CS 61BL Sorting

Summer 2021 Recurring Section 12: Tuesday August 9, 2021

1 Mechanical Sorts

Show each pass of the following sorts on the following unordered list of integers (duplicate items are denoted with letters):

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

1. Insertion Sort

   Note that each line below corresponds to **one** pass of insertion sort. Note that this is different that the number of swaps.

   ```
   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

   Note that each line below corresponds to **one** pass of selection sort. Note a pass in selection sort involves one swap, unless the element is already in the correct place.

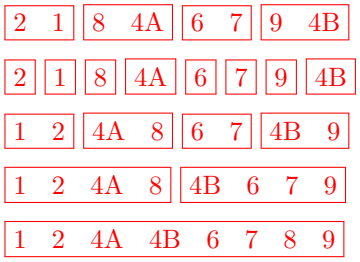
   ```
   1 | 2 8 4A 6 7 9 4B
   1 2 | 8 4A 6 7 9 4B
   1 2 4A | 8 6 7 9 4B
   1 2 4A 4B | 6 7 9 8
   1 2 4A 4B 6 | 7 9 8
   1 2 4A 4B 6 7 | 9 8
   1 2 4A 4B 6 7 8 | 9
   1 2 4A 4B 6 7 8 9 |
   ```

3. Merge Sort

   Note that each line below corresponds to **one** pass of merge sort. Note that each pass in merge sort involves merge operations or halving the arrays.

2	1	8	4A	6	7	9	4B

2	1	8	4A	6	7	9	4B



4. Heapsort. Write each swap of the bottom-up heapification process and then the result of each `removeMax` call. *Note that if both children are equal, sink to the left.*

   Note that each line below corresponds to one **swap** in the bottom-up heapification process.

   Heapification:

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

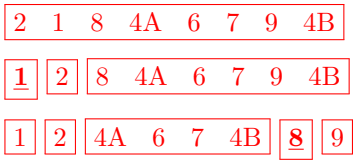
   Now, note that each line below corresponds to one `removeMax` call, and the subsequently needed swaps.

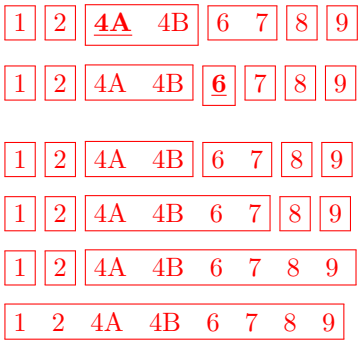
   Remove Max Calls:

```
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
```

5. Quicksort

   Note that each line below corresponds to **one** pass of quick sort. Note that each pass in merge sort involves partitioning or concatening arrays.





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 Runtime	Worst-Case Runtime	Stability
Selection Sort			
Insertion Sort			
Heapsort			
Mergesort			
Quicksort			
Counting Sort			
LSD Radix Sort			
MSD Radix Sort			

	Best-Case	Worst-Case	Stability
Selection Sort	$\Theta\left(N^2\right)$	$\Theta\left(N^2\right)$	No
Insertion Sort	$\Theta\left(N\right)$	$\Theta\left(N^2\right)$	Yes
Heapsort	$\Theta\left(N\right)$	$\Theta\left(NlogN\right)$	No
Mergesort	$\Theta\left(NlogN\right)$	$\Theta\left(NlogN\right)$	Yes
Quicksort	$\Theta\left(NlogN\right)$	$\Theta\left(N^2\right)$	Yes
Counting Sort	$\Theta\left(N+R\right)$	$\Theta\left(N+R\right)$	Yes
LSD Radix Sort	$\Theta\left(L(N+R)\right)$	$\Theta\left(L(N+R)\right)$	Yes
MSD Radix Sort	$\Theta\left(N+R\right)$	$\Theta\left(L(N+R)\right)$	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.
- For the non-in-place implementation of quicksort we've learned, 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?

   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.

   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.

   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?

   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?

   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 Challenge: Bears and Beds

The hot new Cal startup AirBearsnBeds has hired you to create an algorithm to help them place their customers in the best possible homes to improve their experience. They are currently in their alpha stage so their only customers (for now) are bears. Now, a little known fact about bears is that they are very, very picky about their bed sizes: they do not like their beds too big or too little - they like them just right. Bears are also sensitive creatures who don't like being compared to other bears, but they are perfectly fine with trying out beds.

The Problem:
Given a list of Bears with unique but unknown sizes and a list of Beds with corresponding but also unknown sizes (not necessarily in the same order), return a list of Bears and a list of Beds such that that the ith Bear in your returned list of Bears is the same size as the ith Bed in your returned list of Beds. Bears can only be compared to Beds and we can get feedback on if the Bed is too large, too small, or just right. In addition, Beds can only be compared to Bears and we can get feedback if the Bear is too large for it, too small for it, or just right for it.

The Constraints:
Your algorithm should run in $O(N \log N)$ time on average. It may be helpful to figure out the naive $O(N^2)$ solution first and then work from there.

Solution:
Our solution will modify quicksort. Let's begin by choosing a pivot from the Bears list. To avoid quicksort's worst case behavior on a sorted array, we will choose a random Bear as the pivot. Next we will partition the Beds into three groups — those less than, equal to, and greater than the pivot Bear. Next, we will select a pivot from the Beds list. This is very important — our pivot Bed will be the Bed that is equal to the pivot Bear. Given that the Beds and Bears have unique sizes, we know that **exactly** one Bed will be equal to the pivot Bear. Next we will partition the Bears into three groups — those less than, equal to, and greater than the pivot Bed.

Next, we will "match" the pivot Bear with the pivot Bed by adding them to the Bears and Beds lists at the same index, which is as easy as just adding to the end. Finally, in the same fashion as quicksort, we will have two recursive calls. The first recursive call will contain the Beds and Bears that are **less** than their respective pivots. The second recursive call will contain the Beds and Bears that are **greater** than their respective pivots.

Here is a video walkthrough of the solutions.