Problem 1a

T(n) = 8T(n/4) + O(n) – this recurrence fits the form necessary for application of the master theorem. A = 8, b = 4, d= 1
Log base b of a translates to log base 4 of 8 which is 1.5. d is less than log base b of a, so the runtime is $O(n^{\log_b a}) = O(n^{1.5})$.

Problem 1b

$T(n) = 2T(n/4) + O(n^{.5})$. We can also apply the master theorem to this problem. D = 1 is equal to log base b of (log base 4 of 2), so the runtime is $O(n^d \log n)$ or in this problem: $O(n^{.5} \log n)$

Problem 1c

T(n) = T(n-4) + O(n^2)
= T(n-8) + O((n-4)^2) + O(n^2)
= T(n-12) + O((n-8)^2) + O((n-4)^2) + O(n^2)
ETC

I reasoned that on every level, there's an O(n^2) operation. It then only becomes a matter of figuring out how deep the recursion goes. Since n decreases by 4 every time, it'll take about n/4 operations to get n to the base case. This completes in linear time, so the total runtime is O(n^3)

Problem 1d

T(n) = T(n^.5) + O(n)
= T(n^.25) + O(n^.5) + O(n)
= T(n^1/16) + O(n^.25) + O(n^.5) + O(n)

I can see a pattern here, but I still needed to find the depth of the recursion.

The base case is n=2, because we have to reduce n by at least 1 every time. Now we need to find how many times, k, we need to apply the square root operation to a number n to reach n=2. This is expressed by the equation.

$n^{(.5^k)} = 2 \rightarrow which\ reduces\ to \rightarrow k = \log_{\frac{1}{2}}(\log_n 2)$

This gives the summation which can then be reduced like so:

$$\sum_{k=0}^{\log_{\frac{1}{2}}(\log_n 2)} n^{\frac{1}{2}^k} = \frac{1 - n^{1/2}{}^{\log_{\frac{1}{2}}(\log_n 2)+1}}{1 - n^{1/2}} \rightarrow O(n^{\log_{\frac{1}{2}}(\log_n 2)})$$

Problem 2

I came up with a variation of counting sort.

Here is the python code:

```
def sort(unsorted):
  min_val = min(unsorted)
  r = (max(unsorted) - min_val)+1
  count = [0] * r
  output = [0] * len(unsorted)

  for i in range(len(unsorted)):
    count[unsorted[i]-min_val]+=1

  total = 0
  for i in range(r):
    previous = count[i]
    count[i] = total
    total += previous

  for i in range(len(unsorted)):
    output[count[unsorted[i]-min_val]]=unsorted[i]
    count[unsorted[i]-min_val]+=1

  return output
```

Quick summary:

The code computes the min and max and then initializes an array, output, of size n and an array, count, of size r. It loops through the input array and maps the occurrence of each number to an index in count.

Then it loops through the count array and adds all the occurrences in all lower indexes to the current index, so each index contains the cumulative number of all entries with values below the integer value of the current index.

Finally, it loops through the output array and uses the histogram stored in the count array to compute all the proper positions in ascending order.

The algorithm has several O(n) and O(R) operations, but no nested operations, so the big oh notation is just O(n+R).

Problem 3

a) Worst case is that the smallest value is one of the values compared in the last comparison, so n-1 comparisons on a list of n values.

b) The algorithm is exactly the same with a few additions. You must keep track of which entries were smaller than other entries – we could use an associative set for this. After the algorithm finishes in n-1 comparisons and finds the smallest, we go back and compare the values that were compared with the smallest entry. This is a set of size ceil(log(n)), so comparing all the entries in this set as a function of the original set is ceil(log(n)) – 1. Adding the # of comparisons yields a total # equal to n + ceil(log(n)) – 2.

c) I'll use an example to help prove the correctness.

Assume we have the sorted list of size 8: A, B, C, D, E, F, G, H. We know that A < B < C < D < E < F < G < H but assume the algorithm doesn't know this until it actually runs a comparison. We'll start the tree by comparing adjacent values.

A < B | C < D | E < F | G < H
A < C | E < H
A < E
A

We know that A is the smallest. Logically, the 2nd smallest can only be eliminated by the smallest, so we only need to check the values that "lost" to the smallest. The number of these values will be equal to the depth of the tree which we know is log(n) because the number of values is halved with every layer.