Hunter Lauder

CS240: CA4 Write-Up

4/26/2021

Professor Lewis

For section 3A, I analyzed the placement of the ticker.tick() in useTicker.cpp and nlogn.cpp. It is evident that the placement of these ticker.tick() calls is vital to accurately depict the runtime complexity of a function of code. With this being said, we can see, from these given examples, that the ticker.tick() should be placed within the for loop or within the inner loop if it is a nested for loop; this is known as the “critical” region of code.

For section 3B, I implemented this knowledge by instantiating a Ticker in Sorter.cpp and using the reset() and tick() Ticker member functions to count the operations of each search method. For linear search, I simply put ticker.reset() at the top of the function and put ticker.tick() within the for loop which iterates through each array member. The operation count returned equivalent to the big O of a linear search when the searched value was not found, O(n); for instance, given N=100, the worst-case operation count (value not found) returned 100. Additionally, I did this same procedure for the binary search, except for this search I needed to place the ticker.tick() at the top of the recursive function within the first if statement that checks if the r value is >= 1. I placed this here instead of at the very top (before this if statement) because of the calculations that I am about to explain. The operation count returned equivalent to the big O of a linear search, O(log(n)); for instance, given N=100, the worst-case operation count (value not found) returned was 7. This makes logical sense since $log_2(100)$= 6.6439; meanwhile, if the ticker. tick() was placed at the very top of the recursive function, as aforementioned, the worst-case operation count would have returned 8 which is less consistent with the calculation.

For section 3C, I did this same general procedure for each sorting method. However, it should be noted that for each of these sorting methods, ticker.tick() calls are placed in parts of code that make comparisons since these methods are comparison-based sorts. For insertion sort, I simply put ticker.reset() at the top of the function and put the ticker.tick() within the while loop which is within a for loop. It makes sense that the ticker.tick() should be placed within this while loop since it makes comparisons and is embedded within a for loop. The operation count returned lies within the best and worse case time complexity of an insertion sort, Ω(n) and O(n^2); for instance, given N=250, the operation count returned was 15831. This makes logical sense since Ω(250) = 250 and O(250^2) = 62500. This remains true for all other values of N, several examples are shown below:

```
insertionSortMap
10 --> 13
50 --> 536
250 --> 15831
1250 --> 389816
6250 --> 10022725
```

For selection sort, I placed ticker.reset() at the top of the function and ticker.tick() in the inner loop of the nested for loop where a comparison is being made. The operation count returned should be approximately equal to the time complexity of a selection sort, Ω(n^2)/O(n^2); for instance, given N=10, the operation count returned was 45. Although this is not quite equal, this is expected because big O is always multiplied by some constant. In this case, it is approximately 0.5. Here are several other operation counts for other N values:

```
selectionSortMap
10 --> 45
50 --> 1225
250 --> 31125
1250 --> 780625
6250 --> 19528125
```

For quick sort, I placed ticker.reset() at the top of the main function and ticker.tick() in the for loop of the split function (quickSortA). The operation count returned lies within the best and worse case time complexity of a quick sort, Ω(n log(n)) and O(n^2); for instance, given N=250, the operation count returned was 2203. This makes logical sense since Ω$(250 * log_2(250))$= 1991.4461 and O(250^2) = 62500. Here are several other operation counts for other N values:

```
quickSortMap
10 --> 30
50 --> 295
250 --> 2203
1250 --> 15362
6250 --> 96349
```

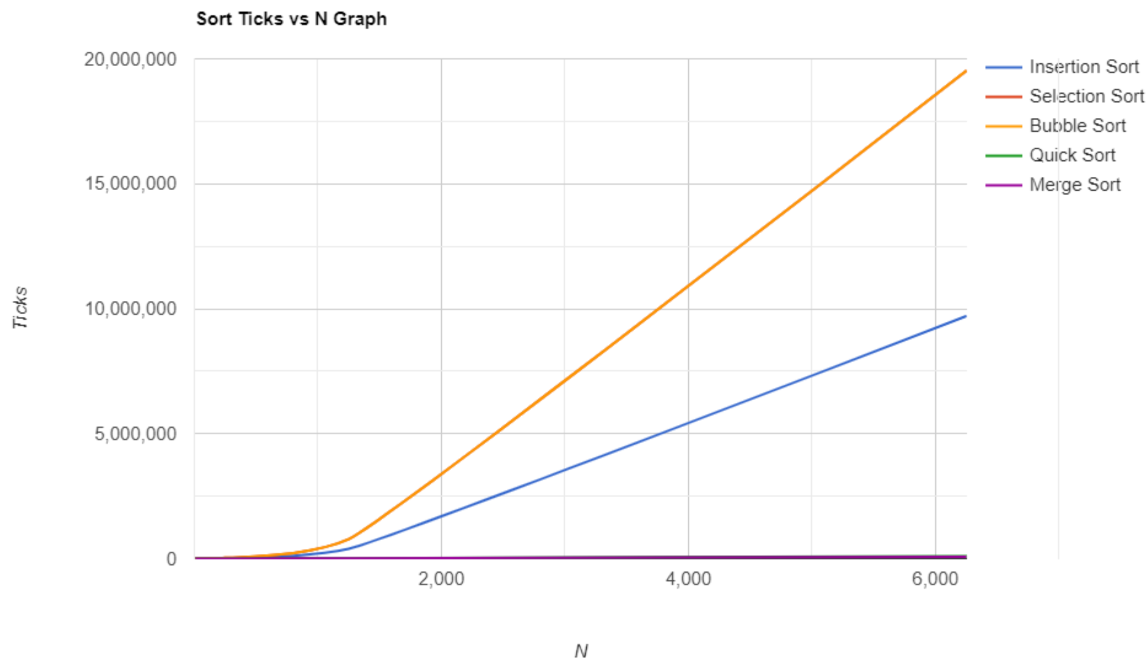
For merge sort, I placed a ticker.reset() at the top of the main function and a ticker.tick() in each of the three while loops of the merge function (mergeSortA). The operation count returned should be approximately equal to the time complexity of a quick sort, Ω(n log(n))/O(n log(n)); for instance, given N=250, the operation count returned was 1994. This makes logical sense since $250 * log_2(250)$= 1991.4461 and so this value is quite close to this (only varying a small amount which is expected). Here are several other operation counts for other N values:

```
mergeSortMap
10 --> 34
50 --> 286
250 --> 1994
1250 --> 12952
6250 --> 79308
```

For bubble sort, I placed a ticker.reset() and ticker.tick() in the inner loop of the nested for loop where a comparison is being made. The operation count returned lies within the best and worse case time complexity of a quick sort, Ω(n) and O(n^2); for instance, given N=250, the operation count returned was 31125. This makes logical sense since Ω(250)= 250 and O(250^2) = 62500. Here are several other operation counts for other N values:

```
bubbleSortMap
10 --> 45
50 --> 1225
250 --> 31125
1250 --> 780625
6250 --> 19528125
```

As shown, I used maps in the program to output these displays of the ticks.



For section 3D, we are looking at the various differences between these operation counts for these sorts/searches when these are called upon shuffled, already sorted and reverse sorted arrays. When comparing the ticks of these for reverse sorted arrays, I notice that there is no difference in ticks for merge sort, bubble sort, or selection sort. However, there are noticeable differences for quick sort and insertion sort. For instance, here is the quick sort of a shuffled array vs the quick sort of a reverse sorted array:

```
quickSortMap
10 --> 30
50 --> 262
250 --> 2036
1250 --> 15484
6250 --> 93018
```

(shuffled)

```
quickSortMap
10 --> 54
50 --> 1274
250 --> 31240
1250 --> 762590
6250 --> 16615175
```

(reversed)

Here is the insertion sort of a shuffled array vs the insertion sort of a reverse sorted array:

```
insertionSortMap
10 --> 24
50 --> 631
250 --> 15634
1250 --> 385505
6250 --> 9710264
```
(shuffled)

```
insertionSortMap
10 --> 45
50 --> 1224
250 --> 31123
1250 --> 780540
6250 --> 19526135
```
(reversed)

When comparing the ticks of these for already sorted arrays, I notice that there is no difference in ticks for merge sort, bubble sort, or selection sort. However, there are noticeable differences for quick sort and insertion sort.

```
quickSortMap
10 --> 30
50 --> 262
250 --> 2036
1250 --> 15484
6250 --> 93018
```
(shuffled)

```
quickSortMap
10 --> 54
50 --> 1274
250 --> 31348
1250 --> 730897
6250 --> 14746504
```
(already sorted)

```
insertionSortMap
10 --> 24
50 --> 631
250 --> 15634
1250 --> 385505
6250 --> 9710264
```
(shuffled)

```
insertionSortMap
10 --> 0
50 --> 0
250 --> 0
1250 --> 0
6250 --> 0
```
(already sorted)

For binary search, the search already requires the data to be sorted ascendingly; if you were to reverse the data, you would have to reverse the code of the binary search itself. For linear search, there would be no difference in ticks.