Divide & Conquer Sorting Algorithms Efficiency & Extension

- Efficiency/time complexities for the topic are considered. Note that these discussions are not exhaustive or representative of the types of questions that may be found on the exams.
- Practice problems are given to test your understanding of the topic. These problems are not for a grade, but promote a deeper level of understanding of the topic. Unless otherwise specified, assume you are using *n* data/objects/elements in the data structure or algorithm.
- We look at the worst/average/best case of Big O; however, do not expect all three cases to be explored extensively.
- Feel free to discuss these problems in recitation or during office hours at the TA Help Desk.

Efficiency of Divide & Conquer Sorting Algorithms

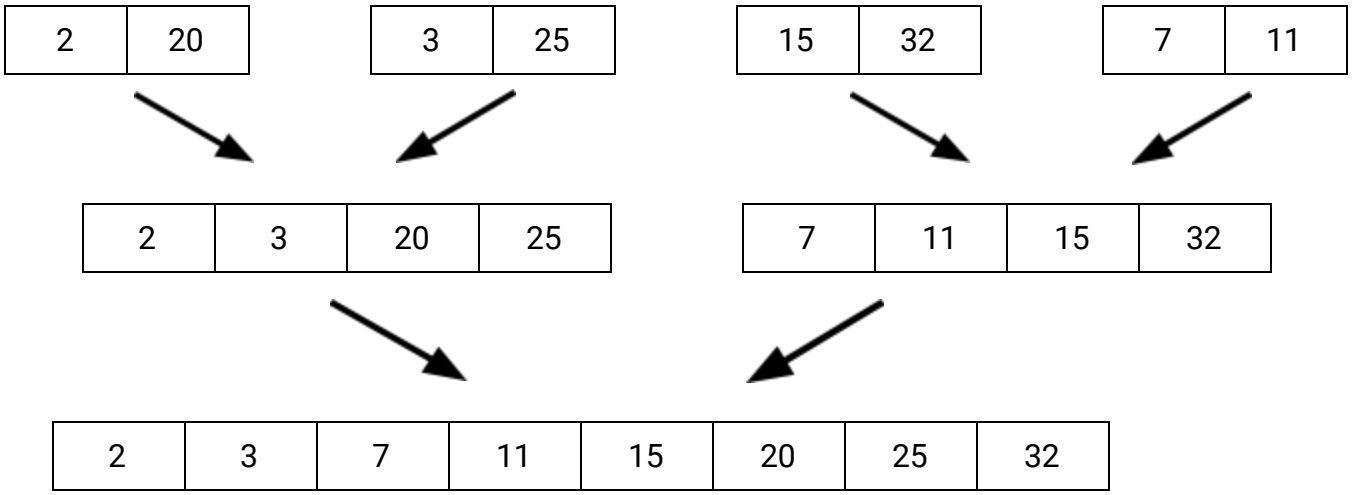
Divide and conquer algorithms explore breaking up the sorting to be done in chunks. Continuing to break down the chunks into small sizes that are easily sorted. Merge and Quick sort use recursion when breaking the array into chunks. By dividing the array in half each time, the algorithm reduces one factor of time complexity to log n. We still consider whether they are adaptive, stable and in or out of place.

Table is provided so you can record the information

Algorithm	Best Case	Average Case	Worst Case	Adaptive	Stable	In Place vs. Out of Place
Merge Sort	O (nlog n)	O(nlog n)	O (nlog n)	No	Yes	Out
In-Place Quick Sort	O (nlog n)	O (nlog n)	O (n^2)	No	No	In
LSD Radix Sort	O (kn)	O (kn)	O (kn)	No	Yes	Out
Quick Select	O (n)	O (n)	O (n^2)	Does not Apply	No	In

Merge Sort is a more complicated algorithm to understand and explain. It begins by breaking down the problem into subarrays, then merging the sorted subarrays back together. Essentially, the array is recursively split in half until it reaches size 1 subarrays. Size 1 is the base case. (You can have other base cases.) Compare the elements of size 1 or 2 array beginning at the front of each subarray, and merge them back into the larger array. The algorithm returns after each merge, and continues to merge by iterating through each subarray placing the least to most elements from the two subarrays into the larger array. The last recursive call returns to the original array. Thus, the algorithm is sorting *n* elements *log n* times. The time complexity is O(n log n).

Let's take a look at a diagram where arrays are merging back together after the recursive calls



1) How many elements are there in the array? n_____
(Think of this number in terms of n.)

2) How many levels does this diagram have in terms of n? log n_____

3) What is the average case time complexity of merge sort, given some randomized array?
_____

4) What is the worst case time complexity of merge sort, given an array where the data is in reversed order? _____

5) What is the best case time complexity of merge sort, given an array where the data is already sorted? _____

6) Is merge sort adaptive?_____

7) Does merge sort ever copy the elements into an external data structure?_____

8) Is merge sort in or out of place?_____

9) Does merge sort ever swap elements that are not adjacent to each other?_____
(Swap is not the same thing as copying elements into an array.)

10) Is merge sort stable?_____

In-Place Quick Sort is the most difficult to code and understand. However, it is easier to understand the time complexity if we think of it like we did with merge sort. It recursively subdivides the array based on a random pivot choice that will in most cases be generally in half. The base case is size 1, but you can add another base case if you choose. By subdividing, we mean it tracks the indices in the array that the algorithm operates on to place the pivot in the correct position. The algorithm is making log *n* subdivisions on *n* elements. Thus, the algorithm is sorting *n* elements *log n* times. The time complexity is O(n log n).

11) What is the average case time complexity of in-place quick sort, given some randomized array? _____

12) What is the worst case time complexity of in-place quick sort, if a bad pivot is chosen every time? _____

13) What is the best case time complexity of in-place quick sort, given an array where the data is already sorted? _____

14) Is in-place quick sort adaptive?_____

15) Does in-place quick sort ever copy the elements into an external data structure?_____

16) Is in-place quick sort in or out of place?_____

17) Does in-place quick sort ever swap elements that are not adjacent to each other?_____

18) Is in-place quick sort stable?_____

LSD Radix Sort is based on the construction of the number itself. It decomposes the digits of the number, and systematically sorts the numbers in order based on the digit. Hence the use of 'digit' buckets. The algorithm iterates through the array looking at the one's digit and placing numbers in their bucket. The buckets are then emptied inorder and the elements are put back into the array. The process is repeated for the 10's digit, 100's digit, and so on. The algorithm terminates at the end of the last iteration, knowing that the array is now sorted. The algorithm iterates based on the number of digits there are in the longest number. We refer to the number of digits as *k.* Thus, the algorithm is sorting *n* elements *k* times. The time complexity is O(k*n).

19) What is the average case time complexity of LSD radix sort, given some randomized array? O (kn)

20) What is the worst case time complexity of LSD radix sort, given an array where the data is in reversed order? O (kn)

21) What is the best case time complexity of LSD radix sort, given an array where the data is already sorted? O (kn)

22) Is LSD radix sort adaptive?_____

23) Does LSD radix sort ever copy the elements into an external data structure?_____

24) Is LSD radix sort in or out of place?_____

25) Does LSD radix sort ever swap elements that are not adjacent to each other?_____

26) Is LSD radix sort stable?_____

Quick Select searches for a single largest element (max) in the entire array. When the search completes it moves the max to last position in the array. The algorithm repeats looking for the next largest, and so on. The algorithm has to search the entire array each time to be sure it found the max. There are no comparisons of adjacent elements, only comparisons with the max where ever it is in the array. The algorithm is sorting *n* elements *n* times. The time complexity is $O(n^2)$.

27) What is the average case time complexity of selection sort, given some randomized array?
O (n^2)_____

28) What is the worst case time complexity of selection sort, given an array where the data is in reversed order? O (n^2)_____

29) What is the best case time complexity of selection sort, given an array where the data is already sorted? O (n^2)_____

30) Is selection sort adaptive?_____

31) Does selection sort ever copy the elements into an external data structure?_____

32) Is selection sort in or out of place?_____

33) Does selection sort ever swap elements that are not adjacent to each other?_____

34) Is selection sort stable?_____

Extension
Conceptual

1. Which of the sorting algorithms would sort the following array of integers most efficiently? Explain your answer.

Index	0	1	2	3	4	5	6	7	8
Element	0	1	2	3	4	5	6	-1	8

A.Quick Sort
B.Cocktail Shaker Sort
C.Radix Sort
D.Merge Sort
E.Selection Sort

Cocktail Shaker Sort because one traveersal to back and front will sort the array completely.

2. Which of the following sorting algorithm is most efficient for sorting a very large array of unsorted integers? Explain your answer.

A.Quick Sort
B.Cocktail Shaker Sort
C.Radix Sort
D.Merge Sort
E.Insertion Sort

Radix Sort would work well because we need to make very less iterations unless the the magnitude of largest element is very large

Diagramming

3. Perform one iteration of quick sort on the following array of integers, showing each step of the process. Clearly show where the i and j pointers are at each step. Use index 5 as your pivot. There may be more rows than necessary.

Index	0	1	2	3	4	5	6	7	8
Element	12	20	16	2	3	8	6	9	1

4. Will David Wang see this question?

5. Perform merge sort on the following array of integers, using arrows to clearly mark each split and merge during the process. The first split has been done for you.

Index	0	1	2	3	4	5	6	7	8
Element	12	2	8	20	4	-2	3	9	15

12	2	8	20

4	-2	3	9	15

6. Perform LSD radix sort on the following array of integers, showing the contents of the buckets and the array after each iteration. There may be more iterations than necessary.

Index	0	1	2	3	4	5	6	7	8
Element	150	401	220	80	120	400	1007	208	109

Iteration 1

Bucket	0	1	2	3	4	5	6	7	8	9
Elements										

Array										

Iteration 2

Bucket	0	1	2	3	4	5	6	7	8	9
Elements										

Array										

Iteration 3

Bucket	0	1	2	3	4	5	6	7	8	9
Elements										

Array										

7. Perform quick select on the following array of numbers finding the 3rd smallest value in the array. Choose the pivot to always be the left index.

Index	0	1	2	3	4	5	6	7	8
Element	15	4	22	8	12	-4	10	2	1