Elementary Sorting Algorithms

1. Which method runs faster for an array with all keys identical, selection sort or insertion sort? Why?

> **Solution:** Insertion sort will run faster when all keys are identical. Insertion sort only keeps inserting elements until it finds a value less than or equal to the element currently being inserted. In the case of all equal elements, this will be right away. Selection sort goes through the entire remainder of the array every time to find the minimum value. It does this work even if all the elements are the same.

2. Which method runs faster for an array in reverse order, selection sort or insertion sort? Why?

> **Solution:** Both insertion sort and selection sort will take roughly the same amount of time when sorting an array in reverse order. Reverse order is insertion sort's worst case input, leading to a run time of $O(n^2)$. Selection sort has a very predictable runtime of $O(n^2)$, regardless of pre-sorted input. Selection sort will be slightly faster because it requires fewer swaps than insertion sort.

3. Suppose that we use insertion sort on a randomly ordered array where elements have only one of three values. Is the running time linear, quadratic, or something in between?

> **Solution:** The worst case in such an array is all elements bunched together but in reverse order. In such an array, insertion sort would run in $O(n^2)$ time.

4. A colleague suggests you use insertion sort to sort a singly linked list in ascending order. Why is this a bad idea? How does insertion sort's runtime complexity change if you were to use it to sort a linked list?

> **Solution:** Insertion sort repeatedly attempts to "insert" elements into their proper positions by swapping with prior elements. For a singly linked list, this would result in very poor performance because the algorithm would need to find the prior element. Finding the prior element is an $O(n)$ operation. Using insertion sort on a singly linked list would then result in $O(n^3)$ runtime complexity.

5. A colleague suggests you use selection sort for h-sorting in shellsort? Why is this a bad idea?

> **Solution:** The h-sorting process is effectively insertion sort but with larger than one element skips across the array. Selection sort has a fixed cost and a very tight $\Theta(n^2)$ bound. Selection sort cannot take advantage of skipping work when data is partially sorted already like insertion sort can. Using selection sort when h-sorting in shell sort would then undo some of the performance gains over insertion sort that shell sort provides.

6. A clerk at a shipping company is charged with the task of rearranging a number of large crates in order of the time they are to be shipped out. Thus, the cost of compares is very low (just look at the labels) relative to the cost of exchanges (moving the crates). The warehouse is nearly full: there is extra space sufficient to hold any one of the crates, but not two. What sorting method should the clerk use? Justify your answer.

> **Solution:** If the cost of exchanges is high, we should select a sorting algorithm which minimizes the number of exchanges as much as possible. In this case, one such algorithm would be selection sort.

Merge Sort

7. A colleague thinks that input arrays to the merge operation of merge sort don't need to be in sorted order. Is your colleague right or wrong? Why?

> **Solution:** The colleague is incorrect. Input arrays to the merge operation of merge sort must be sorted in order to successfully produce a result array that is also sorted. The merge operation looks at only the first element of each array to determine which element to consume. If a smaller element were to appear later in an input array, the merge operation would have no way of knowing to wait for it, thereby producing an incorrect result.

8. Give traces showing how the keys `69 65 83 89 81 85 69 83 84 73 79 78` are sorted with top-down mergesort.

Solution:

69 65 83 89 81 85 69 83 84 73 79 78			
69 65 83 89 81 85		69 83 84 73 79 78	
69 65 83	89 81 85	69 83 84	73 79 78
65 69 83	81 85 89	69 83 84	73 78 79
65 69 81 83 85 89		69 73 78 79 83 84	
65 69 69 73 78 79 81 83 83 84 85 89			

9. Give traces showing how the keys `69 65 83 89 81 85 69 83 84 73 79 78` are sorted with bottom-up mergesort.

Solution:

69 65 83 89 81 85 69 83 84 73 79 78			
69 65 83	89 81 85	69 83 84	73 79 78
65 69 83	81 85 89	69 83 84	73 78 79
65 69 81 83 85 89		69 73 78 79 83 84	
65 69 69 73 78 79 81 83 83 84 85 89			

10. An array contains n numbers, and you want to determine whether two of the numbers sum to a given number k. For instance, if the input is 8, 4, 1, 6 and k is 10, the answer is yes (4 and 6). A number may be used twice.

    (a) Describe an $O(n^2)$ algorithm to solve this problem.

> **Solution:** For each element in the array, scan through the remaining elements in the array. For each combination of elements, check if the sum equals k.

    (b) Describe an $O(n \log_2 n)$ algorithm to solve this problem. Hint: sort the items first. After doing so, you can solve the problem in linear time.

> **Solution:** Sort the array using some efficient sorting algorithm (e.g. quicksort). Once the array is sorted, place $i = 0$ and $j = |a| - 1$ (largest valid index in the array). Then there are three cases to check for:
>
> 1. If $a[i] + a[j] = k$, we've found our combination of elements

2. If $a[i] + a[j] < k$, we need a bigger sum so increment i by 1, check again
3. If $a[i] + a[j] > k$, we need a smaller sum so decrement j by 1, check again
4. If $i >= j$, there are no two elements which sum to equal k

11. Suppose instead of dividing in half at each step of merge sort, you divide into thirds, sort each third, and combine using a 3-way merge. We call this new sorting method 3-way merge sort. What is the time complexity of 3-way merge sort? Is it worth it to use 3-way merge sort over 2-way merge sort?

Solution: The time complexity of a 3-way merge sort would be $O(n \log_3 n)$. The time complexity of the normal, 2-way, merge sort is $O(n \log_2 n)$. The difference between $\log_3(n)$ and $\log_2(n)$ is not very much, even at very large values of n. Furthermore, 3-way merge sort will require more comparisons when merging three arrays together which may undo any performance gains we get via a runtime of $O(n \log_3 n)$. Therefore, it is likely not worth using 3-way merge sort.

Quick Sort

12. Show the result of standard quicksort (with no optimizations or improvements) partitioning a subarray containing `69 65 83 89 81 85 69 83 84 73 79 78`.

Solution: The first key, `69`, is the partition key. The partitioned array would have all elements less than the partition key to the left of it and all elements greater than the partition key to the right of it. Such a partition would look like `65 69 69 83 89 81 85 83 84 73 79 78`.

13. Show a trace of how standard quicksort (with no optimizations or improvements) sorts an array containing `69 65 83 89 81 85 69 83 84 73 79 78`. For the purposes of this exercise, ignore the initial shuffle.

Solution:

69 65 83 89 81 85 69 83 84 73 79 78											
69	65 83 89 81 85 69 83 84 73 79 78										
65 69		69	83 89 81 85 83 84 73 79 78								
65	69	69	83 89 81 85 83 84 73 79 78								
65	69	69	83	89 81 85 83 84 73 79 78							
65	69	69	81 83 73 79 78				83	89 85 84			
65	69	69	81 83 73 79 78				83	89	85	84	
65	69	69	81 83 73 79 78				83	84	85	89	
65	69	69	81	83 73 79 78			83	84	85	89	
65	69	69	73 79 78			81	83	83	84	85	89
65	69	69	73	79 78		81	83	83	84	85	89
65	69	69	73	78	79	81	83	83	84	85	89

14. Explain what happens when standard quicksort (with no optimizations or improvements) is run on an array having items with only two distinct keys.

Solution: If there are only two distinct keys, some a and b, either key a will be the first element of the array or key b will be. Regardless of which key is the first element, the partition sequence of quicksort will result in moving all occurrences of the other key to either the left or the right of the array. In other words, because there are only two distinct keys, the array will be sorted after the first partitioning. The work to sort the array could stop here, but without any improvements to detect such a situation, quicksort will continue to recurse and partition, even though the array is already sorted. In such a scenario, the time complexity of quicksort will be $O(n^2)$.

15. On average, standard quicksort (with no optimizations or improvements) runs in $O(n \log_2 n)$ time. However, its worst case time complexity is still listed as $O(n^2)$. Under what scenarios will quick sort perform so poorly?

Solution: Standard quicksort (with no optimizations or improvements) will perform at $O(n^2)$ when the input array is already sorted in either ascending or descending order.

16. What benefits does 3-way quicksort provide over the traditional 2-way quicksort? Remember that 3-way quicksort is a flavor of quicksort where partitioning is done based on three relational buckets (less than, equal to, and greater than the subarray).

Solution: 3-way quicksort allows quicksort to eliminate some work by detecting an array has some number of equal keys. By having a third partitioning bucket for equal keys, quicksort can avoid recursing for that particular partitioning bucket (a subarray with all elements equal to each other is already sorted).