To run my code, I seperated the quicksort and the rest. I had some issues getting it to work out. To run, run quickSort.java, and Timer.java. Graph was made using python. To run, python3 makegraph.py

Andrew Durkiewicz

**Problem 1)**

Let A = [22,91,24,78,6,89,98,99]

**part A (selection sort)**

- [22,91,24,78,6,89,98,99]
- sorted portion = [], unsorted portion = [22,91,24,78,6,89,98,99]
  - We need to find the minimum unsorted element and put it at the beginning of the sorted list.
  - First look at [22] and compare it to other values until we find a value which is larger than 22.
  - When we arrive at comparing [22] to [6], we find that 6<22. Thus we swap the two and that becomes our sorted algorithm.
- Updated: sorted portion = [6,91,24,78,22,89,98,99]
  - Now we do the same with [91]. Immediately we pick up on the fact that 91 > 24. So those are swapped:

- updated sorted portion = [6,24,91,78,22,89,98,99]
  - now we pick [24] and compare it. When reaching 22, these are swapped as 22< 24:
- updated sorted portion = [6,22,91,78,24,89,98,99]
- The pattern continues, I will simplify and continue with the continued list items in red are compared and swapped, while blue is post swap:
  [6,22,91,78,24,89,98,99]
  [6,22,78,91,24,89,98,99]
  [6,22,78,91,24,89,98,99]
  [6,22,24,91,78,89,98,99]
  [6,22,24,91,78,89,98,99]
  [6,22,24,78,91,89,98,99]
  [6,22,24,78,91,89,98,99]
  [6,22,24,78,89,91,98,99] (The list is sorted.)
  What I dont really show here are the comparisons which are made repeatedly. We have to check every successive index to make sure there doesn't need to be a swap. This makes this sorting method very inefficient.

**Part B (Merge Sort)**

This is done by comparing 'pairs', and soring them. Then we take advantage of this:
[22,91,24,78,6,89,98,99]
becomes 4 list of 2:
[22,91] [24,78] [6,89] [98,99]
For each of these, the first index is compared to the second. For list [A,B] if A>B then they are swapped:
In our case, there is no swap.
Now we can these into two list of size four. These are sorted using the selection sort. So, now we have [22,24,78,91] and [6,89,98,99]

These are furthur sorted by pairs in increasing index values. If a swap occurs, that is compared with the next index from its pair:

[22,6] → [6,22] (swap) so we compare 22 to 24 (no swap)
[6,22, 24,78, 89, 91, 98, 99]
Now our list is sorted

**Part C (quicksort)**
[22,91,24,78,6,89,98,99]
pivot is [22]
For any element less than our pivot, that element is brought to the front. This shifts the position of where 22 was. So,
[ ,91,24,78,6,89,98,99]
[22] pivot

Since 6<22, it is brought to the front:
[6, , 91,24,78, 89,98,99]
For all furthur elements 22 is smaller so it goes into its place and now 91 becomes the pivot:
[6,22, , 24,78, 89, 98, 99]
91> 24 so that takes the blank spot and the empty spot shift over. This happens for 89 and 78 as well:
[6,22,24,78, 89, 91,98,99] Now this is sorted.

**Problem 2)**
a) Time complexity of an algorithm demonstrates the efficiency of a particular algorithm. It takes advantage of the fact that computer instructions work off a very specific clock. The frequency of calculations that a computer can make gives us insight into the amount of time a computer will take to solve an algorightm. Time complexity allows us to use this clock to determine time to solve an algorithm vs the array length that is given to it.

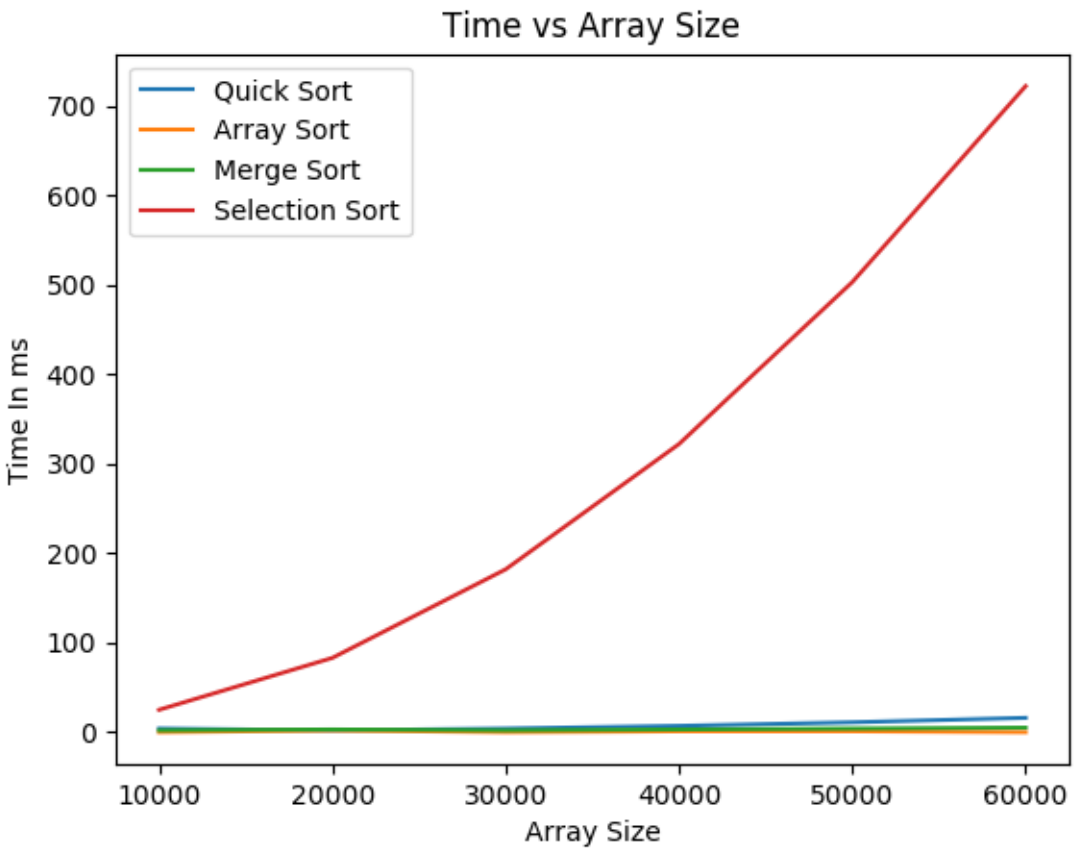
b) Space compexity is the amount of space that an algorithm needs with respect to the input size. So, this becomes an issue if you input two large arrays and take their cross product. This can produce a much larger array which consumes memory.

c) What we notice is that the process follows C(n/(2^e)) = 1. e, is the exponenent, and is directly proportional to the height of the tree. C is the ceiling function. It rounds up since you cant have a length that isn't an integer. Thus, using our log functions we find that C[log(n)] = e*[C(log(2))]. C(log(2)) = 1 so the final solution since we know that n is a large value and log(n) > 1, is that m= log(n)
D)
Quick:
Best: O(nlog(n))
Worst: O(n^2)
average: O(nlog(n))

Merge:
Best: O(nlog(n))
Worst: O(nlog(n))
Average: O(nlog(n))

Select Sort:
Best O(n^2)
Worst O(n^2)
average O(n^2)

**Problem 3)**
Please note that I did this with python and made a graph using matplot lib.




| Array Size | Selection Sort | Merge Sort | QuickSort | Arrays.Sort |
|---|---|---|---|---|
| 10000 | 25 | 2 | 4 | 0 |
| 20000 | 83 | 3 | 2 | 2 |
| 30000 | 182 | 2 | 4 | 0 |
| 40000 | 322 | 3 | 7 | 1 |
| 50000 | 503 | 4 | 11 | 1 |
| 60000 | 722 | 5 | 16 | 0 |

Time in ms