Please send any comments or criticisms to dwharder@alumni.uwaterloo.ca
with the subject ECE 250 Notes 8.6.
Assistances and comments will be acknowledged.

## 8.6 Quicksort

Merge sort divided an unsorted list into two approximately equal sub-lists based on location. We will look at an alternate strategy for dividing a list into two sub-lists: select one entry in the list (call it a *pivot*) and separate all other entries as to whether they are smaller than or larger than this pivot.

Using this idea, quicksort is, on average, faster than merge sort and has the following properties, but there are some issues, as are shown in Table 1.

Table 1. The run times of quicksort.

| | Run Time | Memory |
|---|---|---|
| Average Case | $\Theta(n \ln(n))$ | $\Theta(\ln(n))$ |
| Worst Case | $\Theta(n^2)$ | $\Theta(n)$ |

We will look at this algorithm but we will also look at strategies for avoiding the worst-case scenario.

### 8.6.1 Strategy and Run-time Analysis

Suppose, we split a list into two sub-lists by picking one entry (the *pivot*) and dividing all other entries into those less than the pivot and those greater than the pivot. For example, if we select 44 from this list

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 80 | 21 | 95 | 84 | 66 | 10 | 79 | **44** | 26 | 87 | 96 | 12 | 43 | 71 | 3 |

This would produce the list

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 21 | 10 | 26 | 12 | 43 | 3 | **44** | 80 | 95 | 84 | 66 | 79 | 87 | 96 | 71 |

Each of these two lists contains approximately $n/2$ entries. Notice that 44 is now in the correct location if the array is entirely sorted. We could continue by using a similar algorithm on the first six entries and the last eight entries.

Thus, if we were to reapply this algorithm and always get two sub-lists of approximately half the size at each step, the run time would be similar to that of merge sort: $\Theta(n \ln(n))$. We can also apply our simplification of using insertion sort if the size of the list ever drops below some fixed $N$.

### 7.4.2 Worst-case Scenario

Unfortunately, we might get unlucky:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 80 | 21 | 95 | 84 | 66 | 10 | 79 | **2** | 26 | 87 | 96 | 12 | 43 | 71 | 3 |

Using 2 as a pivot results in the partition

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **2** | 80 | 21 | 95 | 84 | 66 | 10 | 79 | 26 | 87 | 96 | 12 | 43 | 71 | 3 |

At this point, we must sort the remaining list of size $n - 1$. Thus, the run time may be described by

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ T(n-1) + \Theta(n) & n > 1 \end{cases}$$

This is no different from the run-time of selection sort: $\Theta(n^2)$.

### 8.6.3 Median-of-Three Strategy

The median of $n$ entries is that entry such that half of all values are less than the median (and, therefore, the other half are greater than that entry).

The politician was astonished to learn that half of all
Canadians were below the median intelligence.

The ideal case is to choose the median; however, we cannot find the median entry quickly. Instead, an alternate strategy is to choose three entries, say, the first, middle, and last entries and choose the median of these three entries. Going back to our initial example, the median of the entries {80, 44, 3} is 44.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **80** | 21 | 95 | 84 | 66 | 10 | 79 | **44** | 26 | 87 | 96 | 12 | 43 | 71 | **3** |

We can now partition the entries based on the pivot 44:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 21 | 10 | 26 | 12 | 43 | 3 | **44** | 80 | 95 | 84 | 66 | 79 | 87 | 96 | 71 |

Applying the median-of-three on the first six,

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 21 | 10 | 26 | 12 | 43 | 3 | **44** | 80 | 95 | 84 | 66 | 79 | 87 | 96 | 71 |

would be the median of {21, 26, 3} which is 21 while the median of the last eight,

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 21 | 10 | 26 | 12 | 43 | 3 | **44** | 80 | 95 | 84 | 66 | 79 | 87 | 96 | 71 |

would be the median of {80, 66, 71} or 71.

Using statistics and distributions, it is possible to show that using a random pivot will, on average, partition the list into two sub-lists of sizes $\frac{1}{4}n$ and $\frac{3}{4}n$. Also, 90 % of the time, the sub-lists will have sizes $\frac{1}{20}n$ and $\frac{19}{20}n$ or better (1:19 or better):

$$2\int_0^{\frac{1}{2}}(x\cdot 1)\,dx = \frac{1}{4}$$

$$\int_{\xi}^{1-\xi}(1)\,dx = 0.90 \Rightarrow \xi = 0.05$$

To do the same for the median-of-three, it is necessary to understand *order statistics*: what is the distribution of the middle of three random numbers chosen from an interval [0, 1]. It happens that the distribution of the smallest, middle, and largest of three random selections is shown in Figure 1.

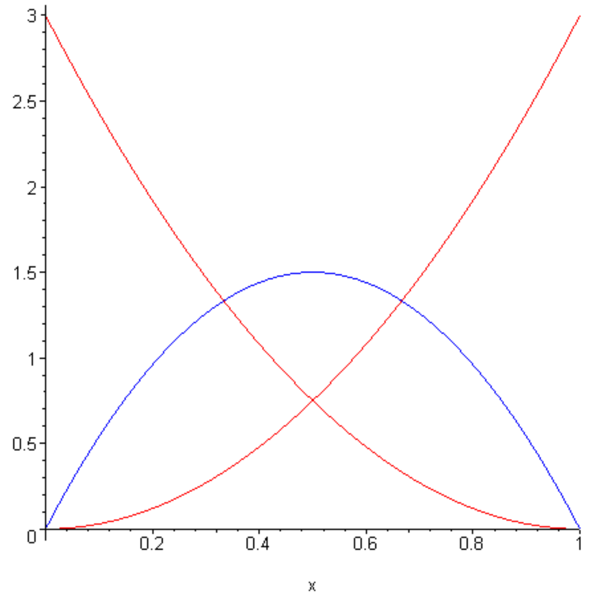

Figure 1. The distribution of the smallest, median, and largest of three random numbers from [0, 1].

Without delving into the details, we have that the median-of-three will, on average, partition a list into two sub-lists of sizes $\frac{5}{16}n$ and $\frac{11}{16}n$. Now, 90 % of the time, the ratio of the 0.135*n* and 0.865*n* or better (1:6.388 or better):

$$2\int_0^{\frac{1}{2}}\left(x\cdot\left(6x(1-x)\right)\right)dx = \frac{5}{16}$$

$$\int_{\xi}^{1-\xi}\left(6x(1-x)\right)dx = 0.90 \Rightarrow \xi = 0.135$$

Recall that merge sort always partitions its lists into halves. Thus, the depth of recursion for choosing the median-of-three will require $\frac{\ln(1/2)}{\ln(11/16)} \approx 1.8499$ or a 85 % increase in the depth of recursion. Choosing just a single random pivot will result in $\frac{\ln(1/2)}{\ln(3/4)} \approx 2.4094$ or 141 % increase in the depth of recursion.

**Question**: should we go to, for example, the median-of-*five*? Unfortunately, we choosing more points has diminishing returns. From order statistics, the average partition from

$$2\int_0^{\frac{1}{2}} x \cdot \left(180x^2(1-x)^2\right)dx = \frac{11}{32}$$

is $\frac{11}{32}n$ and $\frac{21}{32}n$ and still requires 65 % more recursive steps.

### 8.6.4 Basic Implementation

The easiest implementation of quicksort would be to create a new array and copy the entries smaller than the pivot to the start while copying the larger entries to the end of the array. For example, given the pivot 44 in the array

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 80 | 21 | 95 | 84 | 66 | 10 | 79 | **44** | 26 | 87 | 96 | 12 | 43 | 71 | 3 |

we would copy the first six entries into a new array as follows:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 21 | 10 | | | | | | | | | | 66 | 84 | 95 | 80 |

After all 14 entries other than 44, we have

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 21 | 10 | 26 | 12 | 43 | 3 | | 71 | 96 | 87 | 79 | 66 | 84 | 95 | 80 |

and we can now place 44 into the one remaining vacant location and we can recursively call quicksort on the entries 0 through 5 and 7 through 14.

The problem with this implementation is that it requires a new array, or $\Theta(n)$ additional memory. Our goal, however, is to perform sorts in-place.

### 8.6.5 In-place Implementation

Our goal is to simply partition the entries into those greater than and those less than the pivot. Consider the following algorithm:

Given the first, middle, and last entries in the list, assign the median to a local variable pivot, replace the first entry with the smallest of the three, and copy the largest of the three into the middle.

For example, given

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **80** | 21 | 95 | 84 | 66 | 10 | 79 | **44** | 26 | 87 | 96 | 12 | 43 | 71 | **3** |

we would end up with `pivot = 44` and

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **3** | 21 | 95 | 84 | 66 | 10 | 79 | **80** | 26 | 87 | 96 | 12 | 43 | 71 | |

Next, perform the following algorithm:

1. Search forward from the front in the list until you find an entry greater than the pivot, and
2. Search backward from the back of the list until you find an entry less than the pivot.

Having found these two, swap them. For example, given the above list, we would search forward until we find 95 and search back until we find 43:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 21 | **95** | 84 | 66 | 10 | 79 | 80 | 26 | 87 | 96 | 12 | **43** | 71 | |

Swapping these two corrects the inversion:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 21 | **43** | 84 | 66 | 10 | 79 | 80 | 26 | 87 | 96 | 12 | **95** | 71 | |

Repeating this process, we find 84 and 12 and swap these two:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 21 | 43 | **84** | 66 | 10 | 79 | 80 | 26 | 87 | 96 | **12** | 95 | 71 | |

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 21 | 43 | **12** | 66 | 10 | 79 | 80 | 26 | 87 | 96 | **84** | 95 | 71 | |

Again, searching forward and backward, we find 66 and 26 and swap these two:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 21 | 43 | 12 | **66** | 10 | 79 | 80 | **26** | 87 | 96 | 84 | 95 | 71 | |

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 21 | 43 | 12 | **26** | 10 | 79 | 80 | **66** | 87 | 96 | 84 | 95 | 71 | |

Notice that at this point, the first five points are all less than the pivot and the last six are all greater than the pivot. So we continue, searching forward and backward finding 79 and 10:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 21 | 43 | 12 | 26 | **10** | **79** | 80 | 66 | 87 | 96 | 84 | 95 | 71 | |

At this point, we note that the pair (10, 79) do not form an inversion: consequently, by our algorithm, all entries before 10 must be less than the pivot and all entries after 79 must be greater than the pivot. All we must do now is put the pivot back into the list. We can do this by copying 79 (being greater than the pivot) into the last position and replacing it with 44:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 21 | 43 | 12 | 26 | 10 | **44** | 80 | 66 | 87 | 96 | 84 | 95 | 71 | **79** |

Thus the list is partitioned between those entries greater than the pivot and those less than the pivot.

The next step is to recursively call quicksort on both halves. Like with merge sort, we will call insertion sort as soon as the size of a list is less than $N = 8$, so we would simply call insertion sort on the left half:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 10 | 12 | 21 | 26 | 43 | 44 | 80 | 66 | 87 | 96 | 84 | 95 | 71 | 79 |

Focusing on the right half, we choose the median of the entries in positions 7, 14, and (7 + 14)/2 = 10:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 10 | 12 | 21 | 26 | 43 | 44 | **80** | 66 | 87 | **96** | 84 | 95 | 71 | **79** |

The new `pivot = 80`, we copy 79 to position 7 and leave 96 in position 10:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 10 | 12 | 21 | 26 | 43 | 44 | **79** | 66 | 87 | **96** | 84 | 95 | 71 | |

As before, we search forward finding the first entry greater than the pivot and backward from the last entry until we find an entry less than the pivot, 87 and 71, and swap them:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 10 | 12 | 21 | 26 | 43 | 44 | 79 | 66 | **87** | 96 | 84 | 95 | **71** | |

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 10 | 12 | 21 | 26 | 43 | 44 | 79 | 66 | **71** | 96 | 84 | 95 | **87** | |

Again, searching forward and backward, we find 96 and 71:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 10 | 12 | 21 | 26 | 43 | 44 | 79 | 66 | **71** | **96** | 84 | 95 | 87 | |

As before, (71, 96) forms a sorted pair, so we are finished: copy 96 to the last position and replace it with the pivot 80:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 10 | 12 | 21 | 26 | 43 | 44 | 79 | 66 | 71 | **80** | 84 | 95 | 87 | **96** |

Now, we recursively call quicksort on both the left and right halves; however, both have a length less than $N = 8$, so we simply call insertion sort on both:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 10 | 12 | 21 | 26 | 43 | 44 | 66 | 71 | 79 | 80 | 84 | 87 | 95 | 96 |

You will now note that the entire list is sorted.

### 8.6.6 Memory Requirements

From ECE 222, you know that each function call places a frame on a stack. Thus, while it may seem that we require only local variables, the function-call stack will have as many entries as the number of recursions that are required. In the ideal case (always choosing the median), this will be approximately $\Theta(\lg(n))$; however, in the worst case, it could be as bad as $\Theta(n)$.

### 8.6.7 Comparing Heap, Merge, and Quicksort

Thus, we have the following table comparing these three algorithms:

| | **Run Time** | | **Memory Requirements** | |
|---|---|---|---|---|
| | **Average Case** | **Worst Case** | **Average Case** | **Worst Case** |
| Heap Sort | $\Theta(n \lg(n))$ | $\Theta(n \lg(n))$ | $\Theta(1)$ | $\Theta(1)$ |
| Merge Sort | $\Theta(n \lg(n))$ | $\Theta(n \lg(n))$ | $\Theta(n)$ | $\Theta(n)$ |
| Quicksort | $\Theta(n \lg(n))$ | $\Theta(n^2)$ | $\Theta(\ln(n))$ | $\Theta(n)$ |