1 & i^{-2} & i^{-4} & i^{-6} \\ 1 & i^{-3} & i^{-6} & i^{-9} \end{pmatrix} \begin{pmatrix} 4 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \frac{1}{4} \begin{pmatrix} 4 \\ 4 \\ 4 \\ 4 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}.$$

We conclude that the resulting polynomial is $1 + x + x^2 + x^3$.

(b) We still have $\omega = i$. In this case, the FFT of the first polynomial is

$$\mathrm{FFT}((1,1,2,0),i) = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & i^2 & i^3 \\ 1 & i^2 & i^4 & i^6 \\ 1 & i^3 & i^6 & i^9 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 2 \\ 0 \end{pmatrix} = \begin{pmatrix} 1+1+2 \\ 1+i-2 \\ 1-1+2 \\ 1-i-2 \end{pmatrix} = \begin{pmatrix} 4 \\ i-1 \\ 2 \\ -1-i \end{pmatrix}.$$

For the second polynomial, we have

$$\mathrm{FFT}((2,3,0,0),i) = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & i^2 & i^3 \\ 1 & i^2 & i^4 & i^6 \\ 1 & i^3 & i^6 & i^9 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 2+3 \\ 2+3i \\ 2-3 \\ 2-3i \end{pmatrix} = \begin{pmatrix} 5 \\ 2+3i \\ -1 \\ 2-3i \end{pmatrix}.$$

Now we obtain the coefficients of the product polynomial as

$$\mathrm{FFT}^{-1} \begin{pmatrix} 20 \\ -5-i \\ -2 \\ i-5 \end{pmatrix} = \frac{1}{4}\mathrm{FFT}((20,-5-i,-2,i-5),i^{-1})$$

$$= \frac{1}{4}\begin{pmatrix}1 & 1 & 1 & 1\\ 1 & i^{-1} & i^{-2} & i^{-3}\\ 1 & i^{-2} & i^{-4} & i^{-6}\\ 1 & i^{-3} & i^{-6} & i^{-9}\end{pmatrix}\begin{pmatrix}20\\ -5-i\\ -2\\ i-5\end{pmatrix} = \begin{pmatrix}2\\5\\7\\6\end{pmatrix}$$

so the product polynomial is $2 + 5x + 7x^2 + 6x^3$.