CS146 HW4 - Sorting

Part 1: Sorting Comparison

Prediction and Testing

Runtime for every sorting method

	Worst Case	Best Case	Average Cases
Insertion Sort	$\Theta(n^2)$	$\Theta(n)$	$O(n^2)$
Merge Sort	$\Theta(nlgn)$	$\Theta(nlgn)$	$O(nlgn)$
Heap Sort	$\Theta(nlgn)$	$\Theta(n)$	$O(nlgn)$
Quick Sort	$\Theta(n^2)$	$\Theta(nlgn)$	$O(nlgn)$

Identify worst and best case of every sorting method

	Worst Case	Best Case
Insertion Sort	Reversed sorted array	Sorted input array
Merge Sort	N/A	N/A
Heap Sort	Distinct input array	Same (duplicate) input array
Quick Sort	Pivot is either the rightmost (greatest) or leftmost (smallest) element	Pivot is middle element

Prediction

Compare: Insertion sort (avg) > Merge Sort (avg) = Heap Sort (avg) = Quick Sort (avg)
Compare Sorted: Quick sort (avg = best) = Merge Sort (avg) = Heap Sort (worst = avg) > Insertion Sort (best)
Compare Reverse Sorted: Insertion Sort (worst) > Quick Sort (avg = best) = Merge Sort = Heap Sort (worst)
Compare duplicate: Insertion Sort (avg) = Quick Sort (worst) > Merge Sort > Heap Sort (best)

Result

Compare Method

	10^1	10^2	10^3	10^4
Insertion Sort	2,300	68,100	1,877,300	25,057,100
Merge Sort	9,001	75,701	429,601	1,895,800
Heap Sort	513,799	605,599	1,038,300	2,857,299
Quick Sort	440,400	406,501	660,300	1,672,400

Compare Sorted Method

	10^1	10^2	10^3	10^4
Insertion Sort	599	2,500	8,100	35,500
Merge Sort	5,700	69,600	159,699	1,260,799
Heap Sort	5,700	20,400	152,800	1,931,700
Quick Sort	2,100	61,500	908,600	32,680,500

Compare Reverse Sorted Method

	10^1	10^2	10^3	10^4
Insertion Sort	1,700	101,500	7,772,399	37,320,100
Merge Sort	4,901	18,599	98,200	664,100
Heap Sort	4,599	14,100	109,400	1,146,900
Quick Sort	2,500	22,500	245,500	26,816,700

Compare Duplicate Method

	10^1	10^2	10^3	10^4
Insertion Sort	600	2,601	70,401	10,400
Merge Sort	6,300	8,999	70,401	636,899
Heap Sort	2,901	8,300	55,500	476,200
Quick Sort	2,401	8,600	51,900	137,800

Answer

1. Relative order of the sorting algorithms (fast to slow order)

- Compare method

10^1	10^2	10^3	10^4
Insertion Sort	Insertion Sort	Merge Sort	Quick Sort
Merge Sort	Merge Sort	Quick Sort	Merge Sort
Quick Sort	Quick Sort	Heap Sort	Heap Sort
Heap Sort	Heap Sort	Insertion Sort	Insertion Sort

- Compare Sorted Method

10^1	10^2	10^3	10^4
Insertion Sort	Insertion Sort	Insertion Sort	Insertion Sort
Quick Sort	Heap Sort	Heap Sort	Merge Sort
Merge Sort (tied)	Quick Sort	Merge Sort	Heap Sort
Heap Sort (tied)	Merge Sort	Quick Sort	Quick Sort

- Compare Reverse Sorted Method

10^1	10^2	10^3	10^4
Insertion Sort	Heap Sort	Merge Sort	Merge Sort
Quick Sort	Merge Sort	Heap Sort	Heap Sort
Heap Sort	Quick Sort	Quick Sort	Quick Sort
Merge Sort	Insertion Sort	Insertion Sort	Insertion Sort

- Compare Duplicate Method

10^1	10^2	10^3	10^4
Insertion Sort	Insertion Sort	Insertion Sort	Insertion Sort
Quick Sort	Heap Sort	Heap Sort	Quick Sort
Heap Sort	Quick Sort	Quick Sort	Heap Sort

Merge Sort	Merge Sort	Merge Sort	Merge Sort

a. Most of my comparing order predictions do not match reality. Some worst and best cases are predictable, For example, in the compare reverse sorted method, the insertion sort performed poorly compared to other methods as expected. I think the factors that may lead to the differences in my predictions are the input size, and implementation (how every sorting method was programmed).
b. The size of the input did affect the performance of the sorting algorithms, which caused the change in order within a comparing method

2. Every sorting algorithm

Insertion sort algorithm: As expected, the insertion sort algorithm performs great in its best case (sorted input array and with hight loads of input) and poorly in its worst case (reversed sorted array). In addition, for higher loads of input in the compare method, insertion sort takes much longer runtime.

Merge sort algorithm: As merge sort does not have any worst and best cases' differences, it runs consistently for all cases

Heap sort algorithm: Heap sort's worst is all distinct input array, which it shows that the runtime is much slower than other sorting methods for the smaller loads of input. However, it still performs sorting faster than insertion sort for hight loads of input

Quick sort algorithm: quick sort performs great for many large loads of input. However, in its best case (sorted array), it still runs much slower than other sorting methods. This is what I do not expect.

3. Quicksort

In the compare method, quick sort runs quite slow compared to other algorithms in the smaller loads of input. However, it performs much better with higher loads of input. Otherwise, I do not see much of outstanding performance of quick sort algorithms in other cases However, the quicksort has a stack overflow problem when it comes to the 10^5 numbers of input, I think specifically in the compareReverseSorted method since the initial pivot is assigned as the arr[low] (the first element in the array). I made a small change that the pivot is the middle element (arr[(low + high) /2) in the array so that it makes most of the cases (especially the reverse sorted array) becomes the best case for the quick sort and prevents the sorting process from stack overflow.

Note: I leave the code in the submitted project file as the code that the professor provided because I was not sure if I was allowed to change the signatures in the provided files

Part 2: Radix Sort Using a Queue

Answer

1. Using arrays of Queues gives us more flexibility for queuing and dequeuing elements in FIFO order. The LIFO order operation from stack would make it difficult to manage the elements' in-and-out flow
2. Justify my program
   The outer loop (get digit place for each element in the array)

```
public static void radixSort(int[] arr) {
// Get the maximum to know how many digits I have
int max = getMax(arr);

for (int place = 1; max / place > 0; place *= 10) {
  queueSort(arr, place);
}
}
```

This loop should be running at most 10 times because the input array's element range is [0, 2147483647]. The greatest element has 10 digits -> O(10)

Sorting method

```
  private static void queueSort(int[] arr, int place) {
    int size = arr.length;
    Queue[] queueArr = new Queue[10];

    for (int i = 0; i < 10; i++) {
      queueArr[i] = new Queue();
    }

    for (int j = 0; j < size; j++) {
      queueArr[(arr[j] / place) % 10].enqueue(arr[j]);
    }

    int j = 0;
    for (int i = 0; i < 10; i++) {
      while (!queueArr[i].isEmpty()) {
        arr[j++] = queueArr[i].dequeue();
      }
    }
  }
```

For the first loop, we have a runtime of O(10)
In the second loop, we have a runtime of O(n) since the input array has a length of n
In the third loop, we have a runtime of O(10n), which we have to run through an array of 10 queues and the array has n elements.

To sum up, we have the runtime of the RadixSort using an array of queues as follows:
$O(10 * (10 + n + 10n)) = O(100 + 110n) = O(n)$ -> linear time

Therefore, my program would run at most O(n) time