

1. Sorting comparison: 0 3 4 2 6 8 7 6 8 1
2. Selection and mergesort
3. Sorting equal keys: $A_0 A_1 A_2 A_3 A_4 A_5 A_6$
 - a. Insertion Sort: $A_0 A_1 A_2 A_3 A_4 A_5 A_6$
 - b. Selection Sort: $A_6 A_5 A_4 A_3 A_2 A_1 A_0$
 - c. 2-way Quick Sort: $A_6 A_5 A_4 A_3 A_2 A_1 A_0$
 - d. 3-way Quick Sort: $A_0 A_1 A_2 A_3 A_4 A_5 A_6$
 - e. top-down merge sort: $A_0 A_1 A_2 A_3 A_4 A_5 A_6$
 - f. bottom-up merge sort: $A_0 A_1 A_2 A_3 A_4 A_5 A_6$
4. (a) quicksort. (b) quicksort (c) Quicksort partitioning (but instead of using the rightmost element, use the value 1 million, swapping all keys strictly greater than the value to the left).

5.

A. If we want to sort a set of randomly ordered items such that we get the best performance and we don't care about stability, we should use quicksort.

A or C. Again, we just want speed, but don't care about stability. If the Observations are randomly ordered, quicksort is the winner. It was also reasonable to assume that the unsorted Observation array was filled in roughly by timestamp, in which case we'd want to use insertion sort to take advantage of the partially ordered nature of the array.

B. In this case, we want speed and stability, and our objects are randomly ordered with respect to importance. The winning sort here is mergesort.

C. Here we have an array that is almost perfectly ordered, so we should use insertion sort.