CS 61B
Summer 2020

Small Group Tutoring
Section 12: Sorting

Worksheet 12 Solutions

1 Step by Step Sorts

Show the steps taken by each sort on the following unordered list of integers (duplicate items are denoted with letters):

```
2, 1, 8, 4A, 6, 7, 9, 4B
```

1. Insertion Sort

```
2 | 1 8 4A 6 7 9 4B
1 2 | 8 4A 6 7 9 4B
1 2 8 | 4A 6 7 9 4B
1 2 4A 8 | 6 7 9 4B
1 2 4A 6 8 | 7 9 4B
1 2 4A 6 7 8 | 9 4B
1 2 4A 6 7 8 9 | 4B
1 2 4A 4B 6 7 8 9 |
```

2. Selection Sort

```
1 | 2 8 4A 6 7 9 4B
1 2 | 8 4A 6 7 9 4B
1 2 4A | 8 6 7 9 4B
1 2 4A 4B | 6 7 9 8
1 2 4A 4B 6 | 7 9 8
1 2 4A 4B 6 7 | 9 8
1 2 4A 4B 6 7 8 | 9
1 2 4A 4B 6 7 8 9 |
```

3. Merge Sort

```
[2 1 8 4A 6 7 9 4B]
[2 1 8 4A] [6 7 9 4B]
[2 1] [8 4A] [6 7] [9 4B]
[2] [1] [8] [4A] [6] [7] [9] [4B]
[1 2] [4A 8] [6 7] [4B 9]
[1 2 4A 8] [4B 6 7 9]
[1 2 4A 4B 6 7 8 9]
```

4. Heapsort *Note: if both children are equal, sink to the left.*

```
9 6 8 4A 1 7 2 4B     <-- heapified!
8 6 7 4A 1 4B 2 | 9
7 6 4B 4A 1 2 | 8 9
6 4A 4B 2 1 | 7 8 9
4A 2 4B 1 | 6 7 8 9
4B 2 1 | 4A 6 7 8 9
2 1 | 4B 4A 6 7 8 9
1 | 2 4B 4A 6 7 8 9
| 1 2 4B 4A 6 7 8 9
```

2 Sorting Runtimes

Fill out the best-case and worst-case runtimes for these sorts as well as whether they are stable or not in the table below.

	Best-Case	Worst-Case	Stability
Selection Sort	$\Theta\left(N^2\right)$	$\Theta\left(N^2\right)$	No
Insertion Sort	$\Theta(N)$	$\Theta\left(N^2\right)$	Yes
Heapsort	$\Theta(N)$	$\Theta(NlogN)$	No
Mergesort	$\Theta(NlogN)$	$\Theta(NlogN)$	Yes
Quicksort	$\Theta(NlogN)$	$\Theta\left(N^2\right)$	Depends
Counting Sort	$\Theta(N+R)$	$\Theta(N+R)$	Yes
LSD Radix Sort	$\Theta(L(N+R))$	$\Theta(L(N+R))$	Yes
MSD Radix Sort	$\Theta(N+R)$	$\Theta(L(N+R))$	Yes

Notes:
- Insertion Sort is good for small and nearly sorted arrays
- Heapsort's best case is achieved when all the items are duplicates
- Mergesort is good for sorting objects
- In practice, quicksort is the fastest comparison sort.
- In in-place implementations of quicksort, the process of moving items across the pivot doesn't guarantee that equivalent items will retain their relative positioning. Non-in-place implementations can be made to be stable, as we can retain relative orderings when adding items to the ¡, =, and ¿ arrays.

3 You Choose

1. We have a system running insertion sort and we find that it's completing faster than expected. What could we conclude about the input to the sorting algorithm?

Solution: The input is small or the array is nearly sorted. Note that insertion sort has a best case runtime of $\theta(N)$, which is when the array is already sorted.

2. Give a 5 element array such that it elicits the worst case runtime for insertion sort.

   Solution: A simple example is: 5 4 3 2 1. Any 5 integer array in descending order would work.

3. Give some reasons why someone would use merge sort over quicksort.

Solution: Some possible answers: mergesort has $\theta(NlogN)$ worst case runtime versus quicksort's $\theta(N^2)$. Mergesort is stable, whereas quicksort typically isn't. Mergesort can be highly parallelized because as we saw in the first problem the left and right sides do not interact until the end. Mergesort is also preferred for sorting a linked list.

4. Which sorts never compare the same two elements twice?

Solution: Quicksort, Mergesort, Insertion
- Quicksort: elements are compared with the pivot we pick
- Mergesort: once we compare 2 elements when we are merging they placed into a sorted lists
- Insertion: 2 sorted elements are never compared with each other - when we are inserting an element, we only compare it to sorted elements

5. When might you decide to use radix sort over a comparison sort, and vice versa?

Solution: Radix sort gives us Nk and comparison sorts can be no faster than N logN. When what we're trying to sort is bounded by a small k (such as short binary sequences), it might make more sense to run radix sort.

Comparison sorts are more general-purpose, and are better when the items you're trying to sort don't make sense from a lexicographic perspective or can't be split up into individual "digits" on which you can run counting sort. However, if comparisons take a long time, radix sort might be a better option. Consider sorting many strings of very long length that are very similar. Using a comparison sort will take at least N logN comparisons, but each comparison may require us to iterate through entire strings, giving us a runtime of NLlogN, where L is the average length of our strings.

Meanwhile, radix sort will give us a better runtime of NL + RL. On the other hand, if our long strings are very dissimilar, our comparisons will take constant time because we can quickly determine that 2 strings are unequal. In this case, a comparison sort's runtime can be N logN, which will likely be smaller than a radix sort's NL runtime.

4 Name That Sort

Below you will find some intermediate steps in performing various sorting algorithms on the same input list. The steps do not necessarily represent consecutive steps in the algorithm, but they are in the correct sequence. Identify the algorithm for each problem:
Input list: 1429, 3291, 7683, 1337, 192, 594, 4242, 9001, 4392, 129, 1000

1.
```
1429, 3291, 7683, 192, 1337, 594, 4242, 9001, 4392, 129, 1000
1429, 3291, 192, 1337, 7683, 594, 4242, 9001, 129, 1000, 4392
192, 1337, 1429, 3291, 7683, 129, 594, 1000, 4242, 4392, 9001
```

Solution: Mergesort. The left and right sides do not interact until the end.

2.
```
1337, 192, 594, 129, 1000, 1429, 3291, 7683, 4242, 9001, 4392
192, 594, 129, 1000, 1337, 1429, 3291, 7683, 4242, 9001, 4392
129, 192, 594, 1000, 1337, 1429, 3291, 4242, 9001, 4392, 7683
```

Solution: Quicksort. We chose the first item, 1429, as the pivot, which breaks up the array into three sections: < 1429, $= 1429$, > 1429

3.
```
1337, 1429, 3291, 7683, 192, 594, 4242, 9001, 4392, 129, 1000
192, 1337, 1429, 3291, 7683, 594, 4242, 9001, 4392, 129, 1000
192, 594, 1337, 1429, 3291, 7683, 4242, 9001, 4392, 129, 1000
```

Solution: Insertion Sort. Insertion sort starts from the left and move elements forward as much as possible. Since these are the first few iterations, the right side is unchanged.

4.
```
1429, 3291, 7683, 9001, 1000, 594, 4242, 1337, 4392, 129, 192
7683, 4392, 4242, 3291, 1000, 594, 192, 1337, 1429, 129, 9001
129, 4392, 4242, 3291, 1000, 594, 192, 1337, 1429, 7683, 9001
```

Solution: Heapsort. The first line is in the process of heapifying the list into a maxheap. After we created the max heap, we continuously remove the max and place it at the end.

5.
```
12, 32, 14, 11, 17, 38, 23, 34
12, 14, 11, 17, 23, 32, 38, 34
```

Solution: MSD radix sort. We have sorted by the first digit so far.