## Opening Questions

1. Algorithm A takes 5 seconds to sort 1000 records, and Algorithm B takes 10 seconds to sort 1000 records. You have the code for both algorithms. When deciding which algorithm to use to sort up to 1,000,000 records, why might Algorithm B be the better choice? Algorithm B might grow slower than algorithm A when more data is added.

2. Why is it helpful to sometimes define a problem in its most basic mathematical terms? Understanding the problem space might help with thinking of algorithmic solutions since there might be a possible shortcut rather than the brute force algorithm. It also helps to know if the problem is easy, which might mean that the brute force method would work effectively.

3. In your own words, explain what a loop invariant is. A statement that's true for every iteration of a loop. It never changes. It's a way to prove an iterative algorithm correctly.

4. What are the three kinds of growth in run time analysis we may do on an algorithm. Best case, worst case, average case.

5. For recursive algorithms, what do we need to define and solve to do the runtime analysis? Recurrence relation. Base recursive?

## Sorting as a Case Study for Algorithms Analysis

1. Write pseudocode for one of these iterative sorts: InsertionSort, BubbleSort, SelectionSort. Then draw pictures for a sample run on 5 random numbers showing comparisons/swaps.

2. Write the loop invariant for your sort and prove your sort is correct by proving Initialization, Maintenance, and Termination.

   An important property of sorting algorithms is whether or not it is stable: numbers with the same value appear in the output array in the same order as they do in the input array. Before the $i^{th}$ iteration, all of the items that were originally before position $i$ are now sorted. After the $i^{th}$ iteration, the original items $1 \rightarrow i$ are sorted (that doesn't mean they're in their final position)

   **Initialization**: Before the $i = 1$ iteration, $A[0]$ is sorted.
   **Maintenance**: Line 2 correctly sets the current starting position, lines 3 and 4 verifies the element is in the current position and walks down the item.
   **Termination**: Before the $i = n$ iteration, which doesn't exist, all items before position $n$ are now sorted.

3. Is your sort stable? (Is the relative order of duplicate values maintained when the algorithm is done?) This is stable because in line 3, the values are only swapped if one value is less than the other. But if the values are the same, the swap won't occur.

   Resource (memory or runtime) Use Analysis – Resource use analysis usually depends on the size of the input to the algorithm. You can write a function T(n) that matches the behavior

of the resource use of the algorithm. NOTE: For recursive algorithms we develop and solve a recurrence relation to find the T(n), the resource use function.

4. For the iterative sort you wrote above, construct a run time analysis function T(n) by assigning a different constant (c1, c2, c3, etc) to each type of statement (the run time for statements of that type), and counting how many times each statement executes for an input size n. Then sum up the constants times the execution counts. You may need to define variables other than n if there are event controlled iterations with an unknown number of loops.

$$T(n) = nc_1 + (n-1)c_2 + c_3 \sum_{i=1}^{n-1} t_i + \sum_{i=1}^{n-1} (t_i - 1)\,[c_4 + c_2]$$

where $t_i$ depends on initial order of the array. Worst case: opposite order where $t_i = i$. Best case: if the array is already sorted where $t_i = 1$.

Table 1.1: Constants

| Line # | Constant | Number of Executions |
|---|---|---|
| 1 | $c_1$ | $n$ |
| 2 | $c_2$ | $n-1$ |
| 3 | $c_3$ | $t_i$ |
| 4 | $c_4$ | |
| 5 | $c_5$ | |

$t_i$ is how far the $i^{th}$ item needs to walk ($0 \rightarrow (i-1)$)

For a more descriptive analysis we need to consider the various generic execution flows and input structures that result in those generic execution flows. There are 3 more descriptive resource use analyses: 1) worst case (usually used) 2) average case (sometimes used) 3) best case (hardly ever used)

5. For your T(n) function from above determine the best case, worst case and average case T(n)s.
Best Case ($t_i = 1$):

$$\begin{aligned}
T(n) &= nc_1 + (n-1)c_2 + c_3 \sum_{i=1}^{n-1} t_i + \sum_{i=1}^{n-1} (t_i - 1)\,[c_4 + c_2] \\
&= nc_1 + (n-1)c_2 + c_3 \sum_{i=1}^{n-1} (1) + \sum_{i=1}^{n-1} (1-1)\,[c_4 + c_2] \\
&= nc_1 + (n-1)c_2 + c_3(n-1) + \sum_{i=1}^{n-1} (0)\,[c_4 + c_2] \\
&= nc_1 + (n-1)c_2 + c_3(n-1) \\
&= nc_1 + (n-1)(c_2 + c_3)
\end{aligned}$$

Worst Case ($t_i = i$):

$$T(n) = nc_1 + (n-1)c_2 + c_3 \sum_{i=1}^{n-1} t_i + \sum_{i=1}^{n-1} (t_i - 1)\left[c_4 + c_2\right]$$
$$= nc_1 + (n-1)c_2 + c_3 \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} (i - 1)\left[c_4 + c_2\right]$$