Sort Comparison

Jeffrey Wan

John Hopkins University

Author Note

It was actually pretty great to learn how to implement recursively and use Quick and Heap sort. Quick sort was fairly tricky to implement with the Insertion sort base case. It took extra care to use the right indices for the low and high indices of the partition in each recursive call. Generating unsorted raw data also was a bit tricker than expected.

Abstract
This program is an application that compares 4 versions of Quick sort and 1 version of Heap sort. The 4 versions of Quick sort differ from each other either by pivot choice or when the sort opts to use insertion sort on a specific partition size. Various types if input data are generated, varying by size and order (one file is sorted in ascending, reversed, or random order). Runtimes for each sort are averaged across 5 runs and recorded. The output of each sort is written to file as well.

Design and Analysis

Here are the runtimes for each sort, in 4 files. You can click on the image and scroll. The file itself is stored in output/timeTables.txt

```
ELAPSED PER SORT FOR FILE: reversed500.dat=============
first element as pivot, and partition size of 1 or 2, TIME TO SORT IN NANOSECON

first element as pivot, and partition size of 50, TIME TO SORT IN NANOSECONDS:

first element as pivot, and partition size of 100, TIME TO SORT IN NANOSECONDS:

median element as pivot, and partition size of 1 or 2, TIME TO SORT IN NANOSECO

TO SORT IN NANOSECONDS: 55555

ELAPSED PER SORT FOR FILE: reversed5000.dat=============
first element as pivot, and partition size of 1 or 2, TIME TO SORT IN NANOSECON

first element as pivot, and partition size of 50, TIME TO SORT IN NANOSECONDS:

first element as pivot, and partition size of 100, TIME TO SORT IN NANOSECONDS:

median element as pivot, and partition size of 1 or 2, TIME TO SORT IN NANOSECO

TO SORT IN NANOSECONDS: 762715
```

4 FILES OF RUNTIMES

```
Quicksort using median element as pivot, and partition size of 1 or 2, TIME TO SORT IN NANOSECONDS: 226229

Heapsort, TIME TO SORT IN NANOSECONDS: 123478

==========TIME ELAPSED PER SORT FOR FILE: reversed2000.dat=============
Quicksort using first element as pivot, and partition size of 1 or 2, TIME TO SORT IN NANOSECONDS: 2036249

Quicksort using first element as pivot, and partition size of 50, TIME TO SORT IN NANOSECONDS: 1682144

Quicksort using first element as pivot, and partition size of 100, TIME TO SORT IN NANOSECONDS: 1846298

Quicksort using median element as pivot, and partition size of 1 or 2, TIME TO SORT IN NANOSECONDS: 807102

Heapsort, TIME TO SORT IN NANOSECONDS: 256180

==========TIME ELAPSED PER SORT FOR FILE: reversed50.dat=============
Quicksort using first element as pivot, and partition size of 1 or 2, TIME TO SORT IN NANOSECONDS: 5588

Quicksort using first element as pivot, and partition size of 50, TIME TO SORT IN NANOSECONDS: 7019

Quicksort using first element as pivot, and partition size of 100, TIME TO SORT IN NANOSECONDS: 2965

Quicksort using median element as pivot, and partition size of 1 or 2, TIME TO SORT IN NANOSECONDS: 2193

Heapsort, TIME TO SORT IN NANOSECONDS: 2865

==========TIME ELAPSED PER SORT FOR FILE: reversed500.dat=============
Quicksort using first element as pivot, and partition size of 1 or 2, TIME TO SORT IN NANOSECONDS: 114165

Quicksort using first element as pivot, and partition size of 50, TIME TO SORT IN NANOSECONDS: 137836

Quicksort using first element as pivot, and partition size of 100, TIME TO SORT IN NANOSECONDS: 110342

Quicksort using median element as pivot, and partition size of 1 or 2, TIME TO SORT IN NANOSECONDS: 60283
```

TYPE TO ENTER A CAPTION.

```
Quicksort using first element as pivot, and partition size of 1 or 2, TIME TO SORT IN NANOSECONDS: 81008

Quicksort using first element as pivot, and partition size of 50, TIME TO SORT IN NANOSECONDS: 64873

Quicksort using first element as pivot, and partition size of 100, TIME TO SORT IN NANOSECONDS: 73549

Quicksort using median element as pivot, and partition size of 1 or 2, TIME TO SORT IN NANOSECONDS: 81345

Heapsort, TIME TO SORT IN NANOSECONDS: 263376

==========TIME ELAPSED PER SORT FOR FILE: rand2000.dat=============
Quicksort using first element as pivot, and partition size of 1 or 2, TIME TO SORT IN NANOSECONDS: 333191

Quicksort using first element as pivot, and partition size of 50, TIME TO SORT IN NANOSECONDS: 309905

Quicksort using first element as pivot, and partition size of 100, TIME TO SORT IN NANOSECONDS: 251950

Quicksort using median element as pivot, and partition size of 1 or 2, TIME TO SORT IN NANOSECONDS: 196811

Heapsort, TIME TO SORT IN NANOSECONDS: 274209

==========TIME ELAPSED PER SORT FOR FILE: rand50.dat=============
Quicksort using first element as pivot, and partition size of 1 or 2, TIME TO SORT IN NANOSECONDS: 7943

Quicksort using first element as pivot, and partition size of 50, TIME TO SORT IN NANOSECONDS: 2338

Quicksort using first element as pivot, and partition size of 100, TIME TO SORT IN NANOSECONDS: 4233

Quicksort using median element as pivot, and partition size of 1 or 2, TIME TO SORT IN NANOSECONDS: 1716

Heapsort, TIME TO SORT IN NANOSECONDS: 9272

==========TIME ELAPSED PER SORT FOR FILE: rand500.dat=============
Quicksort using first element as pivot, and partition size of 1 or 2, TIME TO SORT IN NANOSECONDS: 74630

Quicksort using first element as pivot, and partition size of 50, TIME TO SORT IN NANOSECONDS: 22828
```

TYPE TO ENTER A CAPTION.

```
Quicksort using median element as pivot, and partition size of 1 or 2, TIME TO SORT IN NANOSECONDS: 173747

Heapsort, TIME TO SORT IN NANOSECONDS: 823721

==========TIME ELAPSED PER SORT FOR FILE: asc2000.dat=============
Quicksort using first element as pivot, and partition size of 1 or 2, TIME TO SORT IN NANOSECONDS: 1401497

Quicksort using first element as pivot, and partition size of 50, TIME TO SORT IN NANOSECONDS: 1166425

Quicksort using first element as pivot, and partition size of 100, TIME TO SORT IN NANOSECONDS: 1476313

Quicksort using median element as pivot, and partition size of 1 or 2, TIME TO SORT IN NANOSECONDS: 745347

Heapsort, TIME TO SORT IN NANOSECONDS: 294718

==========TIME ELAPSED PER SORT FOR FILE: asc50.dat=============
Quicksort using first element as pivot, and partition size of 1 or 2, TIME TO SORT IN NANOSECONDS: 2309

Quicksort using first element as pivot, and partition size of 50, TIME TO SORT IN NANOSECONDS: 1970

Quicksort using first element as pivot, and partition size of 100, TIME TO SORT IN NANOSECONDS: 2874

Quicksort using median element as pivot, and partition size of 1 or 2, TIME TO SORT IN NANOSECONDS: 2594

Heapsort, TIME TO SORT IN NANOSECONDS: 5455

==========TIME ELAPSED PER SORT FOR FILE: asc500.dat=============
Quicksort using first element as pivot, and partition size of 1 or 2, TIME TO SORT IN NANOSECONDS: 129207

Quicksort using first element as pivot, and partition size of 50, TIME TO SORT IN NANOSECONDS: 138869

Quicksort using first element as pivot, and partition size of 100, TIME TO SORT IN NANOSECONDS: 159206

Quicksort using median element as pivot, and partition size of 1 or 2, TIME TO SORT IN NANOSECONDS: 74985

Heapsort, TIME TO SORT IN NANOSECONDS: 155844
```

TYPE TO ENTER A CAPTION.

Some observations about the runtimes**:**

- **Quick sort does well with small data sets** (size 50 or 5000 relative to heap sort; it's actually faster. At the smallest data set sizes, Quick sort is faster. it's possible that the simplicity and low overhead of insertion sort is why the Quick sorts do well even though it is one of the elementary sorting algorithms with O(n2) worst-case time.
- **Quicksort performs faster than Heap sort when the data is randomly sorted**. This is probably because the random data makes it likely that the pivot chosen does not lead to uneven partitions. This makes sense because the worst case of Quick sort is O(N^2) which occurs when the picked pivot is always an extreme (smallest or largest) element which happens when input array is sorted or reverse sorted and either first or last element is picked as pivot. But when the data is random, the % that a good pivot is picked is high for quicksort which is why it outperforms or performs equivalent to Heap sort.
- **However, at larger sizes (ascending 2000, ascending 5000, reversed at size 1000+) for non-random unsorted files, Heap sort outperforms Quick sort**. This makes sense since Heap Sort has has an n*logn time complexity and so performs better relative to Quick Sort as n grows larger. Heap sort performs best relative to Quick sort at larger sizes. The time complexity of Quick sort when the data is random is O(n*logn*) time.
- **Here are some relative performances of the sorts.** All of the info is recorded in output/timetable.txt
  - For sorted data at size 5000, heapsort is twice as fast as quicksort.
  - For random data at size 5000, heapsort is 1.5 as fast as quicksort.
  - For reversed data at size 5000, heapsort is 4x as fast as quicksort.
  - For sorted data at size 2000, heapsort is 3-4x times as fast as quicksort.
  - For random data at size 2000, heapsort is actually worse than quicksort.
  - For reversed data at size 2000, heapsort performs 3-4x times as fast as quicksort.
  - For sorted data at size 1000, heapsort is twice times as fast as quicksort.
  - For random data at size 1000, heapsort is actually worse than quicksort.
  - For reversed data at size 1000, heapsort is 4x as fast as quicksort.
- Heap sort's time complexity is O(n*logn) which scales well as n grows unlike Quick sort which grows at n^2. **Unlike Quick sort there's no worse case and it shows… the sort time for Heap sort is fairly constant within input sizes regardless of the data ordering.** The time for Heap sort is approximately the same within data sizes and data sorting, e.g., the time for Heap Sort at size 5000 is approximately the same regardless of whether the data is ordered, reversed, or random.
- **Quicksort performs best within a data size when the data is randomly ordered.** This makes sense because random order drastically increases the chance that a pivot is chosen that divides the data into even partitions and not one-sided partitions.

- Interestingly, **it seems like using Insertion sort on the ascending files earlier (when the partition size is large) instead of using Quick sorting the entire time is faster**. Insertion sort is low overhead and even though it's an O(n^2) algorithm, there is less overhead and fewer method calls compared to Quick sort. Also, insertion sort is fastest when the data is already sorted and has an O(n) runtime in that case. Ex: for an ascending file size of 5000, Quick sort using insertion sort at a size 100 partition is faster than a Quick sort using insertion sort at a size 2 partition.
- **It seems like the biggest effect is pivot choice for Quick sort.** It seems like for larger file sizes, a pivot choice that isn't the worse choice (like using the median of three numbers) halves the runtime. The effect and time saved is magnified as the file size gets bigger.
- For Quick sort, **selecting a median pivot seems to have the same effect as using a random file**; both have the effect of drastically reducing the chance of using an extreme pivot resulting in uneven partitions.

The recursive solutions were easier to implement; the Quick sort partitioning and the subtrees of the heap lent well to recursion. The base cases of partition size for Quick sort and the children nodes not having children for Heap sort were easy to implement for the recursive solutions as well. If iterative solutions were chosen, the methods would have to keep track of the partition sizes on each iteration and if there were partitions left to reorder for Quick sort and whether there were nodes left to swap for Heap sort. In the iteration solutions, I think there would need to a data structure to store subtrees and partitions to sort.

<u>Some additional features;</u>

1.
In order to get reasonably accurate times for each sort, the sort is run 5 times and the average of the times is recorded in the time table. Since each sort algorithm makes a copy of the incoming data, the data does not need to reset on each run; the original data is left unaffected on each run. Here is the copying code in Quick sort;

```
int[] dataCopy = new int[size];
for (int i = 0; i <= size - 1; i++) {
    dataCopy[i] = arr[i];
}
```

and in Heap sort:

```
int[] sorted = new int[sortedSize];
// pluck the min value off from heap. use SortedSize as the stopping case.
for(int i = 0; i < sortedSize; i++) {
```

```
    sorted[i] = removeMin();
}
```

This is the method that runs the sorts 5 times.

```
private static Pair<Integer, int[]> runAndGetAverageTime(Sorter sorter, int[]
dataInput) {
    long totalTime = 0;
    int totalRuns = 4;

    for (int i = 0; i <= totalRuns; i++) {
        long startTime = System.nanoTime();
        sorter.sort(dataInput);
        long endTime = System.nanoTime();
        long runTime = endTime - startTime;

        totalTime += runTime;
    }
    int[] sorted = sorter.sort(dataInput);
    long avgTime = totalTime / totalRuns-1;

    return new Pair(avgTime, sorted);
}
```

The above method runs a sort that implements Sorter (an interface) and calls the sort method on the passed in Sorter. The times are recorded and then averaged. One last sort is called to store the sorted Array and the two results are returned from this single method via the Pair class which is a generic class that can store two values of different types.

2.

I created my own input files since the provided ones were wrong or had incorrectly sorted data. None of the input files have duplicates by default.

```
private static void generateInputFilesSinceTheLabInputFilesAreWrong() throws
IOException, Quicksort.InvalidPartitionStoppingChoice, Quicksort.InvalidPivot {
    int[] INPUT_SIZES = new int[] {50, 500, 1000, 2000, 5000};
    for (int size : INPUT_SIZES) {
        generateRandomFiles(size);
        generateOrderedFiles(size);
        generateReverseOrderedFiles(size);
    }
}
```

The input and output files are located in folders provided by command line arguments:

```
public static void main(String args[]) throws IOException,
Quicksort.InvalidPivot, Quicksort.InvalidPartitionStoppingChoice {
    inputFolderName = args[0] + "/";
    outputFolderName = args[1] + "/";

    // this generates the input and output folders
    generateDirectory(args[0]);
    generateDirectory(args[1]);
    // this generates the input files that need to be sorted
    generateInputFilesSinceTheLabInputFilesAreWrong();
```

3.
I wrote a function to check whether the output files are all sorted:

```
checkIfEverythingIsSorted();
```

That function runs this code:

```
if (!isSorted(data) && !isSortedReversed(data)) {
    throw new FileNotSorted("You made a mistake somewhere you idiot: " +
fileName + " is not sorted");
}
```

If there's an error, it throws an error. If not the output in terminal should look like this:

```
Checking output files are all sorted...
This file was sorted: reversed5000_sorted.txt
This file was sorted: reversed1000_sorted.txt
This file was sorted: asc500_sorted.txt
This file was sorted: reversed500_sorted.txt
This file was sorted: reversed2000_sorted.txt
…
```

4. We also have an args checker at the beginning of the application:

```
checkArgs(args);
```

If the proper number of args aren't provided like this:

```
java SortComparison inputData outputData
```

Then this is the output in terminal:

```
java SortComparison inputData
Proper Usage is: java SortComparison inputData outputData
```

Potential problems:
- The length is inferred from the file name. This is brittle and likely to fail if the title of the file does not have the correct length. The length is used to create array sizes and if the arrays is not large enough or has too many padded zeroes, this algorithm will fail.

Lessons Learned

Quick sort:
- Sorting ordered files in quick sort requires fewer comparisons than reverse order. That's probably why it is faster.
- overall cost of Quick sort can be broken down into two parts: the partitioning and the number of times that we need to partition. The pivot is compared to every element in partition and if it's bigger or smaller than pivot. The partitioning process is linear in relation to the size of the partition. There is linear cost for partitioning for each level, and there are log(n) levels. So, the time complexity of Quick sort is O(n * log(n)).
- Picking an arbitrary element is a pretty good way of picking a pivot. Sometimes a random element is a good pivot and gives equal partitions, other times it's not and gives uneven partitions. That's why the random input data sometimes gives fast times for Quick sort and other times slow times.

Some additional features:

- Included Javadocs in the doc folder.