

## 1 Summary (Fill in Names of the Sorts)

**Sorting.** A sorting algorithm rearranges an array into non-decreasing order. This requires a **total order** on the keys: for any  $\mathbf{a}$  and  $\mathbf{b}$ , exactly one of  $\mathbf{a} < \mathbf{b}$ ,  $\mathbf{a} = \mathbf{b}$ ,  $\mathbf{a} > \mathbf{b}$  holds (trichotomy), and  $<$  is transitive.

A sorting algorithm is considered **stable** if order of equivalent items is preserved.

*Example: Two items, A and B, are equal. B comes before A in the unsorted array. A stable sort would ensure that B comes before A in the sorted array. An unstable sort would not guarantee this.*

\_\_\_\_\_. Pick a pivot, **partition** the array so that every item left of the pivot is  $\leq$  pivot and every item right is  $\geq$  pivot (the pivot ends up in its final sorted position), then recurse on the left and right sides.

- 3-Scan: scan three times into a new array.  $\Theta(N)$  time but  $\Theta(N)$  extra space.
- Hoare: walk one pointer in from the left (stopping on items  $\geq$  pivot) and another in from the right (stopping on items  $\leq$  pivot), swapping whenever both have stopped. In-place,  $\Theta(N)$  per partition.

Runtime: best and average  $\Theta(N \log N)$ ; worst case  $\Theta(N^2)$

Stability: Depends on partitioning scheme. 3-Scan partitioning is stable, Hoare partitioning is not.

**Avoiding the  $\Theta(N^2)$  worst case.** Four philosophies:

- **Randomness:** shuffle the input before sorting, or pick random pivots.
- **Smart pivot selection:** take the median, or an approximation such as median-of-three.
- **Introspection:** track recursion depth and switch to mergesort if recursion goes too deep.
- **Preprocessing:** try to detect pathological inputs in advance (limited effectiveness).

\_\_\_\_\_. Count the number of occurrences of each item, and store the respective counts in an array. Iterate through the unsorted array, using the count array to place each item in the sorted array.

Runtime: With an alphabet of size  $R$ , runs in  $\Theta(N + R)$

Stability: Stable

\_\_\_\_\_. Sort each digit independently from rightmost to leftmost using a stable sort as a subroutine (such as Counting Sort). Terminate once the leftmost digit has been sorted.

Runtime: With an alphabet of size  $R$  and longest key of width  $W$ , runs in  $\Theta(WN + WR)$

Stability: Stable

\_\_\_\_\_. Sort each digit independently from leftmost to rightmost using a stable sort as a subroutine (such as Counting Sort). Terminate when either the rightmost digit has been sorted or if each recursive “bucket” only contains one item.

Runtime: With an alphabet of size  $R$  and longest key of width  $W$ , runs in  $\Theta(N + R)$  in the best case and  $\Theta(WN + WR)$

Stability: Stable

## 2 Radix Sorts

- (a) Sort the following list using LSD Radix Sort with counting sort. Show the steps taken after each round of counting sort. The first row is the original list and the last two rounds are already filled for you.

	30395	30326	43092	30315
1				
2				
3				
4	<u>30315</u>	<u>30326</u>	<u>30395</u>	<u>43092</u>
5	<u>30315</u>	<u>30326</u>	<u>30395</u>	<u>43092</u>

- (b) Sort the following list using MSD Radix Sort with counting sort. Show the steps taken after each round of counting sort. The first row is the original list and the first round is already filled for you.

	21295	22316	30753	21248	30751
1	<u>21295</u>	<u>22316</u>	<u>21248</u>	<u>30753</u>	<u>30751</u>
2					
3					
4					
5					

- (c) Give the best case runtime, worst case runtime, and stability for both LSD and MSD radix sort. Assume we have  $N$  elements, a radix  $R$ , and a maximum number of digits in an element  $W$ .

	Time Complexity (Best)	Time Complexity (Worst)	Stability
LSD Radix Sort			
MSD Radix Sort			

- (d) We saw in part (c) that radix sort has good runtime with respect to the number of elements in the list. Given this fact, can we say that radix sort is always the best sort to use?

### 3 HeightComparator

Ethan, an extremely tall 61B tutor, is trying to sort the TAs by height so he can snap a photo. Can you help him out?

```
public class TA {
    private String name;
    private int height;
    public TA(String name, int height) {
        this.name = name;
        this.height = height;
    }
}
```

- (a) Implement a HeightComparator below such that it compares two TAs' height. Recall that a Comparator's compare method returns a negative number when o1 is "less than" o2, positive number when o1 is "greater than" o2, and 0 when they are the same.

```
public class HeightComparator implements Comparator<_____> {
    @Override
    public int compare(_____ o1, _____ o2){
        return _____;
    }
}
```

- (b) Anniyat suggests that we use Quicksort with our comparator. Given the following list of TAs, who would make the worst pivot? What about the best pivot?

```
TA anniyat = new TA("Anniyat", 70000);
TA aditya = new TA("Aditya", 1);
TA elana = new TA("Elana", 5);
TA sree = new TA("Sree", 7);
TA kevin = new TA("Kevin", 25);
TA elaine = new TA("Elaine", 9);
TA daniel = new TA("Daniel", 4);
TA teresa = new TA("Teresa", 8);
TA diego = new TA("Diego", 8);
```

- (c) Diego points out that even though he got in line after Teresa, he ended up in front of Teresa in the sorted list produced by Quicksort (which he doesn't like because that makes it seem like he's shorter than Teresa)! How might we ensure that Diego ends up behind Teresa?

- (d) Our TAs have just been sorted by height, but suddenly Vika and Wilson come running in late! Which sort will do the most minimal work to get them in their correct spots, and what is the additional runtime it will take (ie. not including the runtime for sorting all the other TAs first)?

## 4 LSD Radix Sort

In this question, we are trying to sort a list of strings consisting of **only lowercase alphabets** using LSD radix sort. In order to perform LSD radix sort, we need to have a subroutine that sorts the strings based on a specific character index. We will use counting sort as the subroutine for LSD radix sort.

- (a) Implement the method `stableSort` below. This method takes in `items` and an `index`. It destructively sorts the strings in `items` by their character at the `index` index alphabetically. It is stable and should run in  $O(N)$  time, where  $N$  is the number of strings in `items`.

```
private static void stableSort(List<String> items, int index) {
    Queue<String>[] buckets = new Queue[26];
    for (int i = 0; i < 26; i++) {
        buckets[i] = new ArrayDeque<>();
    }
    for (String item : items) {
        char c = _____;

        int idx = _____ - (int>('a'));

        _____;
    }

    int counter = 0;
    for (_____ ) {
        while (_____ ) {
            items.set(counter, bucket.poll());
            counter++;
        }
    }
}
```

- (b) Now, using the `stableSort` method, implement the method `lsd` below. This method takes in a `List` of `Strings` and sorts them using LSD radix sort. It should run in  $O(Nc \cdot M)$  time, where  $N$  is the number of strings in the list and  $M$  is the length of each string.

```
public static List<String> lsd(List<String> items) {
    int length = items.get(0).length();

    for (_____ ) {
        _____;
    }

    return items;
}
```