Problem 1.

1. \( T(n) = 2T(n/8) + n\log n \)
   Using Master Theorem:
   \( a = 2, b = 8 \rightarrow n^{\log_8 a} = n^{\log_8 2} = n^{1/3} \)
   \( f(n) = n\log n \rightarrow f(n) = \Omega(n^{1/3}) \)
   \( af(n/b) = 2 \times \frac{n}{8} \times \log \frac{n}{8} \leq c \times n\log n \) with \( 0.25 \leq c < 1, n > 1 \)
   Case 3, \( \Theta(n\log n) \)

2. \( T(n) = 2T(n/4) + \sqrt{n} \)
   Using Master Theorem:
   \( a = 2, b = 4 \rightarrow n^{\log_4 a} = n^{\log_4 2} = n^{1/2} \)
   \( f(n) = \sqrt{n} = n^{1/2} \rightarrow f(n) = \Theta(n^{1/2}) \)
   Case 2, \( \Theta(n^{1/2}\log n) \)

3. \( T(n) = 9T(n/3) + 8n/3 \)
   Using Master Theorem:
   \( a = 9, b = 3 \rightarrow n^{\log_3 a} = n^{\log_3 9} = n^2 \)
   \( f(n) = 8n/3 \rightarrow f(n) = O(n^{2-\epsilon}) \) with \( 0 < \epsilon < 1 \)
   Case 1, \( \Theta(n^2) \)

4. \( T(n) = 4T(n/2) + n^2\log n \)
   Using Recursion Tree:
   Tree has \( \log_2 n \) levels
   1st level cost: \( n^2 \log n \)
   2nd level cost: \( n^2 \log \frac{n}{2} \)
   3rd level cost: \( n^2 \log \frac{n}{4} \)
   4th level cost: \( n^2 \log \frac{n}{8} \)
   .....\n   Total cost = \( n^2 \log n + n^2 \log \frac{n}{2} + n^2 \log \frac{n}{4} + n^2 \log \frac{n}{8} + .... \) (\( \log_2 n \) terms) = \( (n^2 - 0.5) \log^2 n = \Theta(n^2 \log^2 n) \)
   Using Master Theorem:
   \( a = 4, b = 2 \rightarrow n^{\log_2 a} = n^{\log_2 4} = n^2 \)
   \( f(n) = n^2 \log n \rightarrow f(n) = \Theta(n^2\log^k n) \) with \( k = 1 \)
   Case 2, \( \Theta(n^2\log^2 n) \)

5. \( T(n) = T(n/3) + T(2n/3) + n \)
   The recursion tree will have different branches of different lengths. Each level will sum up to cost \( n \). The longest will be of height \( \log_{3/2} n \) (base \( 3/2 \)), this will represent the
upper bound of the algorithm, so the upper bound will be $O(n \log n)$. The shortest branch will be $\log_3 n$ (base 3), this will be the lower bound, still $\Omega(n \log n)$. Then the theta will be $\Theta(n \log n)$ as well.

Problem 2. We will describe the solution idea, however, in an exam “design an algorithm” or “describe an algorithm” means write its pseudo code.

We will start with the first naive attempt: suppose that we have $k$ sorted arrays. Since each array has $n$ elements, we would be merging array 1 with array 2, then merge that new array with array 3, then merge that new array with array 4, etc, until we have a single sorted array of $kn$ elements. We can describe this runtime as $O((n + n) + (2n + n) + (4n + n) + ... + ((k1)n + n) \rightarrow O(2n + 3n + 4n + 5n.. + kn) \rightarrow O(n \frac{k(k+1)}{2}) \rightarrow O(k^2 n)]$, $k \geq 2$ which is far too inefficient for what we want.

We are aware of a known divide-and-conquer algorithm called merge sort that recursively combines more efficiently. We can then apply the same method that merge sort uses to combine arrays: we can combine the first and second arrays then the third and fourth arrays then the fifth and sixth arrays, etc. then finally merge them together, rather than linearly combining like the naive algorithm. We would be doing $O(kn)$ work to merge the arrays $O(\log k)$ times. To elaborate, the first step merges $k/2$ sets of arrays, each set has 2 arrays of size $n$, means total cost $2n * k/2 = kn$. The second step merges $k/4$ sets of arrays, each set has 2 arrays of size $2n$, means total cost $4n * k/4 = kn$, and so on. The number of steps is $\log_2 k$ (as we merge $k/2$, then $k/4$, then $k/8...1$ array). Putting it together, the algorithms overall runtime $O(kn) * \log_2 k = O(kn \log k)$. It can be also given by a recurrence relation similar to merge sorts $T(k) = 2T(k/2) + kn \rightarrow O(kn \log k)$ by Master theorem case 2.

Problem 3. We suppose that $\bar{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ with $x_1$ and $x_2$ are vectors of length $\frac{n}{2}$. So we get:

$$W_n \cdot \bar{x} = \begin{bmatrix} W_{n/2} & -W_{n/2} \\ I_{n/2} & W_{n/2} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} W_{n/2} * x_1 - W_{n/2} * x_2 \\ I_{n/2} * x_1 + W_{n/2} * x_2 \end{bmatrix} = \begin{bmatrix} W_{n/2} * (x_1 - x_2) \\ x_1 + W_{n/2} * x_2 \end{bmatrix}.$$  

Let $T(n)$ is the time complexity to compute the product $W_n \cdot \bar{x}$. Multiple $W_{n/2}$ to a vector of length $n/2$ take $T(n/2)$ operations. Add or subtract 2 vectors of length $n/2$ takes $n/2$ operations. Thus, the time complexity $T(n)$ of the algorithm is defined by the following recurrence relation:

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T \left( \frac{n}{2} \right) + \Theta(n) & n > 1 \end{cases}$$

We apply Master Theorem with $a = 2, b = 2, f(n) = \Theta(n)$. Since $n^{\log_2 2} = n$, the second case applies and we get $T(n) = O(n \log n)$. Therefore, we will conduct a $O(n \log n)$-time algorithm that computes the product $W_n \cdot \bar{x}$ by the following steps:
1. Divide vector $\bar{x}$ into 2 vectors $x_1, x_2$ with length $\frac{n}{2}$.

2. Solve the problem recursively with 2 smaller problems: $W_{n/2}(x_1 - x_2)$ and $x_1 + W_{n/2}x_2$ with a note that $W_1 = [1]$.

3. Combine the result of 2 sub-problems to the final result.

Algorithm 1 Compute product of a $n \times n$ Weirdo matrix $W_n$ and vector $\bar{x}$, $\bar{x}$ is a vector of length $n$ (so $x$ is an array of length $n$).

1: procedure ComputeProduct($n, x$)
2: if $n = 1$ then
3: return $x$
4: end if
5: $y_1 \leftarrow x[0 : n/2 - 1]$
6: $y_2 \leftarrow x[n/2 : n - 1]$
7: $x_1 \leftarrow \bar{y}_1$
8: $x_2 \leftarrow \bar{y}_2$
9: $s_1 \leftarrow \text{ComputeProduct}(n/2, x_1 - x_2)$
10: $s_2 \leftarrow x_1 + \text{ComputeProduct}(n/2, x_2)$
11: $s \leftarrow [s_1, s_2]$
12: return $\bar{s}$ $\triangleright$ $\bar{s}$ is a vector of length $n$.
13: end procedure

**Problem 4.** We recall Merger Sort, a divide-and-conquer sorting algorithm in which we swap elements that satisfies exchanged pair’s characteristic. In other words, we will swap $x_i$ and $x_j$ if $x_i > x_j$, $i < j$. Thus, we can count the number of exchanged pairs based-on Merge Sort algorithm by tracking the number of swap operations. Since Merger Sort has time complexity $O(n \log n)$, our algorithm may get them same time complexity. We will prove it later.

This is the description of the algorithm that counts the number of exchanged pairs in array $X$:

1. Divide $X$ into 2 sub-arrays $X_1 = \{x_1, x_2, ..., x_{n/2}\}$, $X_2 = \{x_{n/2+1}, x_{n/2+2}, ..., x_n\}$.

2. Recursively sort and count the number of exchanged pairs in 2 sub-arrays $X_1$ and $X_2$.

3. After $X_1$ and $X_2$ became sorted arrays, merge the all elements in $X_1$ and $X_2$ into one array. In the merge step, if the smaller element is a member of $X_2$, we increase the counting result with number of remaining elements in $X_1$ (since $X_1$ is sorted array, all remaining elements of $X_1$ will be greater than current smallest element of $X_2$). The final result is sum of: exchanged pairs in $X_1$; exchanged pairs in $X_2$ and exchanged pairs with 1st element in $X_1$ and 2nd element in $X_2$. 
Algorithm 2 Count exchanged pairs of an array X.

1: procedure CountExchangedPairs(X)
2:   n ← length(X)
3:   if n = 1 then
4:     return 0, X
5:   end if
6:   X₁ ← X[0 : n/2 - 1]
7:   X₂ ← X[n/2 : n - 1]
8:   (c₁, X₁) ← CountExchangedPairs(X₁)
9:   (c₂, X₂) ← CountExchangedPairs(X₂)
10:  (c, X) ← MergeAndCount(X₁, X₂)
11:  return (c₁ + c₂ + c, X)
12: end procedure

Algorithm 3 Merge and count exchanged pairs of 2 sorted arrays.

1: procedure MergeAndCount(L, R)
2:   count ← 0
3:   result ← []  ➔ Set result to an empty array
4:   while L is not empty & R is not empty do
5:     if L[0] > R[0] then
6:       count ← count + length(L)
7:       result ← result append R[0]
8:       R ← R remove R[0]
9:     else
10:       result ← result append L[0]
11:       L ← L remove L[0]
12:     end if
13:   end while
14:   if L is not empty then
15:     result ← result append L
16:   end if
17:   if R is not empty then
18:     result ← result append R
19:   end if
20:  return (count, result)
21: end procedure
Now we will prove that this algorithm has the time complexity $T(n) = O(n \log n)$:

First, when $n = 1, T(n) = 1$.

According to the description of the algorithm: $T(n) = 2T(n/2) + \Theta(n)$ with $\Theta(n)$ is time complexity for merge step.

We apply Master Theorem with $a = 2, b = 2, f(n) = \Theta(n)$. Since $n^{\log_2 2} = n$, the second case applies and we get $T(n) = O(n \log n)$. 