```
/Library/Java/JavaVirtualMachines/jdk1.8
elapsed time for quick sort 0.056
elapsed time for merge sort 0.03

Process finished with exit code 0
```

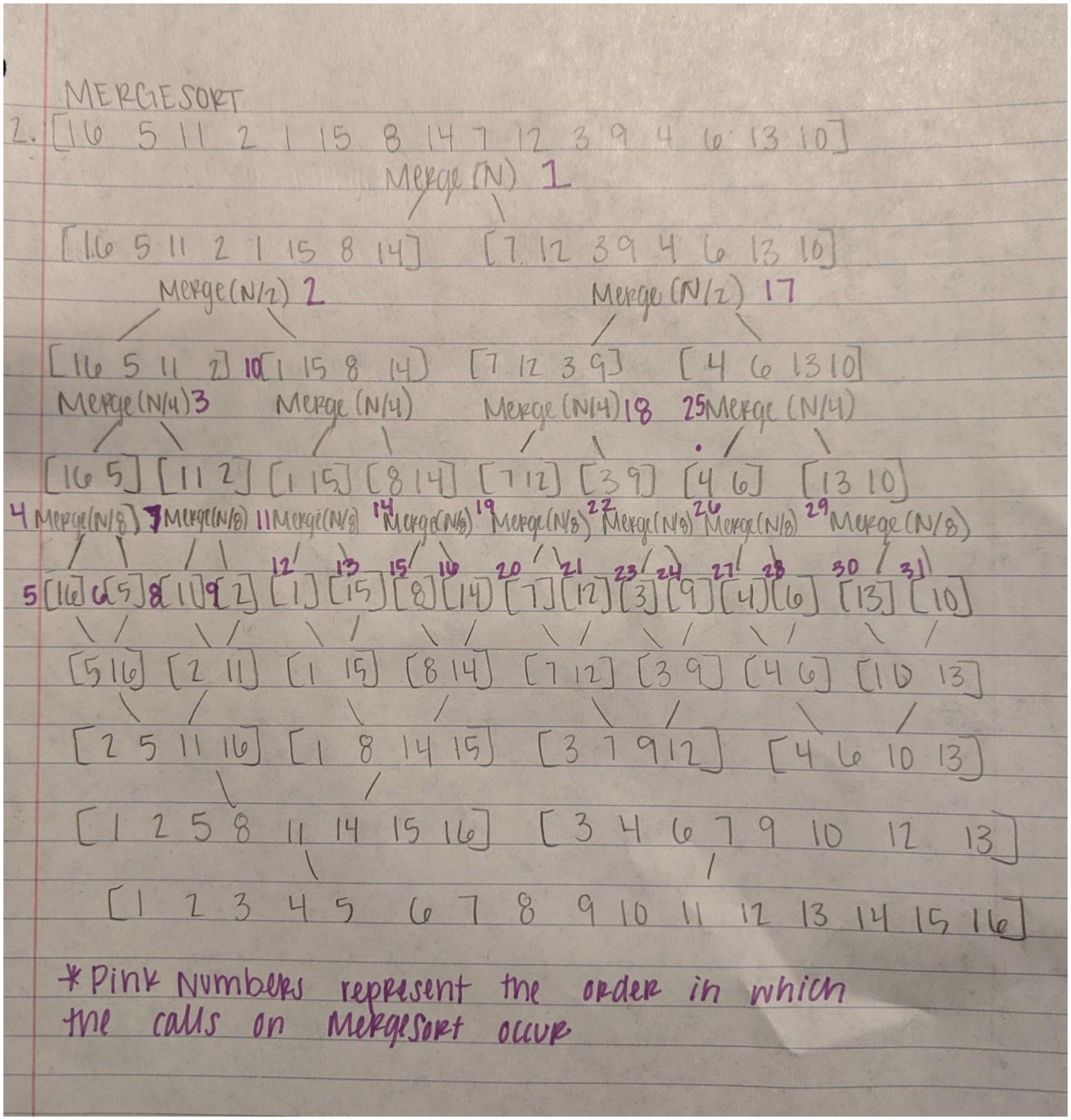
1. The QuickSort took more time (0.056 seconds), while the MergeSort took (0.03 seconds). This is due to the fact that MergeSort compares data that is almost sorted better than QuickSort. MergeSort can guarantee a BigO of *NlogN* whereas QuickSort could have a worst case of ½ N^2. MergeSort ran faster even though QuickSort runs in place and MergeSort does not.

Running time = a * O(N) - using average BigO
QuickSort:
0.056 = a * O(NLogN)
0.056 = a * 100,000*Log(100,000)
0.056/ (100,000*Log(100,000)) = a
1.12×10^{-7} = a

MergeSort:
0.03 = a * O(NLogN)
0.03 = a * 100,000* Log(100,000)
0.03/(100,000 * Log(100,000)) = a
6×10^{-8} = a

If you take the *a* value you get from MergeSort and divide by the *a* value for QuickSort you can see that the running time for MergeSort is related to the running time for QuickSort by a factor of (approximately) 0.5357. Clearly in the specific set of random numbers that I had for this run of both sorts MergeSort was faster than QuickSort.

2.


MERGESORT
2. [16 5 11 2 1 15 8 14 7 12 3 9 4 6 13 10]
Merge (N) 1
[16 5 11 2 1 15 8 14] [7 12 3 9 4 6 13 10]
Merge (N/2) 2
Merge (N/2) 17
[16 5 11 2] [1 15 8 14] [7 12 3 9] [4 6 13 10]
Merge (N/4) 3
Merge (N/4) 10
Merge (N/4) 18
Merge (N/4) 25
[16 5] [11 2] [1 15] [8 14] [7 12] [3 9] [4 6] [13 10]
4 Merge (N/8) 7 Merge (N/8) 11 Merge (N/8) 14 Merge (N/8) 19 Merge (N/8) 22 Merge (N/8) 26 Merge (N/8) 29 Merge (N/8)
5 [16] [5] [11] [2] [1] [15] [8] [14] [7] [12] [3] [9] [4] [6] [13] [10]
12 13 15 16 20 21 23 24 27 28 30 31
[5 16] [2 11] [1 15] [8 14] [7 12] [3 9] [4 6] [10 13]
[2 5 11 16] [1 8 14 15] [3 7 9 12] [4 6 10 13]
[1 2 5 8 11 14 15 16] [3 4 6 7 9 10 12 13]
[1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16]
* Pink Numbers represent the order in which the calls on MergeSort occur

QUICKSORT

Starting array [16 5 11 2 1 15 8 14 7 12 3 9 4 6 13 10]
(v = 16, i = 5, j = 10)

sort() is called
v (the partition) = 16; i = 5; j = 10
v & j swap when j = 10
new array [10 5 11 2 1 15 8 14 7 12 3 9 4 6 13 16]
sort() is called
v = 10; i = 5; j = 10
exch(11,6); exch(15,4); exch(14,9); exch(12,3); exch(3,10)

new arr = [3 5 6 2 1 4 8 9 7 10 12 14 15 11 13 16]
sort() is called on left of 10
v = 3; i = 5; j = 7
exch(5,1); exch(6,2); exch(2,3)

new arr = [2 1 3 6 5 4 8 9 7 10 12 14 15 11 13 16]
sort() is called on left of 3
v = 2; i & j = 1
exch(2,1)

new arr = [1 2 3 6 5 4 8 9 7 10 12 14 15 11 13 16]
sort() called on left of 2; then sort() called on right of 3
v = 6; i = 5; j = 7
exch(4,6)

new arr = [1 2 3 4 5 6 8 9 7 10 12 14 15 11 13 16]
sort() is called on left of 6; then called on right of 4
no exchanges made. sort() is then called on right of 6
v = 8; i = 9; j = 7
exch(9,7); exch(8,7)

new arr = [1 2 3 4 5 6 7 8 9 10 12 14 15 11 13 16]
sort() called on right of 8 then left of 8
sort() called on right of 10
v = 12 ; i = 14 ; j = 13
exch(14, 11) ; exch(12, 11)

new arr = [1 2 3 4 5 6 7 8 9 10 11 12 15 14 13 16]
sort() called on left of 12 then sort() called on right of 12
v = 15 ; i = 14 ; j = 13
exch(15, 13)

new arr = [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16]
sort() called on right of 15
v = 13 ; i ≯ j = 14
sort() called on left of 13
v ≯ i ≯ j = 14

Finally Sorted Array

[1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16]

* = means previous partition that is properly sorted