Abstract

A discussion of *insertion, merge, and quick* sorting algorithms, specific to the way they've been implemented for this assignment (my P3 submission), and related to implementation, time complexity analysis, and a report of runtimes.

Insertion sort

1. **Explanation**: Loops through array (going toward index = size - 1) comparing side-by-side elements, if 2 elements need to be swapped, this triggers a helper function that traverses down the array (going toward index = 0) comparing the swapped element with each element to find it's place in the array.

2. **Time complexity analysis**:

   - **Best case (Big Ω)**: Assume the array is already sorted, then insertion sort loops through the array and is never triggered by 2 elements needing to be swapped. Thus, it's time complexity is Ω(n).
   - **Worst case (Big O)**: Assume the array is in reverse order, then for every item in the array the helper function is triggered and loops down the array all the way back to index 0. Thus, it's time complexity is O(n^2).
   - **Runtime tests**: In place of my own graph, I've borrowed from Paarth Lakhani.

Snippet 1: Chrono test

```
unordered (small N): 44 ms
unordered (large N): 575447 ms
almost_ordered (small N): 15 ms
almost_ordered (large N): 297923 ms
reverse_ordered (small N): 13 ms
reverese_ordered (large N): 218645 ms
already_ordered (small N): 5 ms
already_ordered (large N): 219520 ms
```

Figure 1: Runtimes

Merge sort

1. **Explanation**: Recursively divides array in 2 until it is no longer divisible. Then, begins to merge arrays together. It merges by comparing the first element of each array and putting the smaller of the 2 elements into a 'merged' array and simultaneously removing that element from the sub array (as to not place duplicates into the 'merged' array. It continues to do this until it's finished going up the call tree/stack.

2. **Time complexity analysis**:

   - **Best case(Big Ω)**: O(n log(n)).
   - **Worst case (Big O)**: O(n log(n)).
   - **Explanation**: What are the actions merge sort perform?

     i Divide function finds midpoint of array then sub array and so on. Finding the midpoint of an array is just indexing the array so takes O(1) time.

     ii Sort step in merge function "goes back up" queue created by divide function (i.e., recursively) sorts sub arrays of n/2 elements.

     iii Merge step in merge function is a for loop that copies merged[] array to original array[]. This takes O(1) time.

     How many times are these steps executed? For each level of the call tree, Level x calls merge on x sub arrays of length c*n/x each.

   - **Runtime tests**:

Snippet 2: Chrono test

```
unordered (small N): 5 ms
unordered (large N): 561363 ms
almost_ordered (small N): 14 ms
almost_ordered (large N): 238213 ms
reverse_ordered (small N): 3 ms
reverese_ordered (large N): 266735 ms
already_ordered (small N): 3 ms
already_ordered (large N): 237150 ms
```

Figure 2: Runtimes

Quicksort

1. **Explanation**: Idea is that in a sorted list of numbers, each number divides the list into two pieces, all numbers to left are smaller and all numbers to the right are bigger. And this is true for all numbers. Quick sort recursively splits array by everything less than pivot and everything greater than pivot. Pivot is chosen using median-of-3 technique (choose median of array[0], array[mid], and array[size-1]) to improve time complexity. Specifically, the median-of-3 technique counters the case of sorted, almost sorted, and reverse sorted arrays affecting the time complexity. As a result, the choice to use median-of-3 in this project will affect the runtime performance of quick sort.

2. **Time complexity analysis**:

   (a) **Best case(Big Ω)**: O(n log(n)).

   (b) **Worst case (Big O)**: O(n^2). This is the case for if the pivot chose is the largest or smallest value in the array because the pivot function will partition the original array into 1 empty array and 1 array with all the original elements. This defeats the purpose of the pivot function, which is to partition the array (ideally) closer to evenly.

Snippet 3: Solving for recursive function

```
T(n) = T(n-1) + n                          // T(n-1) = T(n-2) + (n-1)
     = T(n-2) + (n-1) + n
     = T(n-3) + (n-2) + (n-1) + n
     ...

     = T(1) + 2 + 3 + ... + (n-1) + n      // T(1) = 1
     = 1 + 2 + 3 + ... + (n-1) + n

     = n (n+1)
       ---------
          2
```

   (c) **Runtime tests**:

Snippet 4: Chrono test

```
unordered (small N): 15 ms
unordered (large N): 600863 ms
almost_ordered (small N): 5 ms
almost_ordered (large N): 269332 ms
reverse_ordered (small N): 13 ms
reverese_ordered (large N): 208264 ms
already_ordered (small N): 14 ms
already_ordered (large N): 205981 ms
```

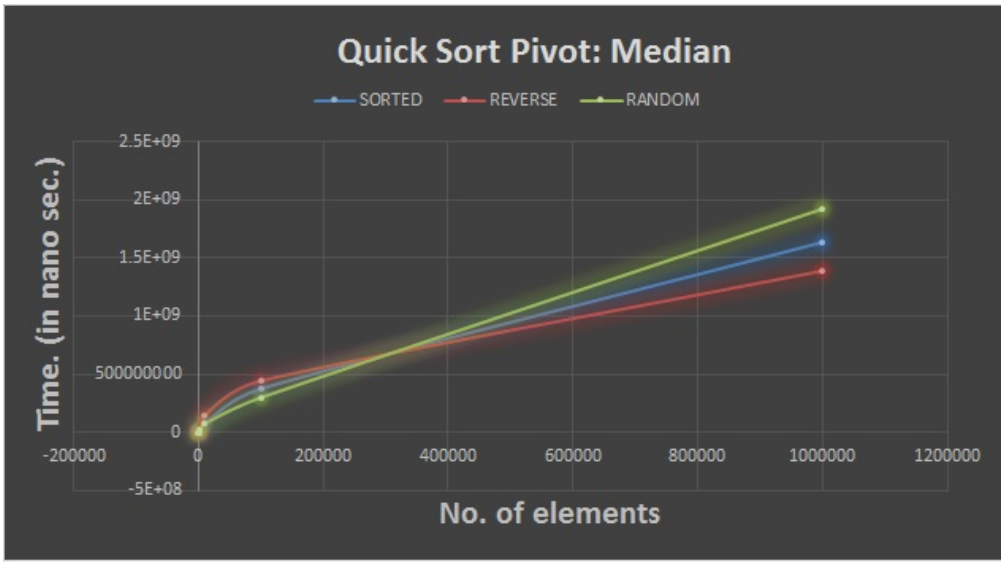

Figure 3: Runtimes

References

See README.md file uploaded with P3.