Question 1)

Master Theorem: $T(n) = aT(\lceil n/b \rceil) + O(n^d)$

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

a) $T(n) = 8T(n/4) + O(n)$

$a = 8 \quad b = 4 \quad d = 1$

$\log_4 8 = 3/2 > d = 1$

By the master theorem, $T(n) = O(n^{\log_4 8})$ or $\boxed{T(n) = O(n^{3/2})}$

b) $T(n) = 2T(n/4) + O(\sqrt{n}) = 2T(n/4) + O(n^{1/2})$

$a = 2 \quad b = 4 \quad d = \frac{1}{2}$

$\log_4 2 = \frac{1}{2} = d = \frac{1}{2}$

By the master theorem, $\boxed{T(n) = O(n^{1/2} \log n)}$ or $O(\sqrt{n} \log n)$

c) $T(n) = T(n-4) + O(n^2)$

$T(n) = \{T((n-4)-4) + O((n-4)^2)\} + O(n^2) = T(n-4(2)) + O(n^2 - 8n + 16) + O(n^2) = T(n-4(2)) + 2O(n^2)$

$T(n) = T(n-4(3)) + 3O(n^2)$

$= T(n-4(4)) + 4O(n^2)$

$\vdots$

$T(n) = T(n-4 \cdot k) + k \cdot O(n^2)$ where $k = \lceil n/4 \rceil$

$= O(k \cdot n^2) = O(\frac{n}{4} \cdot n^2) = O(\frac{n^3}{4})$

Thus, $\boxed{T(n) = O(n^3)}$

d) $T(n) = T(\sqrt{n}) + O(n)$

let's assume $n = 2^k$

$T(2^k) = T(2^{\frac{k}{2}}) + 2^k$

If we take the log of recurrence inputs, we get

$S(k) = S(\frac{k}{2}) + 2^k$

We can now use the Master Theorem

$a = 1 \quad b = 2 \quad d = 1$

$\log_2 1 = 0 < 1 = d$

Thus $S(k) = O(2^k)$

Since $n = 2^k$, $\boxed{T(n) = O(n)}$

Question 2)

An algorithm to sort the elements of A with a runtime of $O(n+R)$ is a counting sort algorithm. This algorithm creates an array of R length (where R is the range of elements in A) and storing the count for each element in the array (with index 0 holding the smallest element and index R-1 holding the largest). Then, it would iterate through the array and insert each element into an output array depending on the count in the previously created array (using an offset value is the smallest element of A is not 0).

This algorithm is $O(n+R)$ because it fully iterates through the A array (of size n) before then fully iterating through an array of size R.

Question 3)

a) There are $\lceil n/2 \rceil$ comparisons made in the original array, and these comparisons create an array that's $\lceil n/2 \rceil$ elements long. This new array will then make $\lceil n/4 \rceil$ comparisons, and make another array of length $\lceil n/4 \rceil$. This array will execute $\lceil n/8 \rceil$ comparisons, and so on until only one element remains. The sum of all the comparisons from each array can be expressed as follows:

$$\sum_{k=1}^{\lceil \log_2 n \rceil} \frac{n}{2^k}$$

We can factor n out of the sum to work with it easier, giving us:

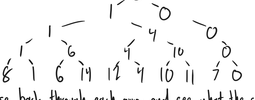
$$n \sum_{k=1}^{\lceil \log_2 n \rceil} \frac{1}{2^k}$$

Since we know that $\sum_{i=1}^{n} \frac{1}{2^i} = 1 - \frac{1}{2^n}$, we can rewrite the sum as:

$$n\left(1 - \frac{1}{2^{\lceil \log_2 n \rceil}}\right) = n\left(1 - \frac{1}{n}\right) = n - \frac{n}{n} = n - 1$$

Thus, we find that this algorithm executes $n-1$ comparisons in the worst case.

b) The current algorithm resembles a binary tree, with the minimum value of A as the root and each leaf being a distinct element in A. The following diagram gives the tree from the algorithm's execution for A = [8, 1, 6, 14, 12, 4, 10, 11, 7, 0]



Because of this, we can traverse back through each array and see what the smallest element (henceforth called E) was compared to. We can then keep track of the minimum element that was compared to E. This is because the second smallest value will always be compared to E, whether at the beginning or end. Thus, the steps to find the next smallest array are:

1) Find E by executing the original algorithm
2) Go down each array and find the value (v) that was compared to E
   i) If E has an odd index (k), v is at index k-1
   ii) If E has an even index (k), v is at index k+1 (if it exists)
3) Compare the currently stored next smallest value to v
   i) If v is smaller than the stored value, replace that value with v
4) Finish comparisons after the original array and return the value stored as the next smallest value.

c) This extended algorithm only builds off of the original algorithm instead of changing it, so the original $n-1$ comparisons are executed. After this, we traverse down the "binary tree," which has $\lceil \log_2 n \rceil$ levels (or, we have $\lceil \log_2 n \rceil$ arrays). We perform a comparison in all but one level since we can disregard the root, giving us $\lceil \log_2 n \rceil - 1$ comparisons. We need to add this to the original $n-1$ number of comparisons and when we do, we get:

$$n-1 + \lceil \log_2 n \rceil - 1 = n + \lceil \log_2 n \rceil - 2$$

Thus, we have proven that the extended algorithm does exactly $n + \lceil \log_2 n \rceil - 2$ comparisons in the worst case.